C-DAC Four Days Technology Workshop

ON

Hybrid Computing – Coprocessors/Accelerators
Power-Aware Computing – Performance of Application Kernels

hyPACK-2013
(Mode-4 : GPUs)

Classroom lecture :

An Overview of GPGPUs /GPU Computing

Venue : CMSD, UoHYD ; Date : October 15-18, 2013

An Overview of GPGPUs /GPU Computing

Lecture Outline

Following topics will be discussed

- ❖ An Overview of GPUs – Past Developments – GPU Prog.
- ❖ An overview of CUDA enabled NVIDIA GPUs – OpenACC CUDA 5.5 &
- ❖ An Overview of AMD GPUs – Programming - OpenCL
- ❖ An Overview of OpenCL – Heterogeneous Prog.

Source : References given in the presentation

Part-I (A)

An Overview of GPUs / Past –
GPU Programming on GPUs

Source & Acknowledgements : NVIDIA, AMD, References

Overview

- ❖ What is GPU ? Graphics Pipeline
- ❖ GPU Architecture
- ❖ GPU Programming – OpenGL, DirectX, NVIDIA (CUDA), AMD (Brook+)
- ❖ Rendering pipeline on current GPUs
- ❖ Low-level languages
  - ➢ Vertex programming
  - ➢ Fragment programming
- ❖ High-level shading languages
- ❖ GPU Architecture - Graphics Programming

Source : References

What is GPU ?

- From Wikipedia : A specialized processor efficient at manipulating and displaying computer graphics

- 2D primitive support – bit block transfers

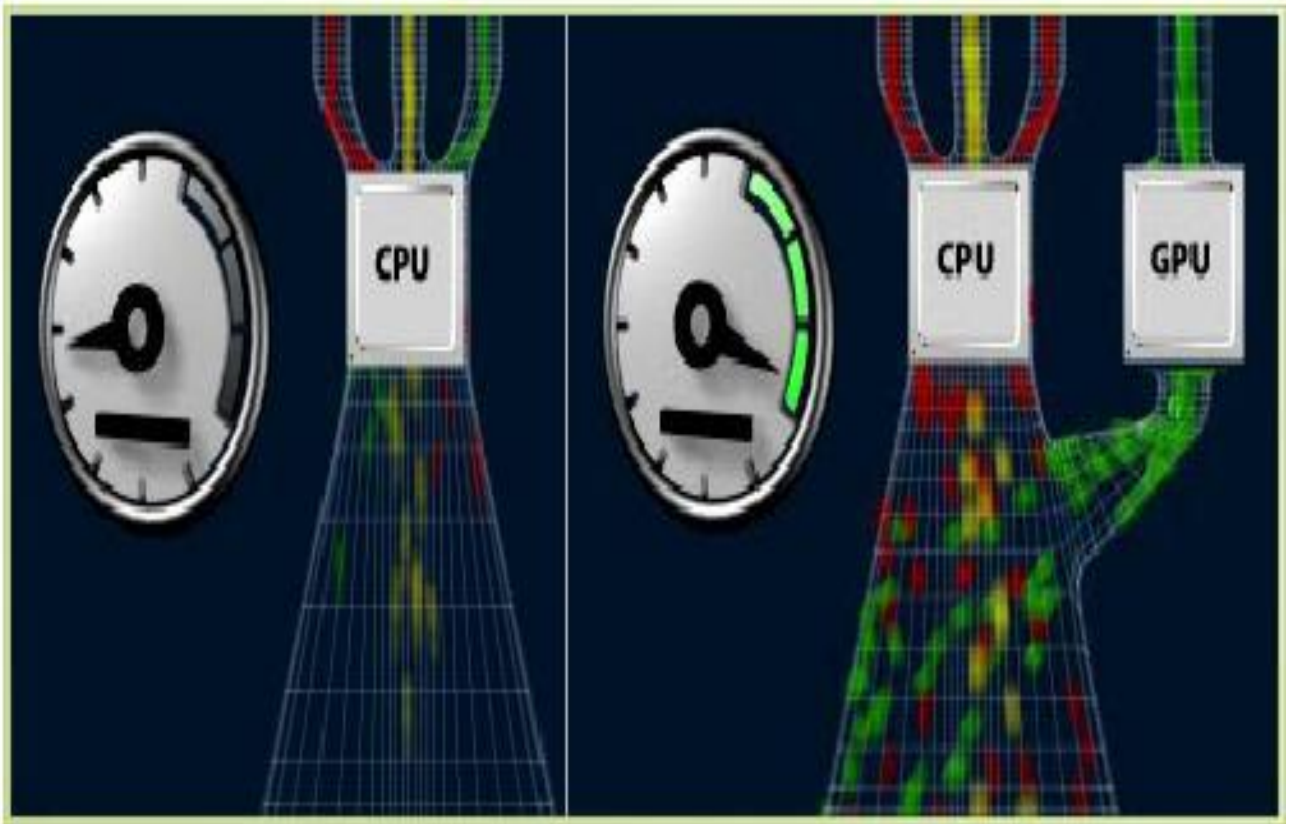
- Some might have video support

- And of course 3D support (a topic at the heart of this presentation)

- GPUs are optimized for raster graphics

Source : References

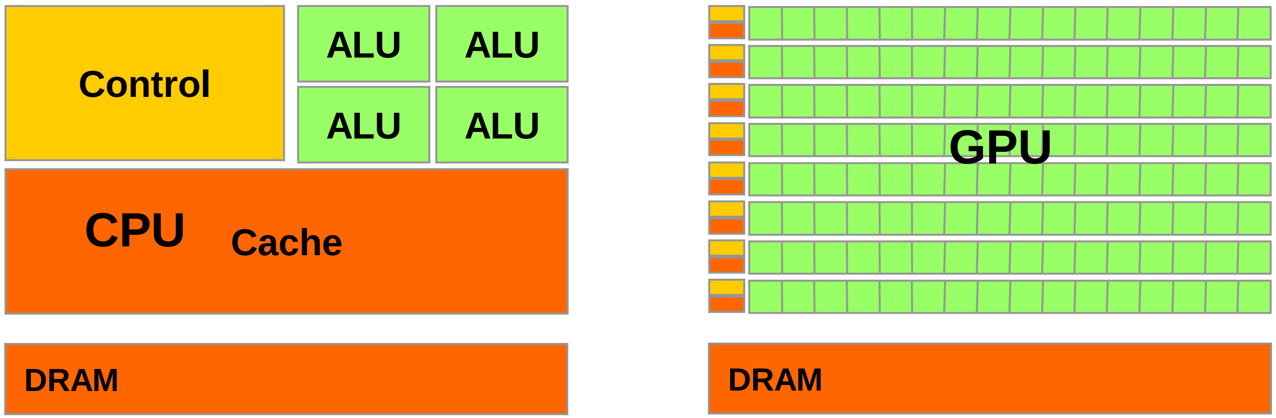
What is GPU ?



Without GPU **With GPU**

Source : References given in the presentation

What is GPU ?



- ❖ The GPU is specialized for compute-intensive, highly data parallel computation (exactly what graphics rendering is about)
  - ✓ So, more transistors can be devoted to data processing rather than data caching and flow control
- ❖ Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads
- ❖ GPU threads are extremely lightweight
- ❖ GPU needs 1000s of threads for full efficiency

What is GPU ?

- ❖ Graphics Processing Unit
- ❖ GPU also occasionally called visual processing unit or VPU
- ❖ It's a dedicated graphics rendering device for a personal computer, workstation, or game console.
- ❖ GPU is viewed as compute device that :
    - Is a coprocessor to CPU or host machine
    - Has its own DRAM (on the device)
    - Runs many threads in parallel
- ❖ Thus GPU is dedicated super-threaded, massively data parallel co-processor

GPGPU

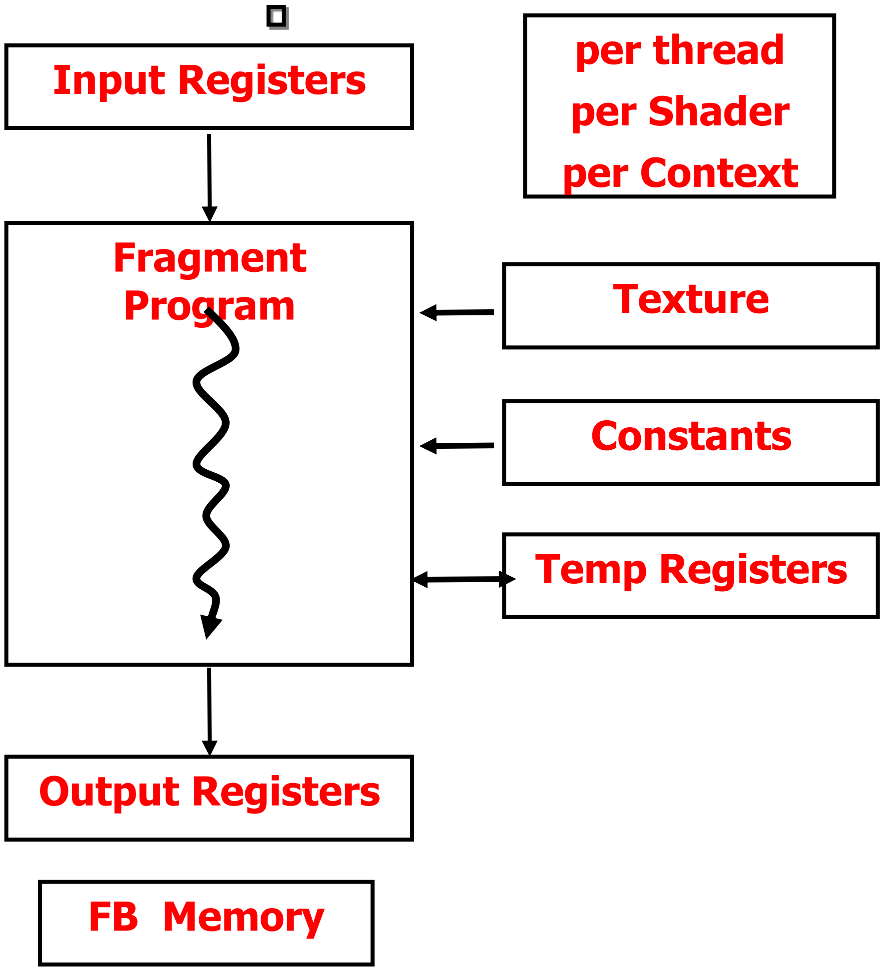
- ❖ Look at GPU as a fast SIMD processor
- ❖ It is a specialized processor, so not all programs can be run
- ❖ Example computational programs – FFT,
- ❖ Cryptography, Ray Tracing, Segmentation and even sound processing!

Source : References given in the presentation

What is GPU ?

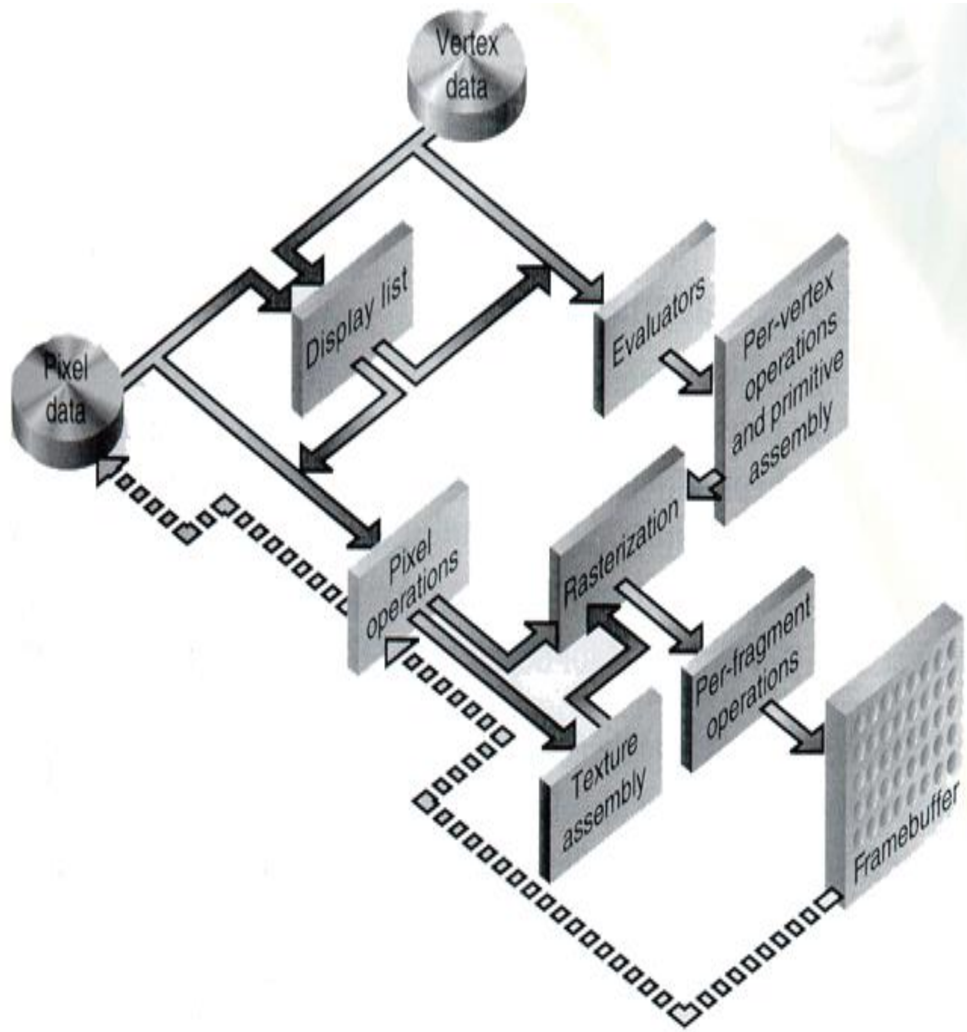
History

- Dealing complex with Graphics API
- Sequential Flow of Execution
- Limited Communication



The Graphics pipeline

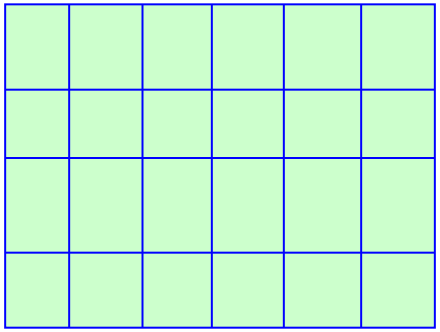
Application → Command → Geometry → Rasterization → Texture → Fragment → Display



3D Graphics Software Interfaces
OpenGL (v2.0 as of now)

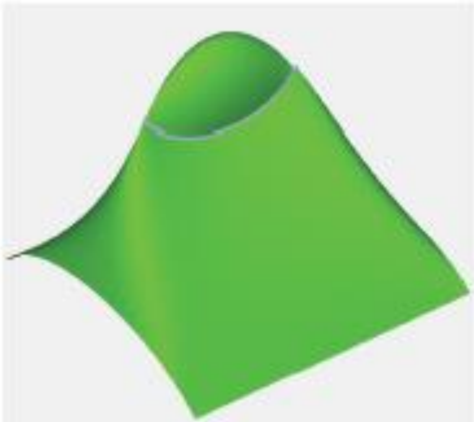
- Low level
- Specification not an API
- Crossplatform implementations
- Popular with some games
- A simple seq of opengl instr (in C)

```
glClearColor(0.0,0.0,0.0,0.0);
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1.0,1.0,1.0);
glOrtho(0.0,1.0,0.0,1.0,-1.0,1.0);
        glBegin(GL_POLYGON);
        glVertex(0.25,0.25,0.0);
        glVertex(0.75,0.25,0.0);
        glVertex(0.75,0.75,0.0);
        glVertex(0.25,0.75,0.0);
glEnd();
```



Source : References

Geometry Processing



Self intersections

Dynamic silhouette refinement

Algebraic Geometry

Preparation of FEM grids

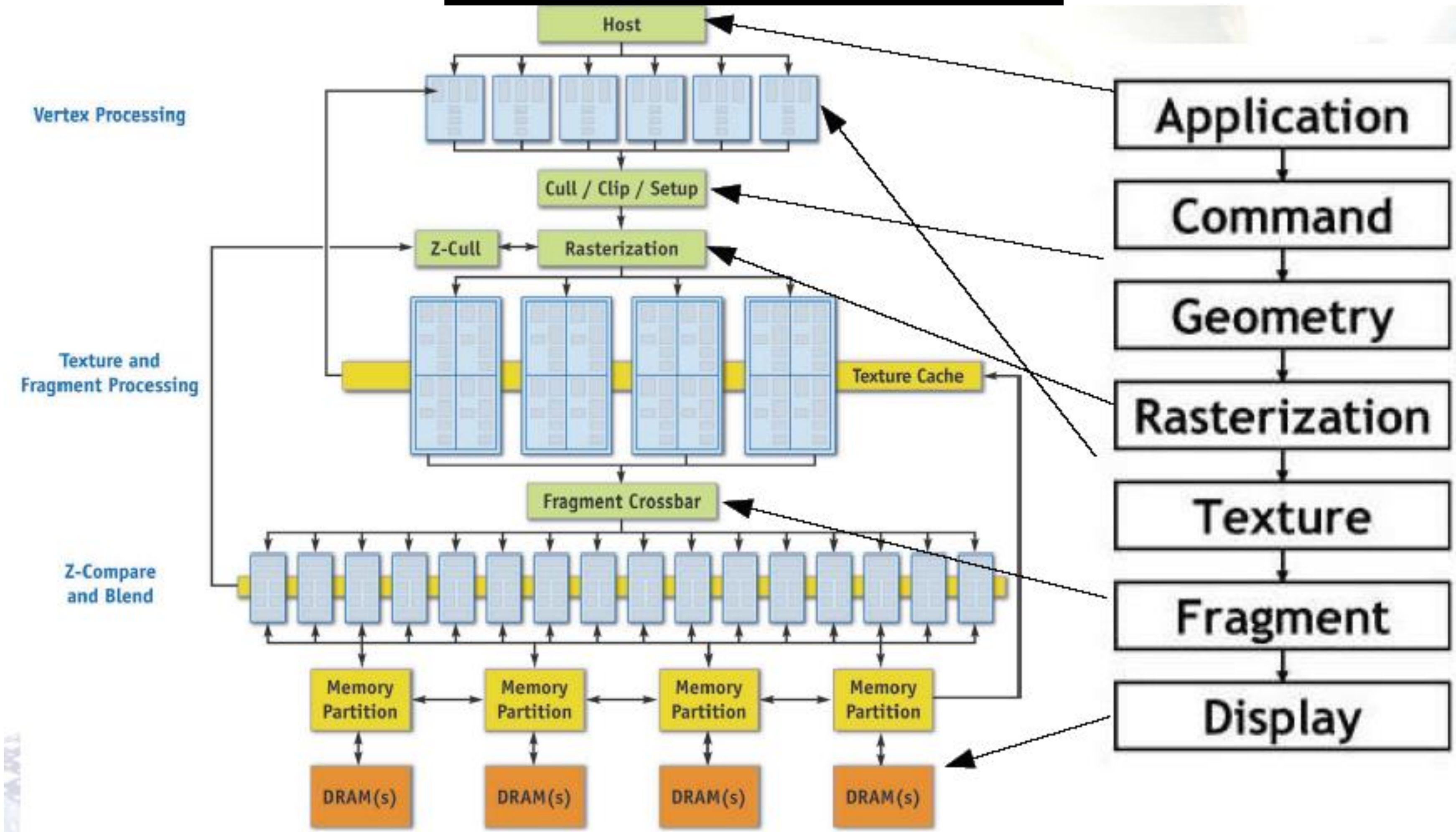
Source : References

NVIDIA GeForce 6800 General Info

- Impressive performance stats
  - 600 Million vertices/s
  - 6.4 billion texels/s
  - 12.8 billion pixels/s rendering z/stencil only
  - 64 pixels per clock cycle early z-cull (reject rate)
- Riva series (1st DirectX compatible)
  - Riva 128, Riva TNT, Riva TNT2
- GeForce Series
  - GeForce 256, GeForce 3 (DirectX 8), GeForce FX, GeForce 6 series

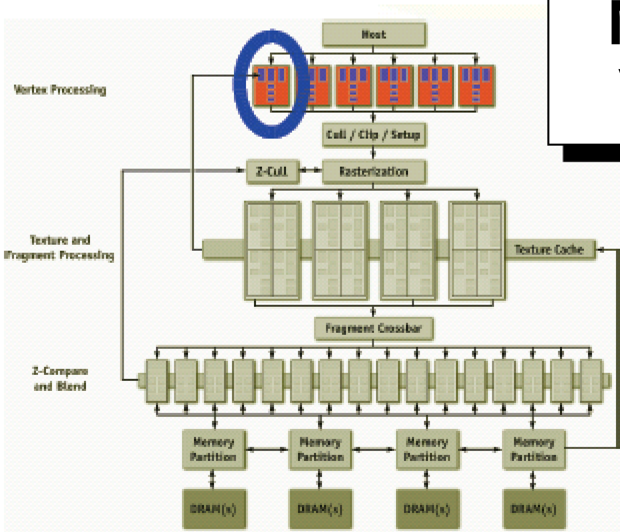
Source : References

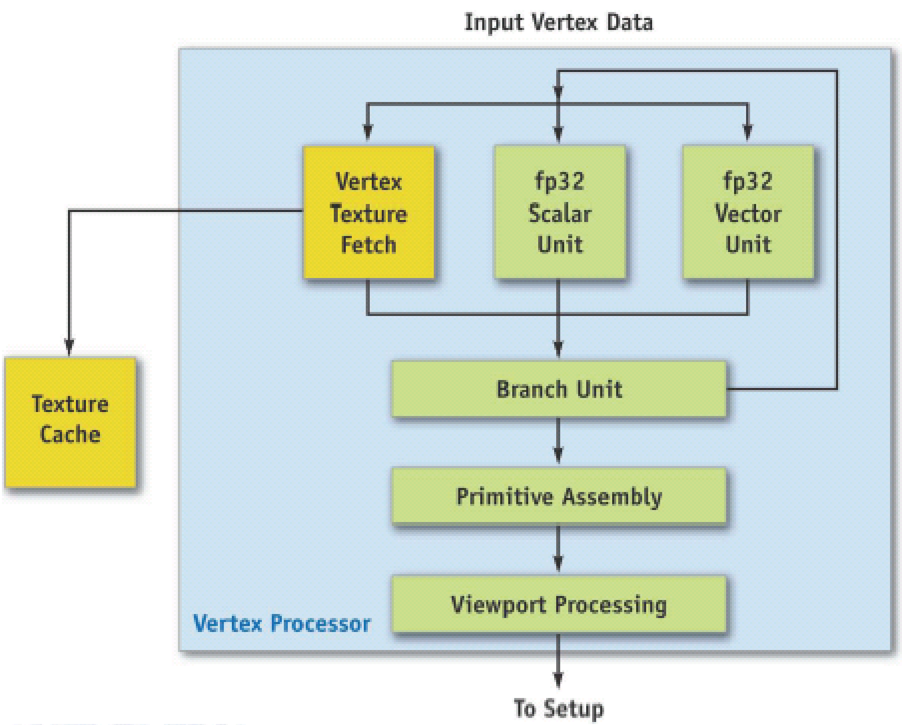
NVIDIA GeForce 6800 Block Diagram



Source : References

NVIDIA GeForce 6800
Vertex Processor (or vertex shader)





- ❖ Allow shader to be applied to each vertex
- ❖ Transformation and other per vertex ops
- ❖ Allow vertex shader to fetch texture data (6 series only)

Source : References

GPU from comp arch perspective
Processing units

- Focus on Floating point math
- fp32 and fp16 precision support for intermediate calculations

- 6 four-wide fp32 vector MADs/clock in shaders and 1 scalar multifunction op

- 16 four-wide fp32 vector MADs/clock in frag-proc plus 16 four-wide fp32 MULs

- Dedicated fp16 normalization hardware

Source : References

GPU from comp arch perspective Memory

- Use dedicated but standard memory architectures (eg DRAM)
- Multiple small independent memory partitions for improved latency
- Memory used to store buffers and optionally textures
- In low-end system (Intel 855GM) system memory is shared as the Graphics memory

GPU from comp arch perspective Memory

- GPU interfaces with the CPU using fast buses like AGP and PCI Express

- Port speeds
  - PCI express upto 8GB/sec (4 + 4)
  - Practically upto (3.2 + 3.2)
  - AGP upto 2 GB/sec (for 8x AGP)

- Such bus speeds are important because textures and vertex data needs to come from CPU to GPU (after that it's the internal GPU bandwidth that matters)

Source : References given in the presentation

GPU from comp arch perspective Memory

- Texture caches (2 level)
  - Shared between vertex procs and fragment procs
  - Cache processed/filtered textures

- Vertex caches
  - cache processed and unprocessed vertexes
  - improve computation and fetch performance

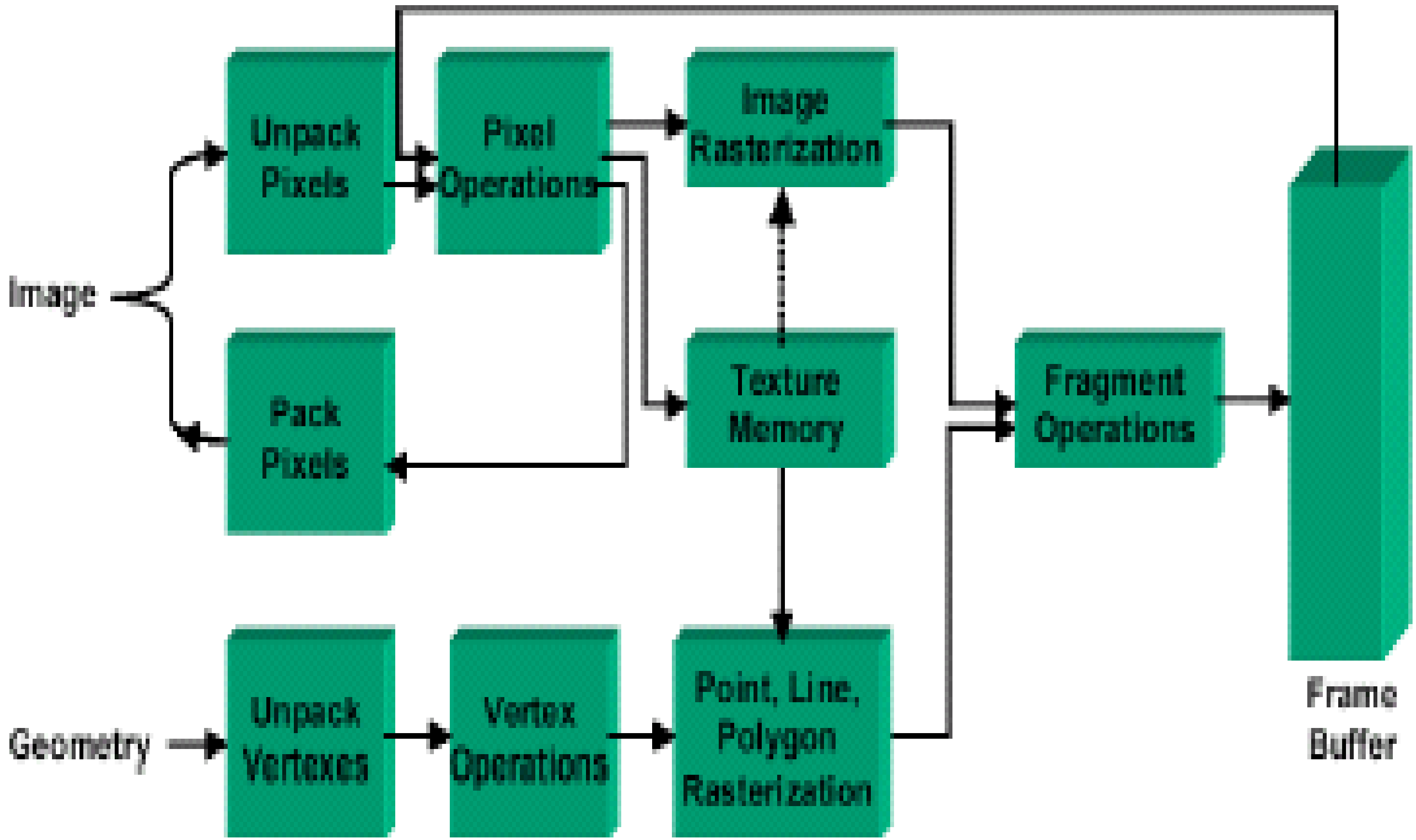
- Z and buffer cache and write queues

3D Graphics Software Interfaces Direct 3D (v9.0 as of now)

- High level
- 3D API – part of DirectX
- Very popular in the gaming industry
- Microsoft platforms only

Source : References given in the presentation

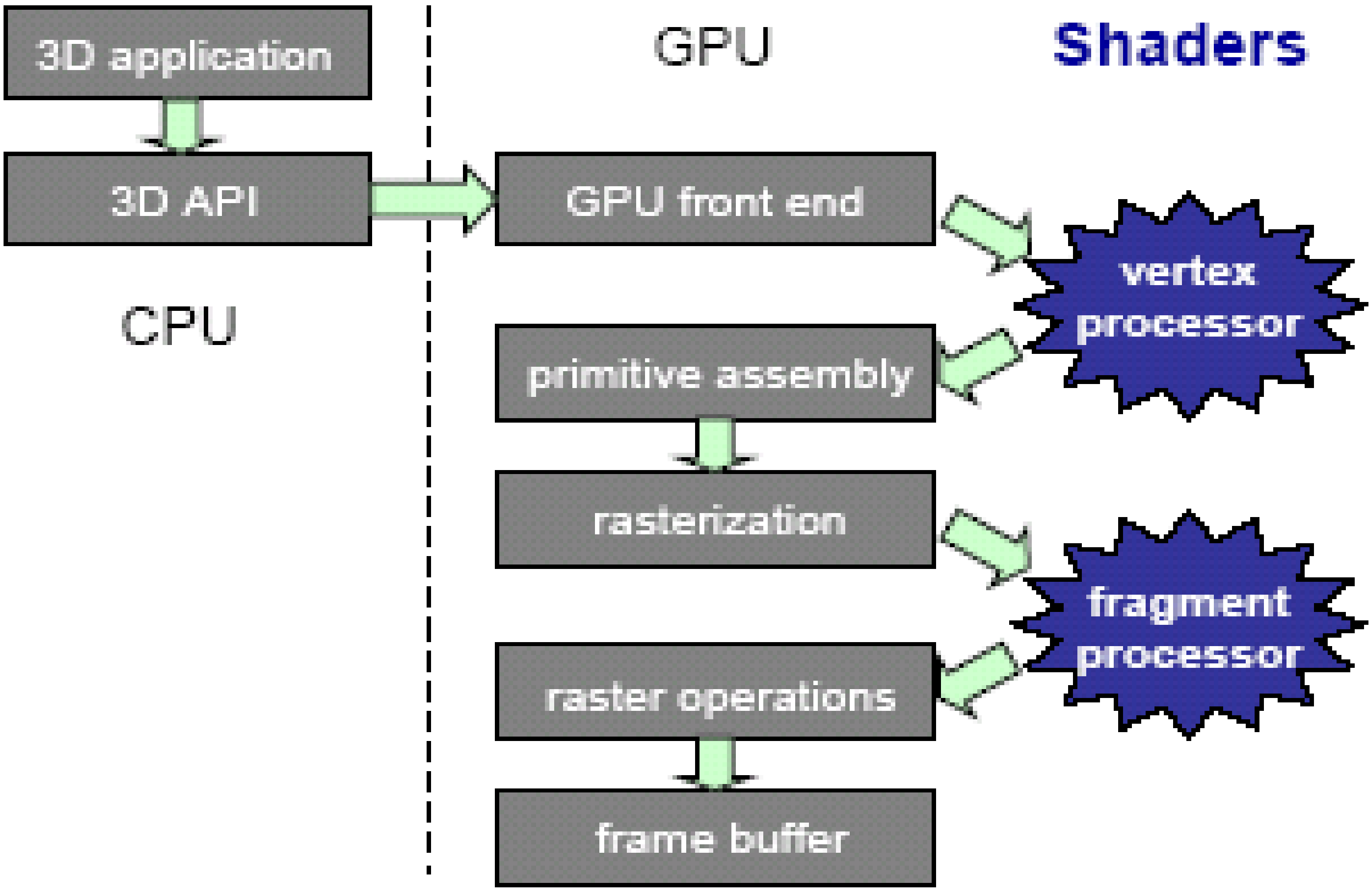
Traditional OpenGL Pipeline



Programmable Pipeline

- Most parts of the rendering pipeline can be programmed
- Shading programs to change hardware behavior
  - Transform and lighting:
    vertex shaders / vertex programs
  - Fragment processing:
    pixel shaders / fragment programs
- History: from fixed-function pipeline to configurable pipeline
  - Steps towards programmability

Programmable Pipeline



GPU - Issues

- How are vertex and pixel shaders specified?
  - Low-level, assembler-like
  - High-level language

- Data flow between components
  - Per-vertex data (for vertex shader)
  - Per-fragment data (for pixel shader)
  - Uniform (constant) data: e.g. modelview matrix, material parameters

GPU Overview

- Rendering pipeline on current GPUs

- Low-level languages
  - Vertex programming
  - Fragment programming

- High-level shading languages

What Are Low-Level APIs?

- Similarity to assembler
  - Close to hardware functionality
  - Input: vertex/fragment attributes
  - Output: new vertex/fragment attributes
  - Sequence of instructions on registers
  - Very limited control flow (if any)
  - Platform-dependent
    BUT: there is convergence

What Are Low-Level APIs?

- Current low-level APIs:
  - OpenGL extensions: GL_ARB_vertex_program,
  - GL_ARB_fragment_program

- DirectX 9: Vertex Shader 2.0, Pixel Shader 2.0
  - Older low-level APIs:
  - DirectX 8.x: Vertex Shader 1.x, Pixel Shader 1.x
  - OpenGL extensions: GL_ATI_fragment_shader, GL_NV_vertex_program, …

Source : References given in the presentation

Why Use Low-Level APIs?

- Low-level APIs offer best performance & functionality
- Help to understand the graphics hardware (ATI's r300, NVIDIA's nv30, ...)
- Help to understand high-level APIs (Cg, HLSL, ...)
- Much easier than directly specifying configurable graphics pipeline (e.g. register combiners)

Applications Vertex Programming

- Customized computation of vertex attributes
- Computation of anything that can be interpolated linearly between vertices
- Limitations:
  - Vertices can neither be generated nor destroyed
  - No information about topology or ordering of vertices is available

OPEN_GL GL_ARB_vertex_program

- ❖ Circumvents the traditional vertex pipeline

- ❖ What is replaced by a vertex program?
  - ➢ Vertex transformations
  - ➢ Vertex weighting/blending
  - ➢ Normal transformations
  - ➢ Color material
  - ➢ Per-vertex lighting
  - ➢ Texture coordinate generation
  - ➢ Texture matrix transformations
  - ➢ Per-vertex point size computations
  - ➢ Per-vertex fog coordinate computations
  - ➢ Client-defined clip planes

OPEN_GL GL_ARB_vertex_program

❖ What is not replaced?

- Clipping to the view frustum
- Perspective divide (division by w)
- Viewport transformation
- Depth range transformation
- Front and back color selection
- Clamping colors
- Primitive assembly and per-fragment operations
- Evaluators

DirectX 9: Vertex Shader 2.0

- Vertex Shader 2.0 introduced in DirectX 9.0
- Similar functionality and limitations as GL_ARB_vertex_program
- Similar registers and syntax
- Additional functionality: static flow control
  - Control of flow determined by constants (not by per-vertex attributes)
  - Conditional blocks, repetition, subroutines

Source : References given in the presentation

Applications for Fragment Programming

- Customized computation of fragment attributes
- Computation of anything that should be computed per pixel
- Limitations:
  - Fragments cannot be generated
  - Position of fragments cannot be changed
  - No information about geometric primitive is available

OPEN_GL_ARB_fragment_program

- Circumvents the traditional fragment pipeline

- What is replaced by a pixel program?
  - Texturing
  - Color sum
  - Fog

  for the rasterization of points, lines, polygons, pixel rectangles, and bitmaps

- What is not replaced?
  - Fragment tests (alpha, stencil, and depth tests)
  - Blending

GPU Overview

❖Rendering pipeline on current GPUs

❖Low-level languages
- ➢ Vertex programming
- ➢ Fragment programming

❖High-level shading languages

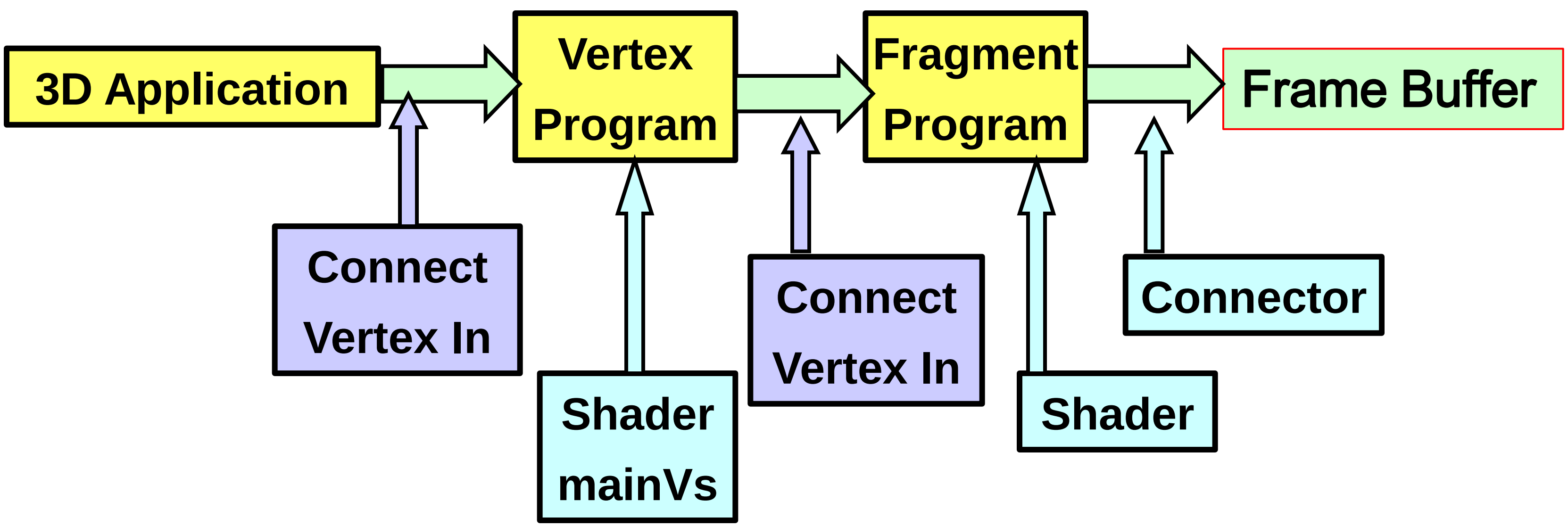
High-Level Shading Languages

- Why?
  - Avoids programming, debugging, and maintenance of long assembly shaders
  - Easy to read
  - Easier to modify existing shaders
  - Automatic code optimization
  - Wide range of platforms
  - Shaders often inspired RenderMan shading language

Source : References given in the presentation

Data Flow through Pipeline

- ❖ Vertex shader program
- ❖ Fragment shader program
- ❖ Connectors



High-Level Shading Languages

- Cg
  - "C for Graphics"
  - By NVIDIA

- HLSL
  - High-level shading language"
  - Part of DirectX 9 (Microsoft)

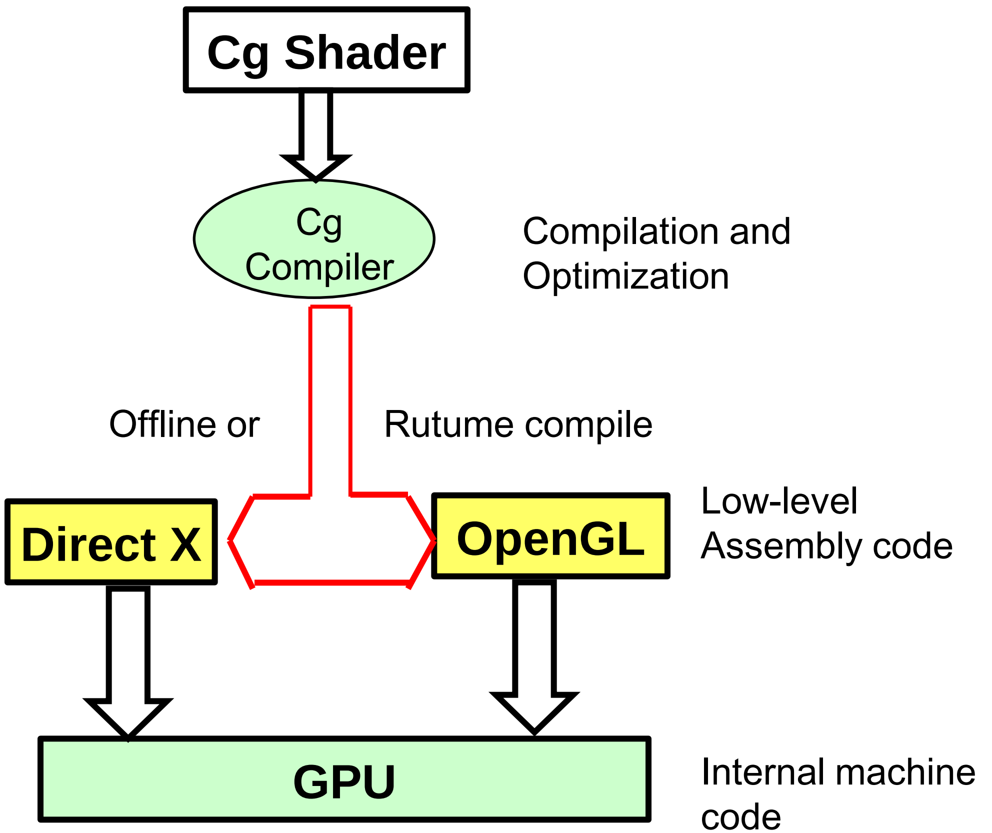
- OpenGL 2.0 Shading Language
  - Proposal by 3D Labs

GPU - Cg

- ❖ Typical concepts for a high-level shading language
- ❖ Language is (almost) identical to DirectX HLSL
- ❖ Syntax, operators, functions from C/C++
- ❖ Conditionals and flow control
- ❖ Backends according to hardware profiles

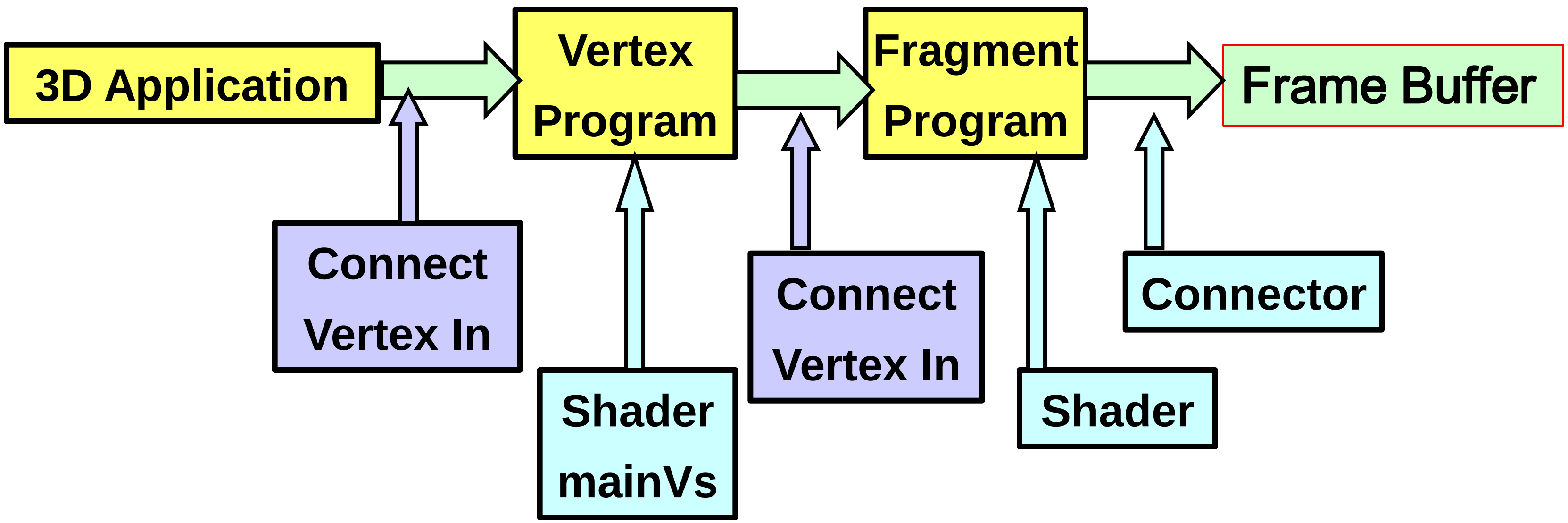
- ❖ Support for GPU-specific features (compare to low-level)
  - ➢ Vector and matrix operations
  - ➢ Hardware data types for maximum performance
  - ➢ Access to GPU functions: mul, sqrt, dot, …
  - ➢ Mathematical functions for graphics, e.g. reflect
  - ➢ Profiles for particular hardware feature sets

Workflow in Cg

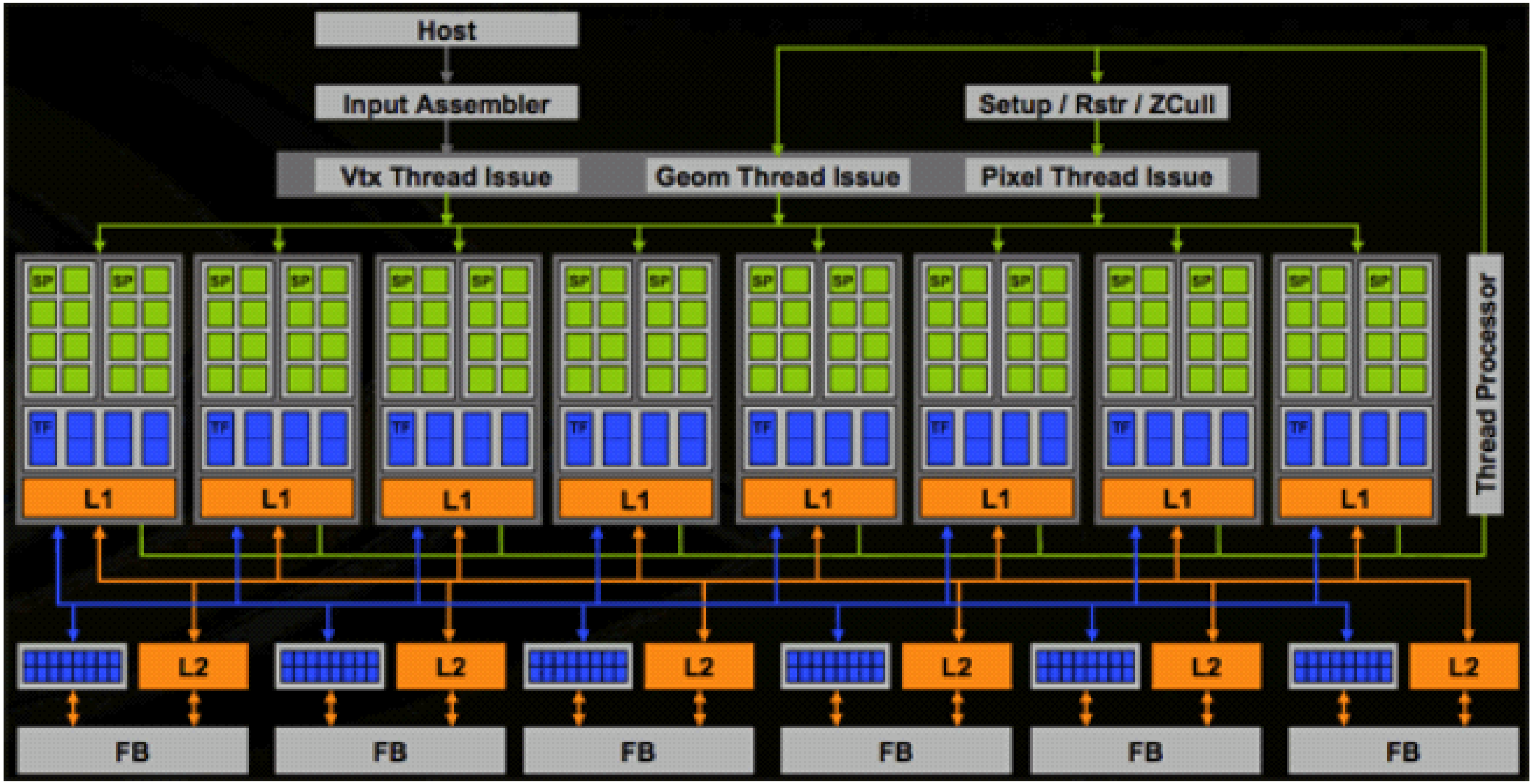


Phong Shading in Cg: Vertex Shader

- First part of pipeline
- Connectors: what kind of data is transferred to/from vertex program?
- Actual vertex shader



NVIDIA G80 Block Diagram



- ❖ Very little of this is graphic specic
- ❖ ...but, assumes threads are independent

Hyper "Core" Computers

Speculation about the computer of the next decade:

- 10s of CPU cores
  - Use for scheduling
  - Use for \irregular" part of problem
  - Maybe higher precision (correction steps)
- 100s of GPU cores
  - Use for \regular" part of problem
- NUMA (Non-Uniform Memory Access) for both
  - Programming languages must expose this
  - Runtime systems?
  - Always out-of-(some)-core
- Clusters of these?
  - OpenMP/MPI not sufficient

Limitations of GPUs

If the GPU is so great, why are we still using the CPU?
You can not simply “port" existing code and algorithms!

- ❖ Data-stream mindset required
  - ➢ Parallel algorithms
  - ➢ New data structures (dynamic data structures are troublesome)
- ❖ Not suitable to all problems
  - ➢ Pointer chasing impossible or inecient
  - ➢ Recursion
- ❖ Debugging is hard
  - ➢ Hardware is designed without debug bus
  - ➢ Driver is closed
- ❖ Huge performance clis
- ❖ No standard API
  - ➢ More about this later...

GPU Programming

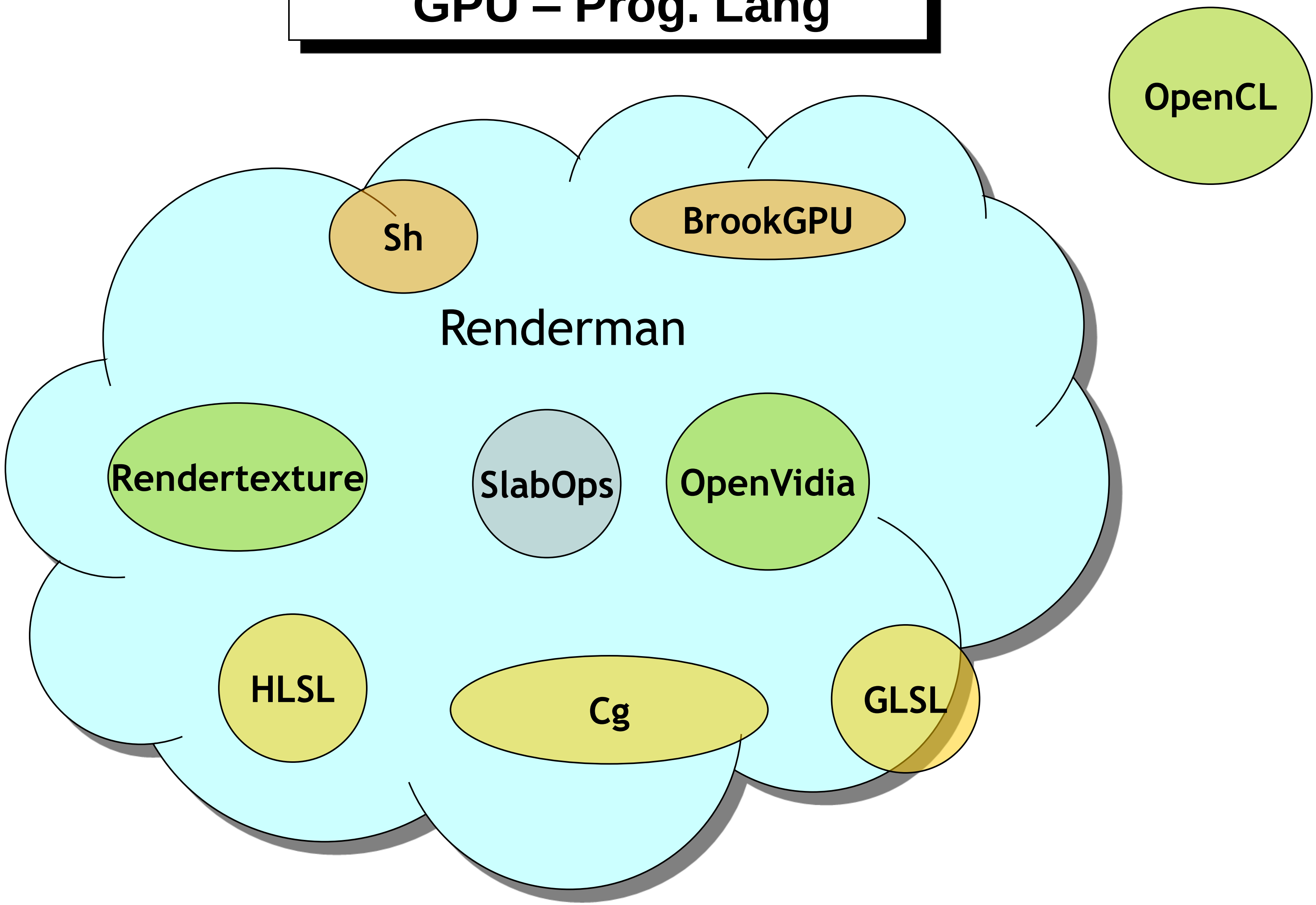
- GPUs have traditionally been closed architectures.
  - Must program them through closed-source graphics driver
  - Driver is like an OS (threads, scheduling, protected memory)
- OpenGL/DirectX are standard, but
  - Designed for graphics, not general purpose computations
  - Many revisions of each standard
    - New revisions for each HW-generation
  - Allows for \capabilities"
  - Large variations between vendors
- Both vendors now have dedicated GPGPU APIs
  - Nvidia CUDA (Compute Unified Device Architecture)
  - AMD CTM (Close To Metal) – AMD ATI - FireStream
- GPGPU version" of hardware as well

Part-I (B)

An Overview of GPU Prog. Languages

Source & Acknowledgements : NVIDIA, AMD, References

GPU – Prog. Lang



GPU - Some History

- Cook and Perlin first to develop languages for performing **shading calculations**
- Perlin computed noise functions procedurally; introduced control constructs
- Cook developed idea of *shade trees* @Lucasfilm
- These ideas led to development of Renderman at Pixar (Hanrahan *et al*) in 1988.
- Renderman is STILL **shader language** of choice for high quality rendering !
- Languages intended for offline rendering; no interactivity, but high quality.

GPU - Some History

- After RenderMan, independent efforts to develop high level shading languages at SGI (ISL), Stanford (RTSL).
- ISL targeted fixed-function pipeline and SGI cards (remember compiler from previous lecture): goal was to map a RenderMan-like language to OpenGL
- RTSL took similar approach with programmable pipeline and PC cards (recall compiler from previous lecture)
- RTSL morphed into **Cg**.

GPU - Some History

- **Cg** was pushed by N**VIDIA** as a platform-neutral, card-neutral programming environment.
- In practice**, Cg** tends to work better on NVIDIA cards (better demos, special features etc).
- ATI made brief attempt at competition with Ashli/RenderMonkey.
- HLSL was pushed by Microsoft as a DirectX-specific alternative.
- In general, HLSL has better integration with the DirectX framework, unlike **Cg** with OpenGL/DirectX.

GPU – Level 1: Better Than Assembly ?

Overview –

C-like vertex, Cg, HLSL, GLSL, Data Types, Shaders,Compilation

GPU Lang. - Prog.: C-like vertex and fragment code

- Languages are specified in a C-like syntax.
- The user writes explicit vertex and fragment programs.
- Code compiled down into pseudo-assembly
  - this is a source-to-source compilation: no machine code is generated.
- Knowledge of the pipeline is essential
  - Passing array = binding texture
  - Start program = render a quad
  - Need to set transformation parameters
  - Buffer management a pain

GPU Lang. - Prog.: Cg

- Platform neutral, architecture "neutral" shading language developed by NVIDIA.

- One of the first GPGPU languages used widely.

- Because Cg is platform-neutral, many of the other GPGPU issues are not addressed
  - managing pbuffers
  - rendering to textures
  - handling vertex buffers

"As we started out with Cg it was a great boost to getting programmers used to working with programmable GPUs. Now Microsoft has made a major commitment and in the long term we don't really want to be in the programming language busies"

David Kirk, NVIDIA

GPU Lang. - Prog.: HLSL

- Developed by Microsoft; tight coupling with DirectX
- Because of this tight coupling, many things are easier (no RenderTexture needed !)
- Xbox programming with DirectX/HLSL (XNA)
- But…
  - Cell processor will use OpenGL/Cg

GPU Lang. - Prog.: GLSL

- GLSL is the latest shader language, developed by 3DLabs in conjunction with the OpenGL ARB, specific to OpenGL.
- Requires OpenGL 2.0
- NVIDIA doesn't yet have drivers for OpenGL 2.0 !! Demos (appear to be) emulated in software
- ATI appears to have native GL 2.0 support and thus support for GLSL.

Multiplicity of languages likely to continue

GPU Lang. - Prog.: Datatypes

- Scalars: float/integer/boolean
- Scalars can have 32 or 16 bit precision (ATI supports 24 bit, GLSL has 16 bit integers)
- vector: 3 or 4 scalar components.
- Arrays (but only fixed size)
- Limited floating point support; no underflow/overflow for integer arithmetic
- No bit operations
- Matrix data types
- Texture data type
  - power-of-two issues appear to be resolved in GLSL
  - different types for 1D, 2D, 3D, cubemaps.

GPU Lang. - Prog.: DatatBinding

Data Binding modes:

- uniform: the parameter is fixed over a glBegin()-glEnd() call.
- varying: interpolated data sent to the fragment program (like pixel color, texture coordinates, etc)
- attribute: per-vertex data sent to the GPU from the CPU (vertex coordinates, texture coordinates, normals, etc).
- Data direction:
- in: data sent into the program (vertex coordinates)
- out: data sent out of the program (depth)
- inout: both of the above (color)

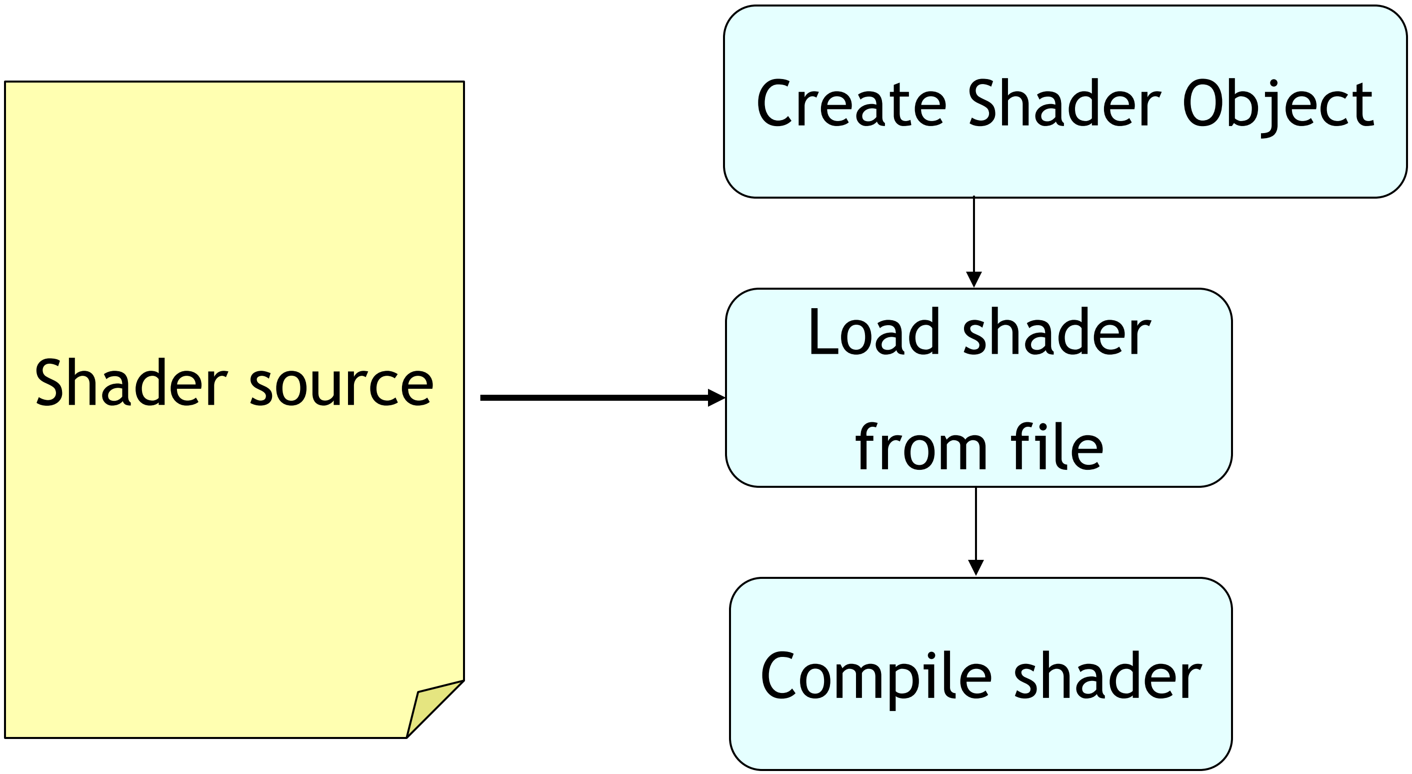
GPU Lang. - Prog.: Operations And Control Flow

- Usual arithmetic and special purpose algebraic ops (trigonometry, interpolation, discrete derivatives, etc)
- No integer mod…
- for-loops, while-do loops, if-then-else statements.
- **`discard`** allows you to kill a fragment and end processing.
- Recursive function calls are unsupported, but simple function calls are allowed
- Always one "main" function that starts the program, like C.

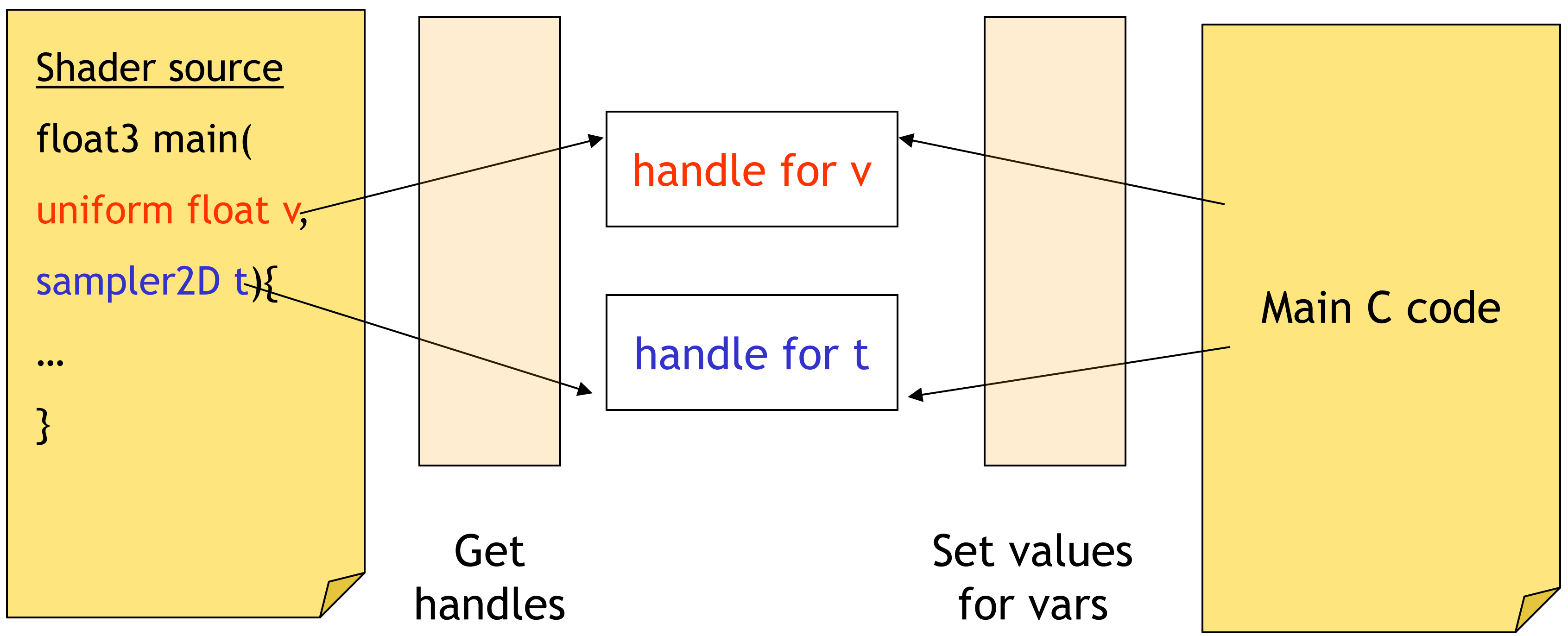
GPU Lang.-Prog.: working with Shaders : The Mechanics

- This is the most painful part of working with shaders.
- All three languages provide a "runtime" to load shaders, link data with shader variables, enable and disable programs.
- Cg and HLSL compile shader code down to assembly ("source-to-source").
- GLSL relies on the graphics vendor to provide a compiler directly to GPU machine code, so no intermediate step takes place.

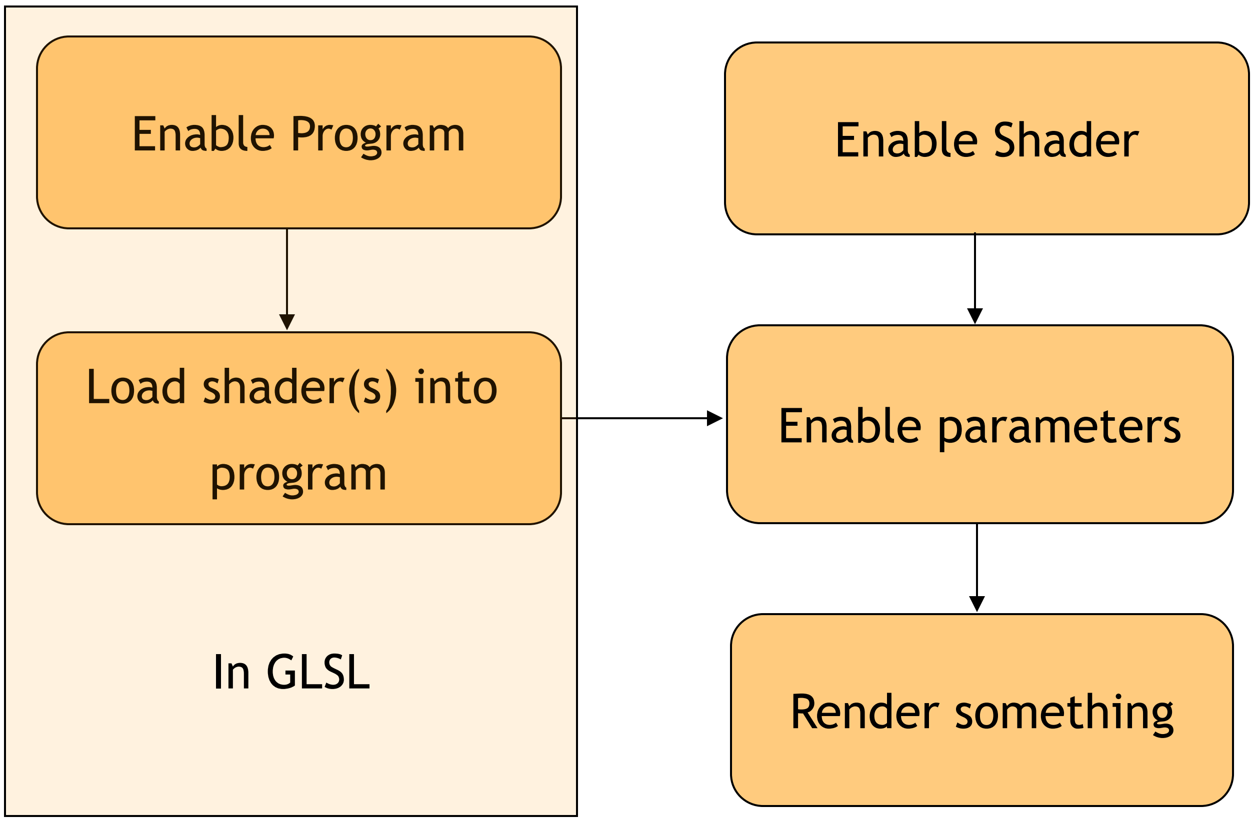
Step 1: Load the shader



Step 2: Bind Variables



Step 3: Run the Shaders



GPU Lang.-Prog.: Direct Compilation

- ❖ **Cg** code can be compiled to fragment code for different platforms (directx, nvidia, arbfp)
- ❖ HLSL compiles directly to directx
- ❖ GLSL compiles natively.
- ❖ It is often the case that inspecting the **Cg** compiler output reveals bugs, shows inefficiencies etc that can be fixed by writing assembly code (like writing asm routines in C)
- ❖ In GLSL you can’t do this because the code is compiled natively: you have to trust the vendor compiler !

GPU Lang.-Prog.: Overview

- ❖ Shading languages like Cg, HLSL, GLSL are ways of approaching Renderman but using the GPU.

- ❖ These will never be the most convenient approach for general purpose GPU programming

- ❖ But they will probably yield the most efficient code
  - you either need an HLL and great compilers
  - or you suffer and program in these.

GPU – Lang. Prog. ; Wrapper libraries

- Writing code that works cross-platform, with all extensions, is hard.
- Wrappers take care of the low-level issues, use the right commands for the right platform, etc.
- **Render Texture**:
  - Handles offscreen buffers and render-to-texture cleanly
  - works in both windows and linux (only for OpenGL though)
  - de facto class of choice for all Cg programming (use Cg for the code, and **RenderTexture** for texture management).

GPU – Lang. Prog. ; OpenVidia

- ❖ Video and image processing library developed at University of Toronto.
- ❖ Contains a collection of fragment programs for basic vision tasks (edge detection, corner tracking, object tracking, video compositing, etc)
- ❖ Provides a high level API for invoking these functions.
- ❖ Works with Cg and OpenGL, only on linux (for now)
- ❖ Level of transparency is low: you still need to set up GLUT, and allocate buffers, but the details are somewhat masked)

GPU – Lang. Prog. : OpenVidia Example

- Create processing object:
  - `d=new FragPipeDisplay(<parameters>);`
- Create image filter
  - `filter1 = new GenericFilter(…,<cg-program>);`
- Make some buffers for temporary results:
  - `d->init_texture(0, 320, 240, foo);`
  - `d->init_texture4f(1, 320, 240, foo);`
- Apply filter to buffer, store in output buffer
  - `d->applyFilter(filter1, 0,1);`

GPU – Lang. Prog. : High Level C-like languages

- Main goal is to hide details of the runtime and distill the essence of the computation.
- These languages exploit the ***stream*** aspect of GPUs explicitly
- They differ from libraries by being general purpose.
- They can target different backends (including the CPU)
- Either embed as C++ code (Sh) or come with an associated compiler (Brook) to compile a C-like language.

GPU Lang. Prog. : High Level C-like languages :Sh

- Open-source code developed by group led by Michael McCool at Waterloo
- Technical term is ‘metaprogramming’
- Code is embedded inside C++; **no** extra compile tools are necessary.
- **Sh** uses a *staged compiler*: parts of code are compiled when C++ code is compiled, and the rest (with certain optimizations) is compiled at runtime.
- Has a very similar flavor to functional programming
- Parameter passing into streams is seamless, and resource constraints are managed by ***virtualization.***

GPU Lang. Prog. : High Level C-like languages :Sh And more ….. DirectX

- All kinds of other functions to extract data from streams and textures.
- Lots of useful 'primitive' streams like passthru programs and generic vertex/fragment programs, as well as specialized lighting shaders.
- **Sh** is closely bound to OpenGL; you can specify all usual OpenGL calls, and **Sh** is invoked as usual via a display() routine.
- Plan is to have **DirectX** binding ready shortly (this may be already be in)
- Because of the multiple backends, you can debug a shader on the CPU backend first, and then test it on the GPU.

GPU Lang. Prog. : High Level C-like languages
Brook GPU

- Open-source code developed by Ian Buck and others at Stanford.
- Intended as a pure stream programming language with multiple backends.
- Is not embedded in C code; uses its own compiler (brcc) that generates C code from a .br file.
- Workflow:
  - Write Brook program (.br)
  - Compile Brook program to C (brcc)
  - Compile C code (gcc/VC)

GPU Lang. Prog. : High Level C-like languages
Brook GPU

- Designed for general-purpose computing (this is primary difference in focus from **Sh**)
- You will almost never use any graphics commands in Brook.
- Basic data type is the stream.
- Types of functions:

GPU Lang. Prog. : High Level C-like languages
Brook GPU

- Types of functions:
  - Kernel: takes one or more input streams and produces an output stream.
  - Reduce: takes input streams and reduces them to scalars (or smaller output streams)
  - Scatter: $a[o_i] = s_i$. Send stream data to array, putting values in different locations.
  - Gather: Inverse of scatter operation. $s_i = a[o_i]$.
- Support of all operations are required … check.

GPU Lang. Prog. : High Level C-like languages Sh Vs Brook GPU

- ☺ Brook is more general: you don’t need to know graphics to run it.
- ☺ Very good for prototyping
- ☹ You need to rely on compiler being good.
- ☹ Many special GPU features cannot be expressed cleanly.

- ☺ Sh allows better control over mapping to hardware.
- ☺ Embeds in C++; no extra compilation phase necessary.
- ☹ Lots of behind-the-scenes work to get virtualization: is there a performance hit ?
- ☹ Still requires some understanding of graphics.

NVIDIA CUDA (Compute Unified Device Architecture)

C-like API for programming newer Nvidia GPUs

- ❖ Computation kernels are written in C
  - ➢ Compiles with dedicated compiler, nvcc
- ❖ Kernels are executed as threads, threads organized into blocks
  - ➢ Programmer decides #threads, #threads/block, and mem/block
- ❖ Exposes different kinds of memory
  - ➢ Thread-local (register)
  - ➢ Shared per block
  - ➢ Global (not cached, write everywhere)
  - ➢ Texture (cached read only memory)
  - ➢ Constant(cached read only memory)
- ❖ Some synchronization primitives
- ❖ cudaMalloc, cudaMemcpy for allocating and copying memory

GPU Lang. Prog. : High Level C-like languages
The Big Picture

- The advent of Cg, and then Brook/Sh signified a huge increase in the number of GPU apps. Having good programming tools is worth a lot !

- The tools are still somewhat immature; almost non-existent debuggers and optimizers, and only one GPU simulator (Sm).

- I shouldn’t have to worry about the correct parameters to pass when setting up a texture for use as a buffer: we need better wrappers.

GPU Lang. Prog. : High Level C-like languages
The Big Picture

- Compiler efforts are lagging application development: more work is needed to allow for high level language development without compromising performance.

- In order to do this, we need to study stream programming. Maybe draw ideas from the functional programming world ?

- Libraries are probably the way forward for now.

Hyper "Core" Computers

Speculation about the computer of the next decade:

- 10s of CPU cores
  - Use for scheduling
  - Use for \irregular" part of problem
  - Maybe higher precision (correction steps)
- 100s of GPU cores
  - Use for \regular" part of problem
- NUMA (Non-Uniform Memory Access) for both
  - Programming languages must expose this
  - Runtime systems?
  - Always out-of-(some)-core
- Clusters of these?
  - OpenMP/MPI not sufficient

Limitations of GPUs

If the GPU is so great, why are we still using the CPU?

You can not simply “port" existing code and algorithms!

- ❖ Data-stream mindset required
  - ➢ Parallel algorithms
  - ➢ New data structures (dynamic data structures are troublesome)
- ❖ Not suitable to all problems
  - ➢ Pointer chasing impossible or inecient
  - ➢ Recursion
- ❖ Debugging is hard
  - ➢ Hardware is designed without debug bus
  - ➢ Driver is closed
- ❖ Huge performance clis
- ❖ No standard API
  - ➢ More about this later...

GPU Programming

- ❖ GPUs have traditionally been closed architectures.
  - ➢ Must program them through closed-source graphics driver
  - ➢ Driver is like an OS (threads, scheduling, protected memory)
- ❖ OpenGL/DirectX are standard, but
  - ➢ Designed for graphics, not general purpose computations
  - ➢ Many revisions of each standard
    - ➢ New revisions for each HW-generation
  - ➢ Allows for \capabilities"
  - ➢ Large variations between vendors
- ❖ Both vendors now have dedicated GPGPU APIs
  - ➢ Nvidia CUDA (Compute Unified Device Architecture)
  - ➢ AMD CTM (Close To Metal) – AMD ATI - FireStream
- ❖ GPGPU version" of hardware as well

Conclusions

- ❖ GPU Programming Language
- ❖ GPU Programming – OpenGL, DirectX, NVIDIA (CUDA), AMD (Brook+)
- ❖ OPECG-2009 -Hands-on session : Examples

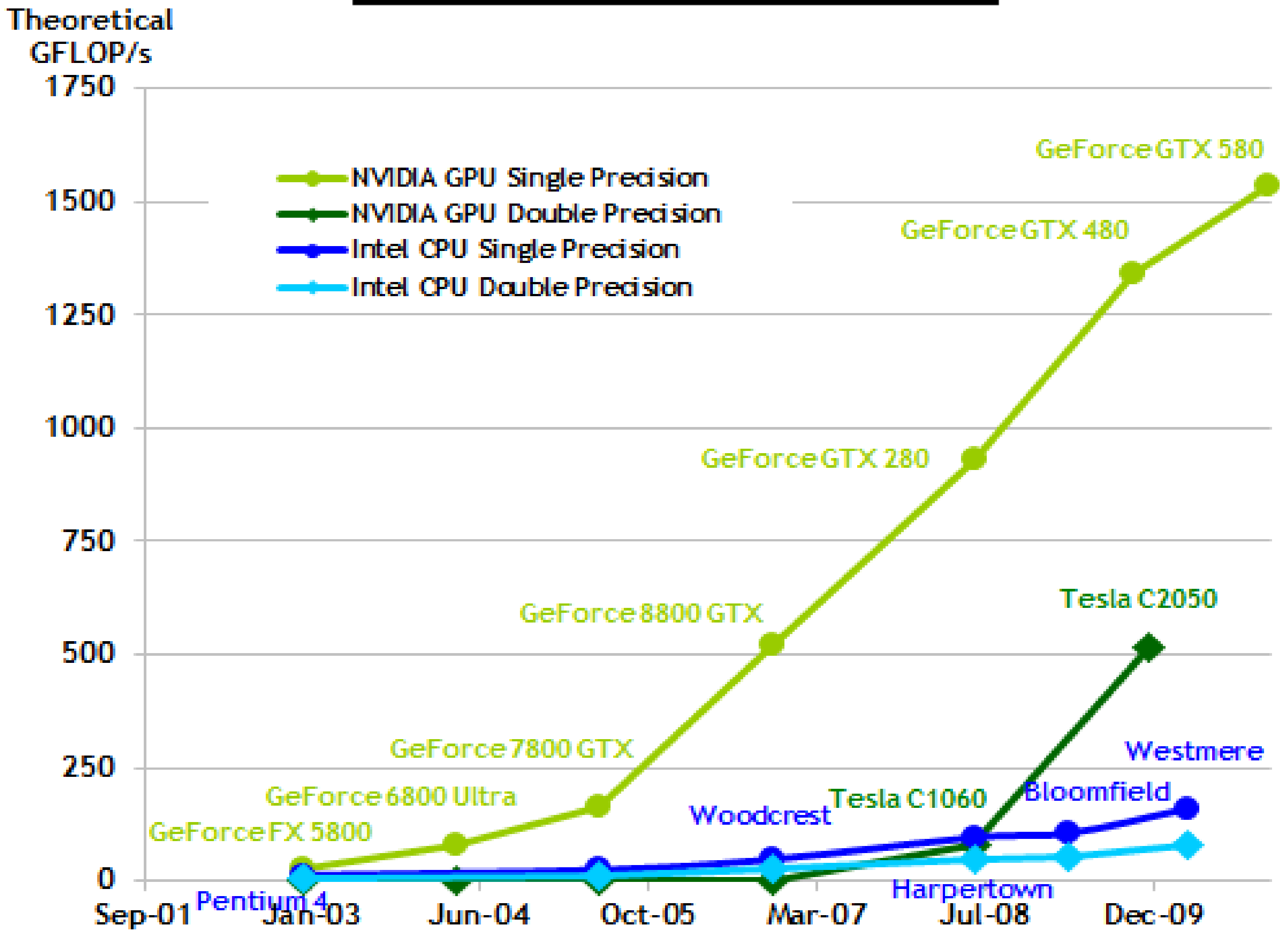
This page is intentionally kept blank

Part-II(A)

An Overview of CUDA enabled NVIDIA GPUs

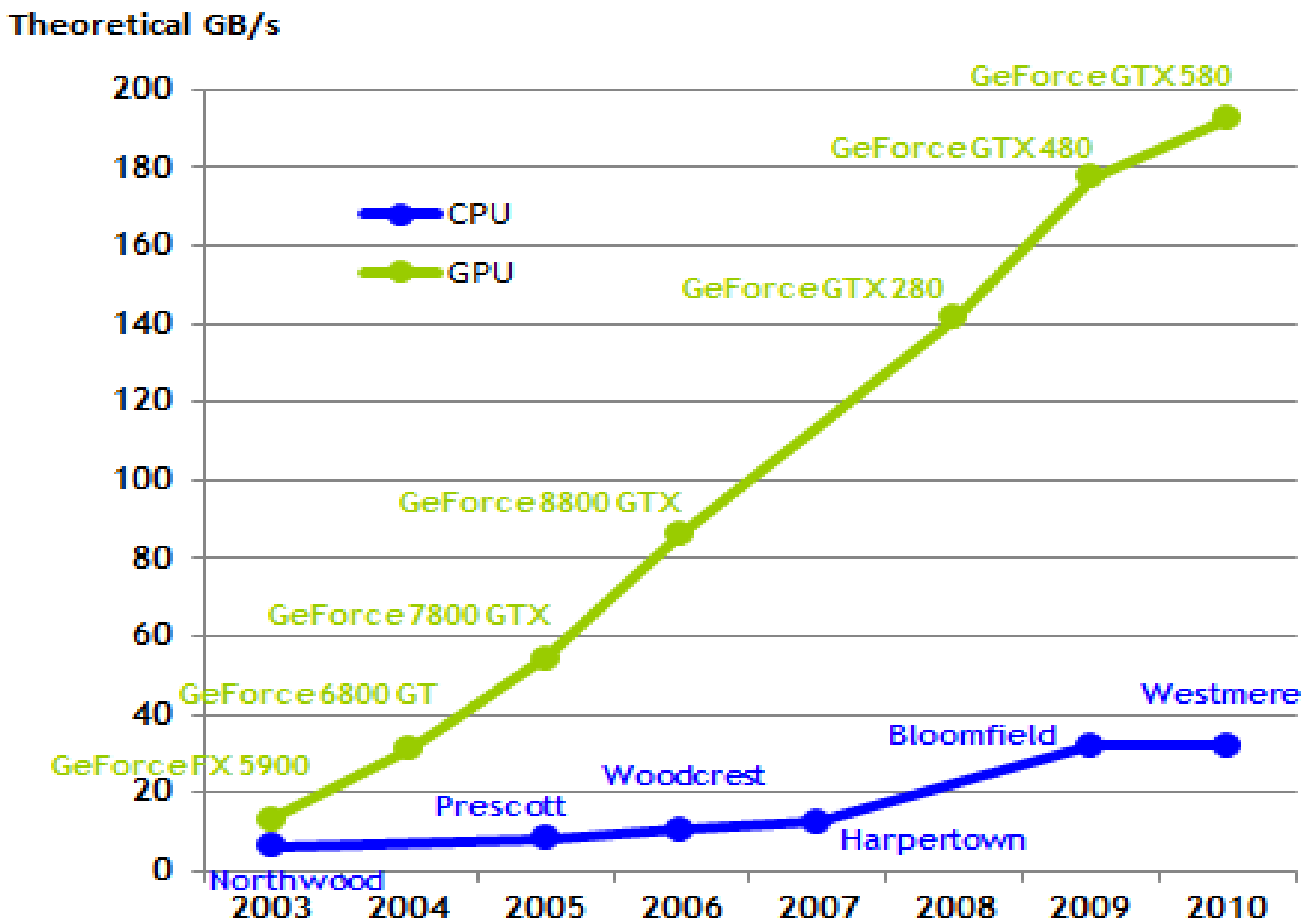
Source & Acknowledgements : NVIDIA, References

Computing - CPU/GPU



Source & Acknowledgements : NVIDIA, References

Computing - CPU/GPU

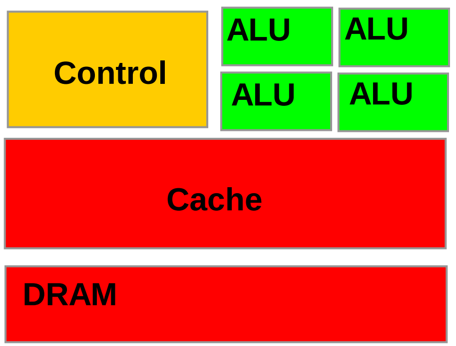


Floating-Point Operations per Second and Memory Bandwidth for the CPU and GPU

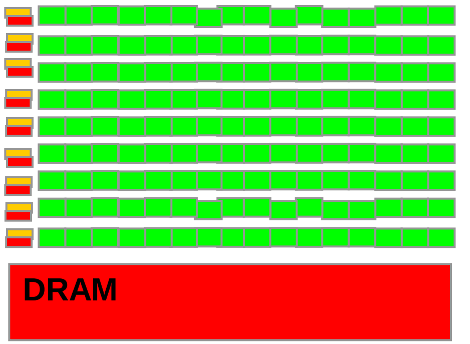
Source & Acknowledgements : NVIDIA, References

Why Are GPUs So Fast?

- GPU originally specialized for math-intensive, highly parallel computation
- So, more transistors can be devoted to data processing rather than data caching and flow control



CPU



GPU

AMD

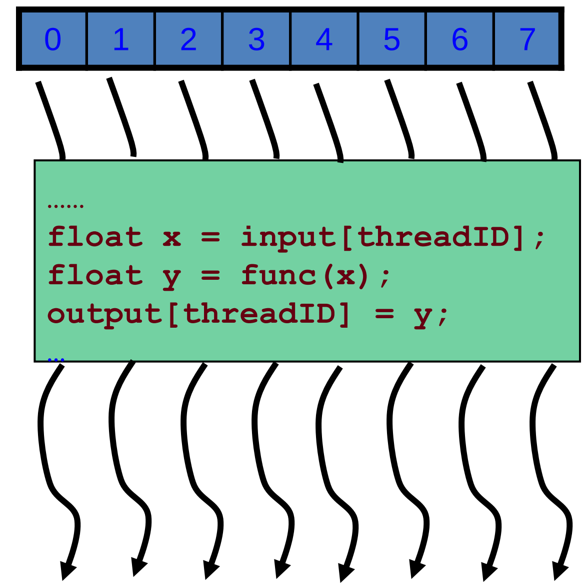
NVIDIA

- Commodity industry: provides economies of scale
- Competitive industry: fuels innovation

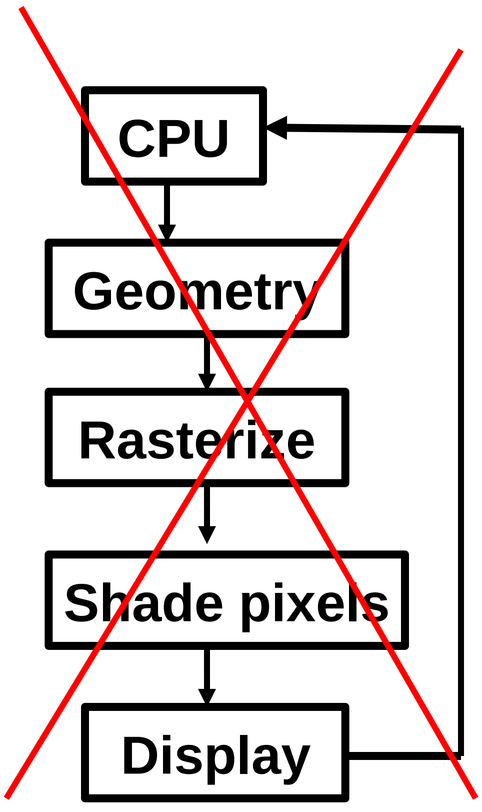
Source : NVIDIA, References

Some Design Goals

- ❖ Scale to 100's of cores, 1000's of parallel
- ❖ threads
- ❖ Let programmers focus on parallel algorithms & Re-writing the Code
  - Not on the mechanics of a parallel programming language
- ❖ Enable heterogeneous systems (i.e. CPU + GPU)
  - CPU and GPU are separate devices with separate DRAMs



Computer Graphics



- ❖ Hardware mimicked graphics APIs
- ❖ It is possible to formulate many problems in this framework
  - ➢ Uses graphics APIs
  - ➢ Classical GPGPU"

DO NOT DO THIS ANYMORE!

(Unless for graphics)

Use GPU Computing with CUDA APIs for Data Parallel Computations .(CUDA = Compute Unified Device Architecture. CUDA is co-designed hardware & software for direct GPU computing)

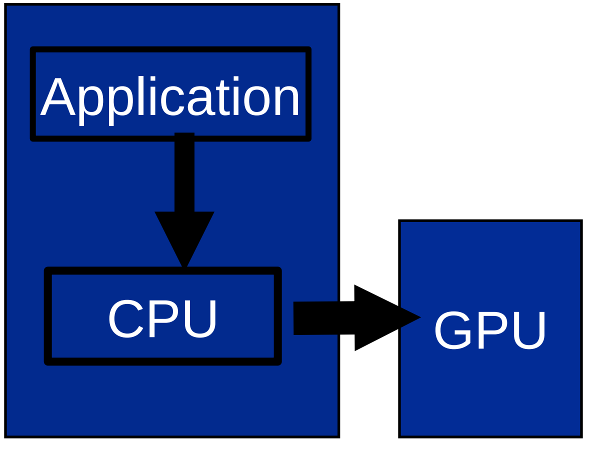
"OpenCL will enable programmers to easily develop portable applications that maximize the performance on GPU architectures.

GPU Computing : Think in Parallel

- Performance = parallel hardware

  \+

  scalable parallel program
- GPU Computing drives new applications
  - Reducing “Time to Discovery”
  - 100 x Speedup changes science & research methods
- New applications drive the future of GPUs
  - Drives new GPU capabilities
  - Drives hunger for more performance



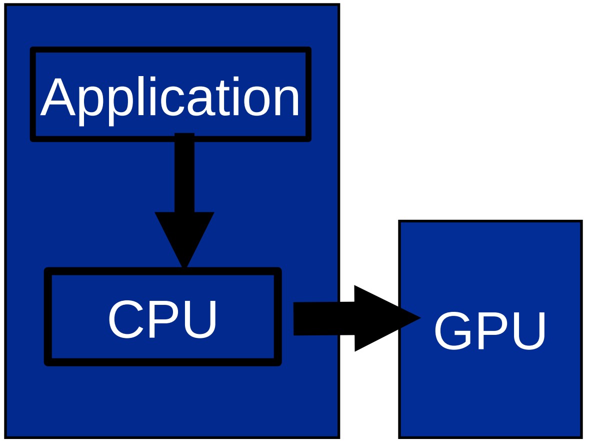
Source & Acknowledgements : NVIDIA, References

GPU Computing : Think in Parallel

❖ Speedups of 8 x to 30x are quite common for certain class of applications

❖ The GPU is a data-parallel processor

- Thousands of parallel threads
- Thousands of data elements to process
- All data processed by the same program
  - ➢ SPMD computation model
- Contrast with task parallelism and ILP



❖ Best results when you "**Think Data Parallel**"

- Design your algorithm for data-parallelism
- Understand parallel algorithmic complexity and efficiency
- Use data-parallel algorithmic primitives as building blocks

Source : NVIDIA, AMD, References

Source & Acknowledgements : NVIDIA, References

Why Are GPUs So Fast?

- ❖ Optimized for structured parallel execution
  - Extensive ALU counts & Memory Bandwidth
  - Cooperative multi-threading hides latency
- ❖ Shared Instructions Resources
- ❖ Fixed function units for parallel workloads dispatch
- ❖ Extensive exploitations of Locality

- Performance /(Cost/Watt); Power for Core
- Structured Parallelism enables more flops less watts

Source : NVIDIA, AMD, References

GPU Computing : Optimise Algorithms for the GPU

- ❖ Maximize independent parallelism
- ❖ Maximize arithmetic intensity (math/bandwidth)
- ❖ Sometimes it's better to recompute than to cache
  - GPU spends its translators on ALUs, not memory
- ❖ Do more computation on the GPU to avoid costly data transfers

  Even low parallelism computations can sometimes be faster than transferring back and forth to host

Source & Acknowledgements : NVIDIA, References

GPU Computing : Use Parallelism Efficiently

- Partition your computation to keep the GPU multiprocessors equally busy
  - Many threads, many thread blocks
- Keep resource usage low enough to support multiple active thread blocks per multiprocessor
  - Registers, shared memory

Source : NVIDIA,AMD, References

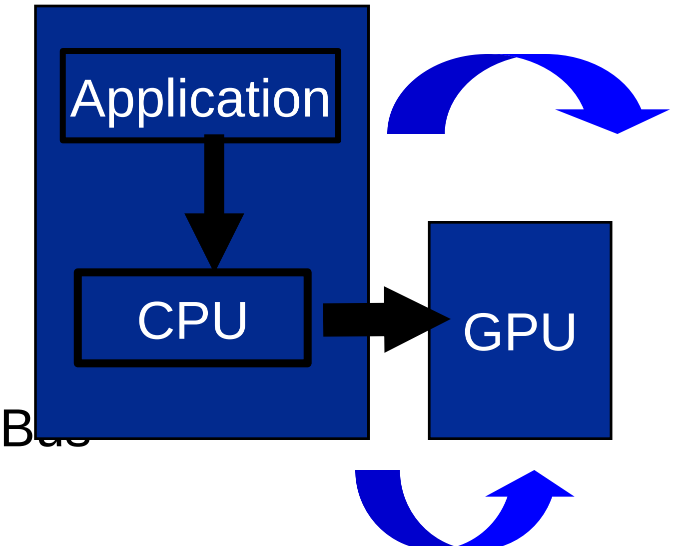
GPU Programming : Two Main Challenges

GPU Challenges with regard to Scientific Computing

Challenge 1 : Programmability

❖ Example : Matrix Computations

- To port an existing scientific application to a GPU



❖ GPU memory exists on the card itself

- Must send matrix array over PCI-Express Bus
  - – Send **A**, **B, C** to **GPU** over PCIe
  - –Perform GPU-based computations on **A,B, C**
  - – Read result **C** from **GPU** over PCIe

❖ The user must focus considerable effort on optimizing performance by manually orchestrating data movement and managing thread level parallelism on GPU.

GPU Programming : Two Main Challenges

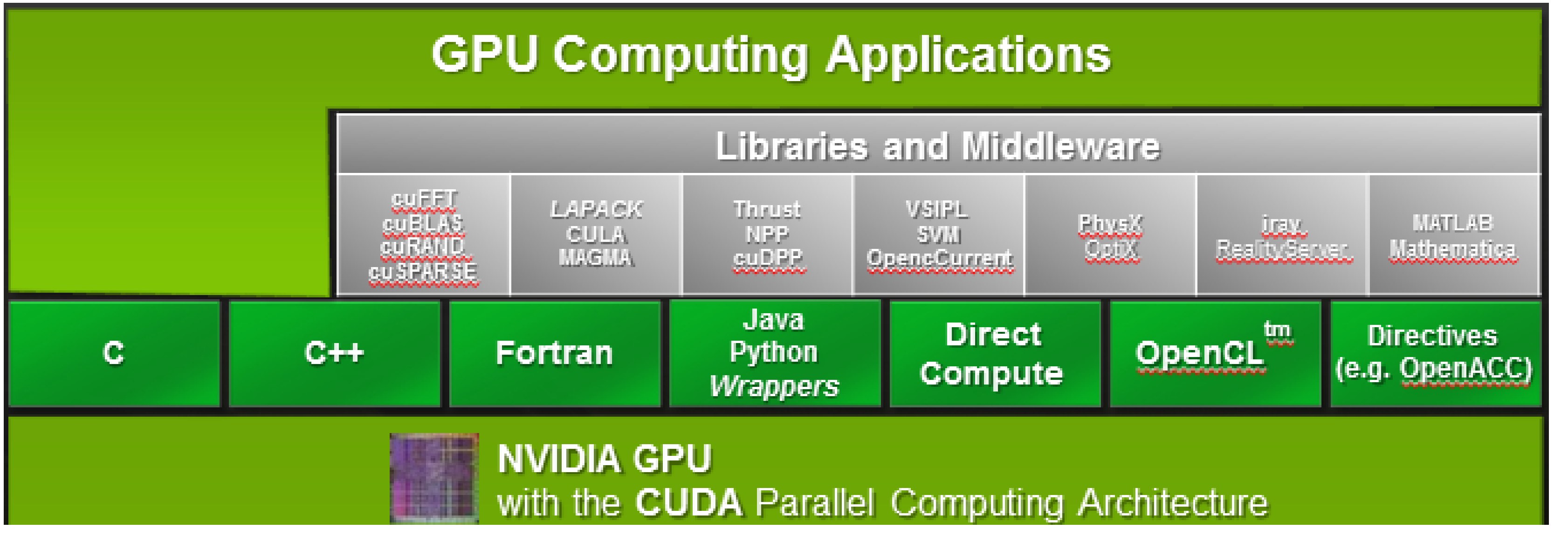
Challenge 2 : Accuracy

- ❖ Example : Non-Scientific Computation - Video Games (Frames) (A single bit difference in a rendered pixel in a real-time graphics program may be discarded when generating subsequence frames)

- ❖ **Past History :** Most GPUs support single/double precision, 32 bit /64-bit floating point operation, - all GPUs have necessarily implemented the full IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754)

Source & Acknowledgements : NVIDIA, References

NVIDIA GPU Computing - CUDA Kernels and Threads



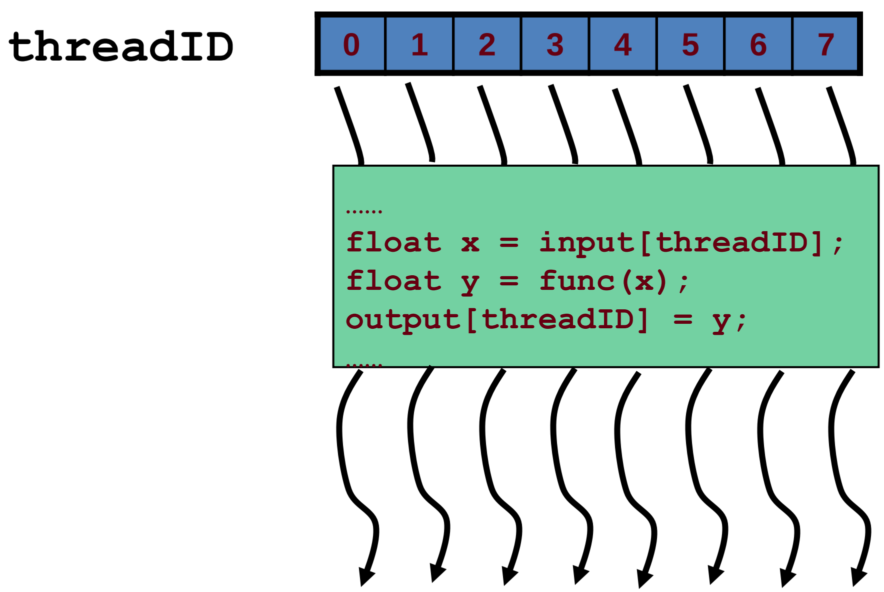
CUDA is Designed to Support Various Languages and Application Programming Interfaces

Source & Acknowledgements : NVIDIA, References

Arrays of Parallel Threads

❖ A CUDA kernel is executed by an array of threads

- All threads run the same code
- Each thread has an ID that it uses to compute memory addresses and make control decisions



Source: NVIDIA

Solution: GPU Computing – NVIDIA CUDA

- **NEW:** *GPU Computing* with CUDA
  - CUDA = ***Compute Unified Driver Architecture***
  - Co-designed hardware & software for direct GPU computing
- Hardware: fully general data-parallel architecture
  - General thread launch
  - Global load-store
  - Parallel data cache
- Software: program the GPU in C
  - Scalable data-parallel execution/ memory model

- Scalar architecture
- Integers, bit operations
- Single / Double precision C with powerful extensions
- CUDA 4.0 /CUDA 5.0

Source & Acknowledgements : NVIDIA, References

NVIDA :CUDA - Quick terminology review

❖ *CUDA* is a development platform designed for writing and running general-purpose applications on the ***nVIDIA*** *GPU*

- Similar to Graphics applications, CUDA applications can be accelerated by data-parallel computation of millions of ***threads.***

❖ ***A thread*** here is an instance of a ***kernel***, namely a program running on the ***GPU.***

❖ *GPU* platform can be regarded as a single instruction, multiple data (***SIMD***) parallel machine rather than graphics hardware

- Keeping **SIMD** in mind, there is no need to understand the graphics pipeline to execute programs on this highly threaded architecture.

Source & Acknowledgements : NVIDIA, References

CUDA - Quick terminology review

- ***Thread:*** concurrent code and associated state executed on the CUDA device (in parallel with other threads)
  - The unit of parallelism in CUDA
  - Note difference from CPU threads: creation cost, resource usage, and switching cost of GPU threads is much smaller

- *Warp:* a group of threads executed *physically* in parallel (SIMD)

- ***Thread Block:*** a group of threads that are execute together and can share memory on a single multiprocessor

- ***Grid:*** a group of thread blocks that execute a single CUDA program *logically* in parallel

- ***Device:*** GPU ***Host:*** CPU

- ***SM:*** Multiprocessor

Source : NVIDIA, References

NVIDA :CUDA – Data Parallelism

❖ ***To a CUDA Developer,***

- The computing system consists of a host, which is a traditional central processing unit (CPU) such as Intel, AMD, IBM, Cray multi-core architecture and one more devices, which are massively parallel processors equipped with a large number of arithmetic execution units.

❖ Computing depends upon the concept of ***Data Parallelism***

Image Processing, Video Frames, Physics, Aero dynamics, Chemistry, Bio-Informatics

- Regular Computations and Irregular Computations.

Source & Acknowledgements : NVIDIA, References

NVIDIA GPU Computing - CUDA Kernels and Threads

❖ **NEW:** *GPU Computing* with CUDA

- CUDA = Compute Unified Device Architecture
- Co-designed hardware & software for direct GPU computing

❖ *Hardware: fully general data-parallel architecture*

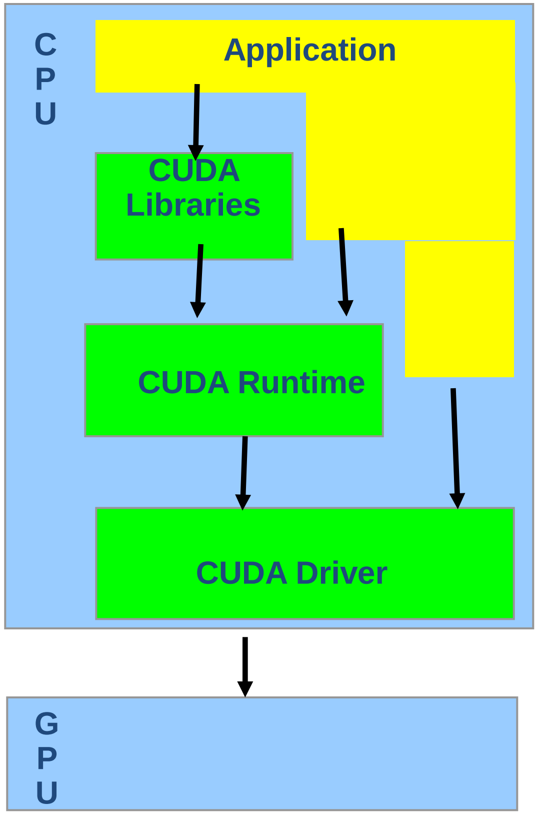
- General thread launch; Global load-store
- Parallel data cache

❖ *Software: program the GPU in C /C++*

- Scalable data-parallel execution/ memory model; Single/Double precision

❖ Hundreds of times faster than global memory

❖ Use one/ a few threads to load/computer data shared by all thread



Compute Unified Device Architecture Software Stack

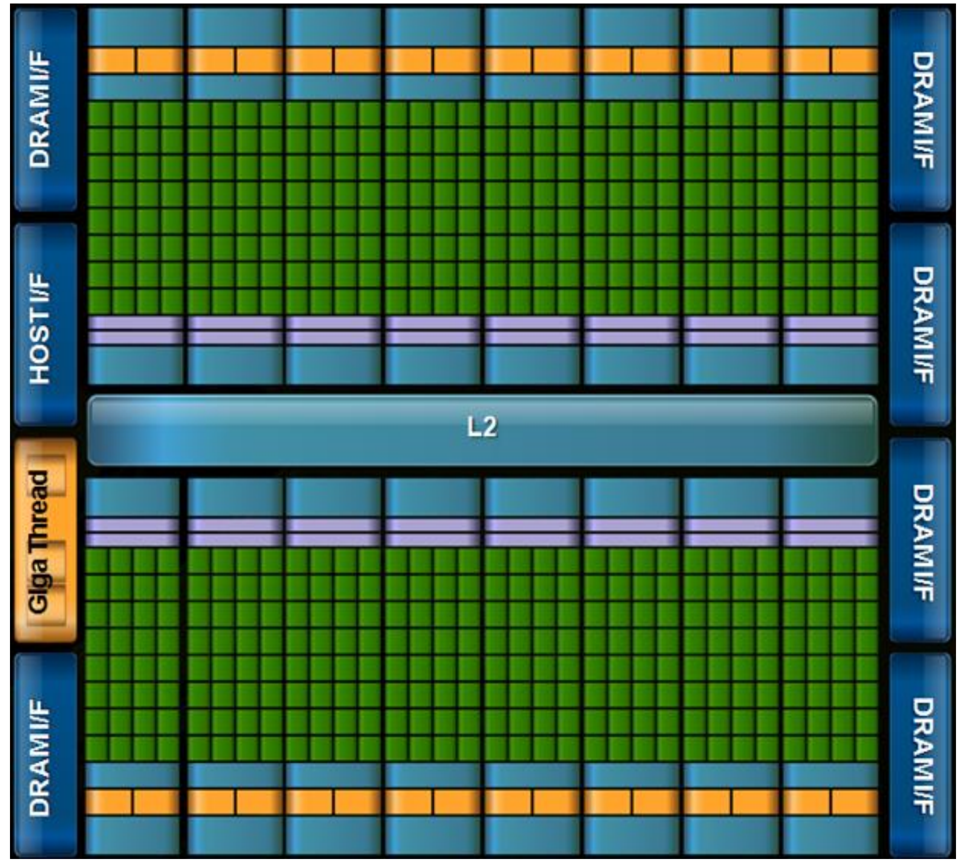
Source & Acknowledgements : NVIDIA, References

GPU : Architecture

Several multiprocessors (MP), each with:
- several simple cores
- small shared memory

The threads executing in the same MP must execute the same instruction

Shared memory must be used to prevent the high latency of the global device memory



Source & Acknowledgements : NVIDIA, References

Glance at NVIDIA GPU's

- NVIDIA GPU Computing Architecture is a separate HW interface that can be plugged into the desktops / workstations / servers with little effort.

- G80 series GPUs /Tesla deliver FEW HUNDRED to TERAFLOPS on compiled parallel C applications

GeForce 8800

Tesla D870

Tesla S870

Source : NVIDIA, References

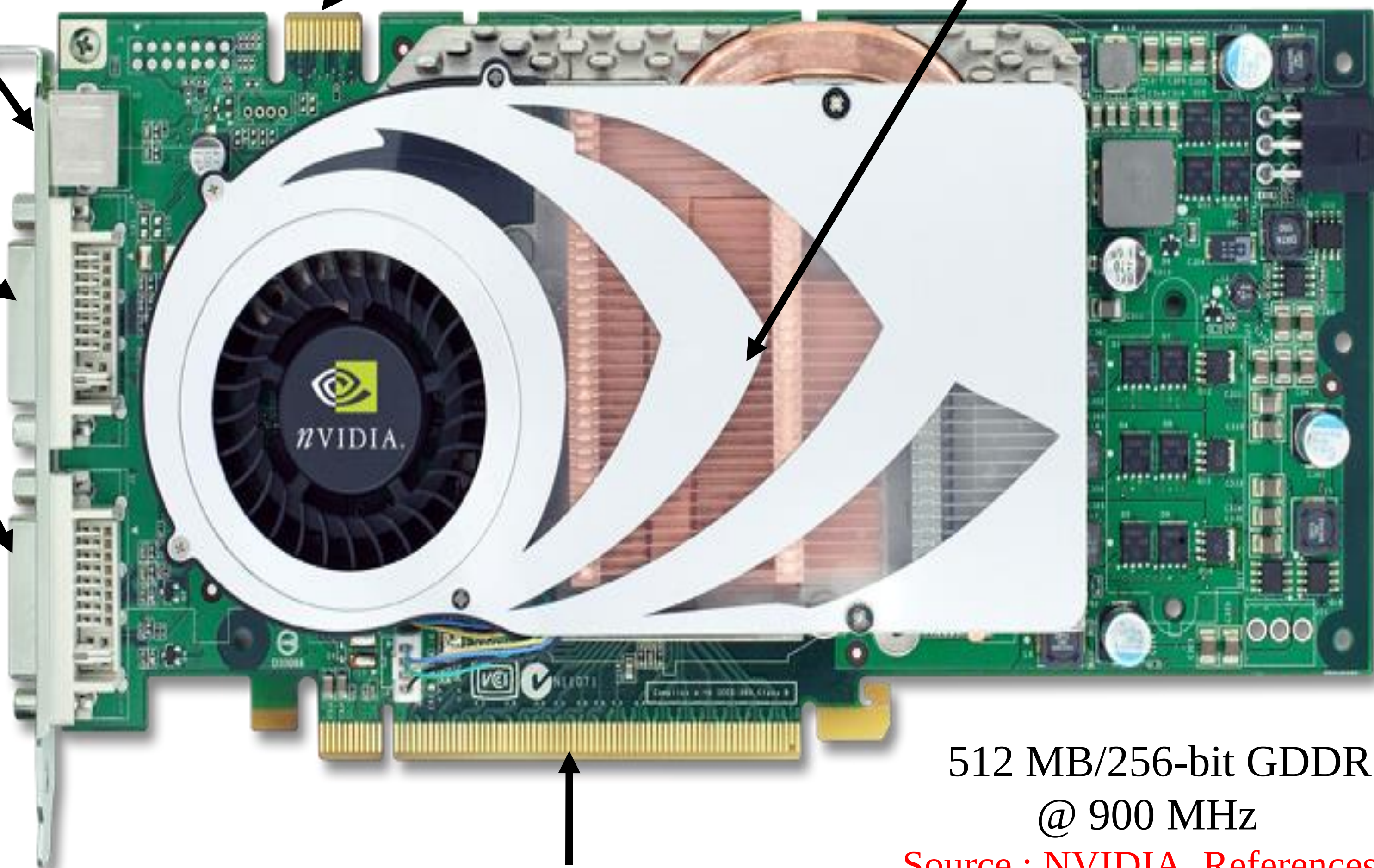
GeForce 8800 GT Card

sVideo
TV Out

SLI Connector

Single slot cooling

DVI x 2



512 MB/256-bit GDDR3
@ 900 MHz

Source : NVIDIA, References

16x PCI-Express

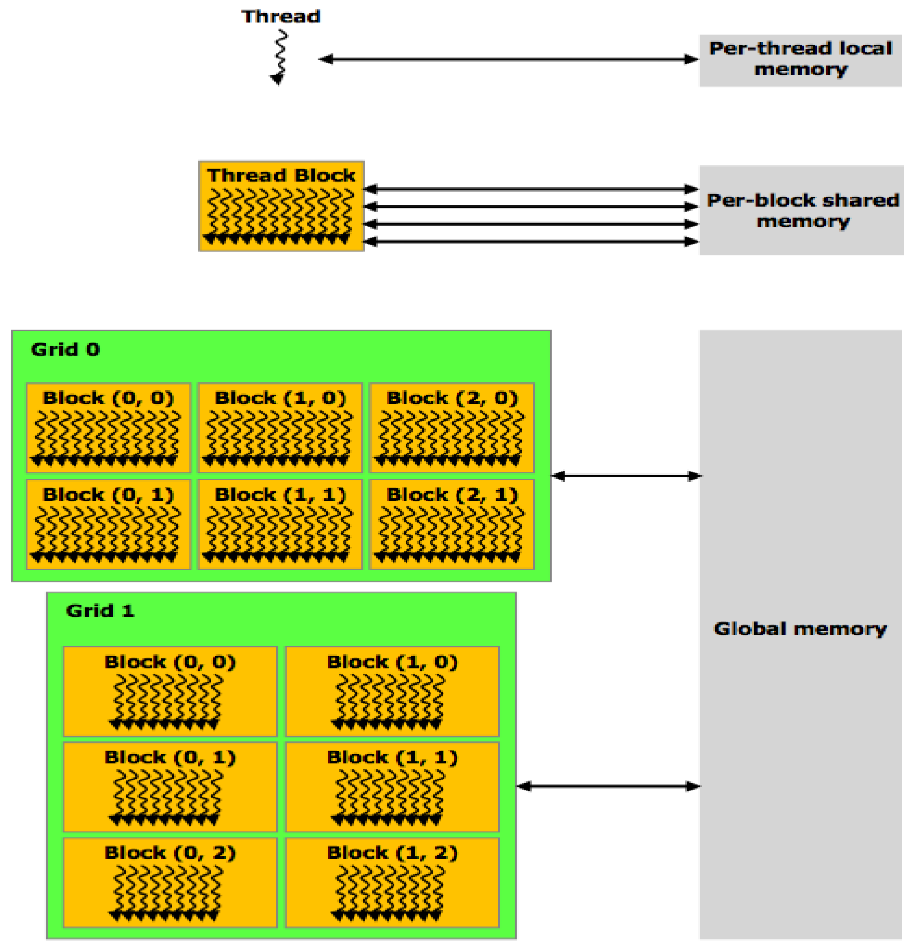
GPU Thread Organisation

Reflects the memory hierarchy of the device

All threads from a single block are executed in the same MP

Shared memory:
- Used for communication and synchronization of thread of the same block

How to map neuronal processing and communications into CUDA threads?



Source & Acknowledgements : NVIDIA, References

NVIDA :CUDA – Data Parallelism

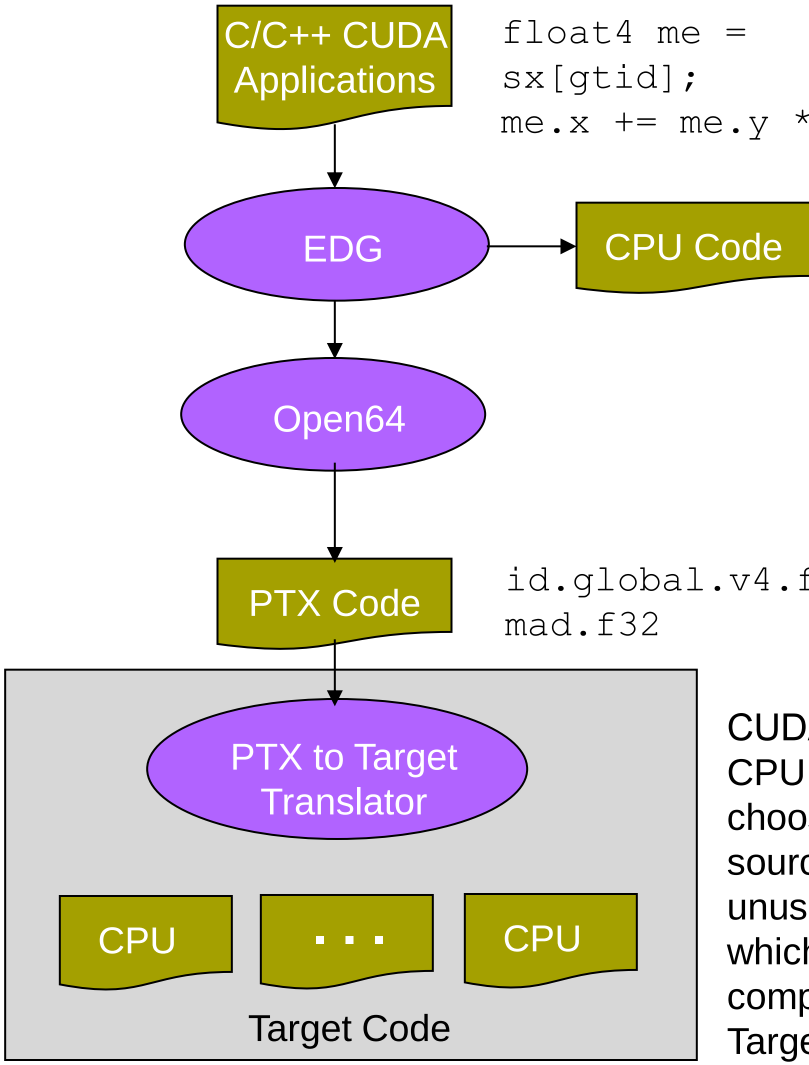
❖ ***Data Parallelism***

- It refers to the program property whereby many arithmetic operations can be safely performed on the data structure in a simultaneous manner.

❖ The concept of ***Data Parallelism is applied to typical matrix-matrix computation.***

Source & Acknowledgements : NVIDIA, References

NVIDIA GPU Computing - CUDA Kernels and Threads



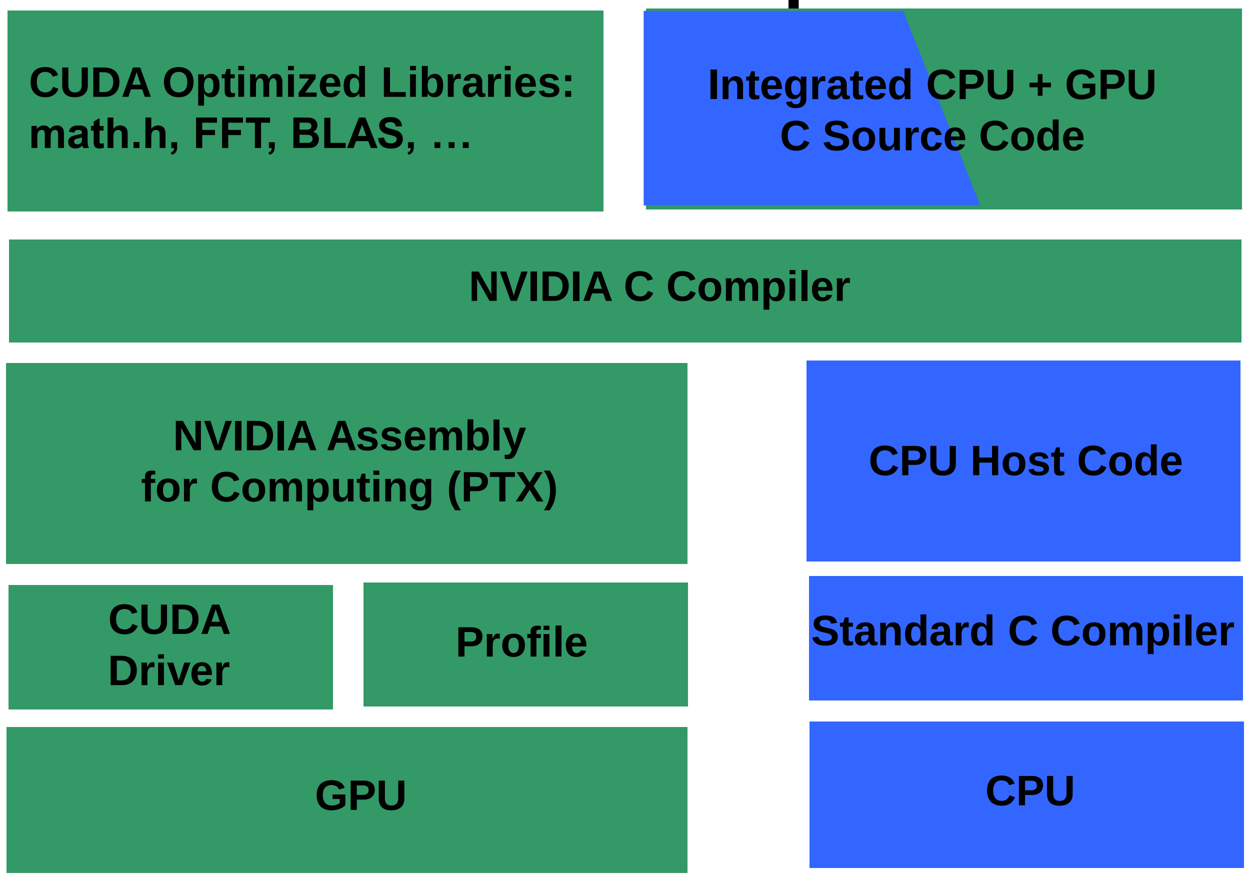

```
float4 me =
sx[gtid];
me.x += me.y * me.z;
```

```
id.global.v4.f31   {$f1,$f3,$f5,$f7),
mad.f32            [$r9+0};
                   $f1, $f5, $f3, $f1;
```

CUDA's compilation process. Source code written for the host CPU follows a fairly traditional path and allows developers to choose their own C/C++ compiler, but preparing the GPU's source code for execution requires additional steps. Among the unusual links in the CUDA tool chain are the EDG preprocessor, which separates the CPU and GPU source code; the Open54 compiler, originally created for itanium; and Nvidia's PTX-to-Target Translator, which converts Open64's assembly-language output into executable code for specific Nvidia GPUs.

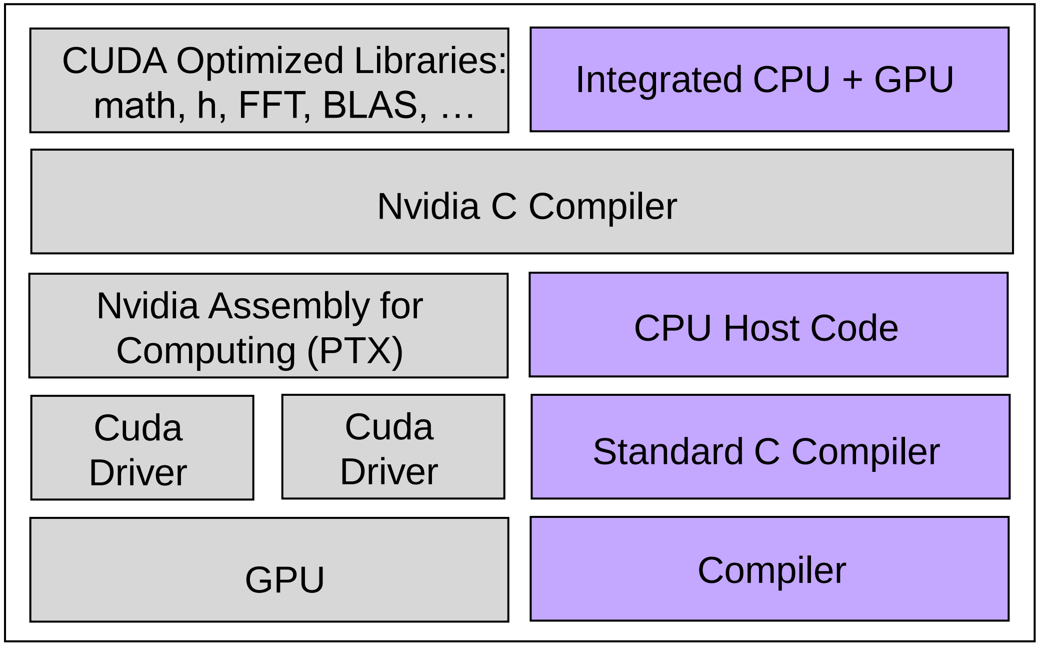
Source & Acknowledgements : NVIDIA, References

CUDA Software Development



Source & Acknowledgements : NVIDIA, References

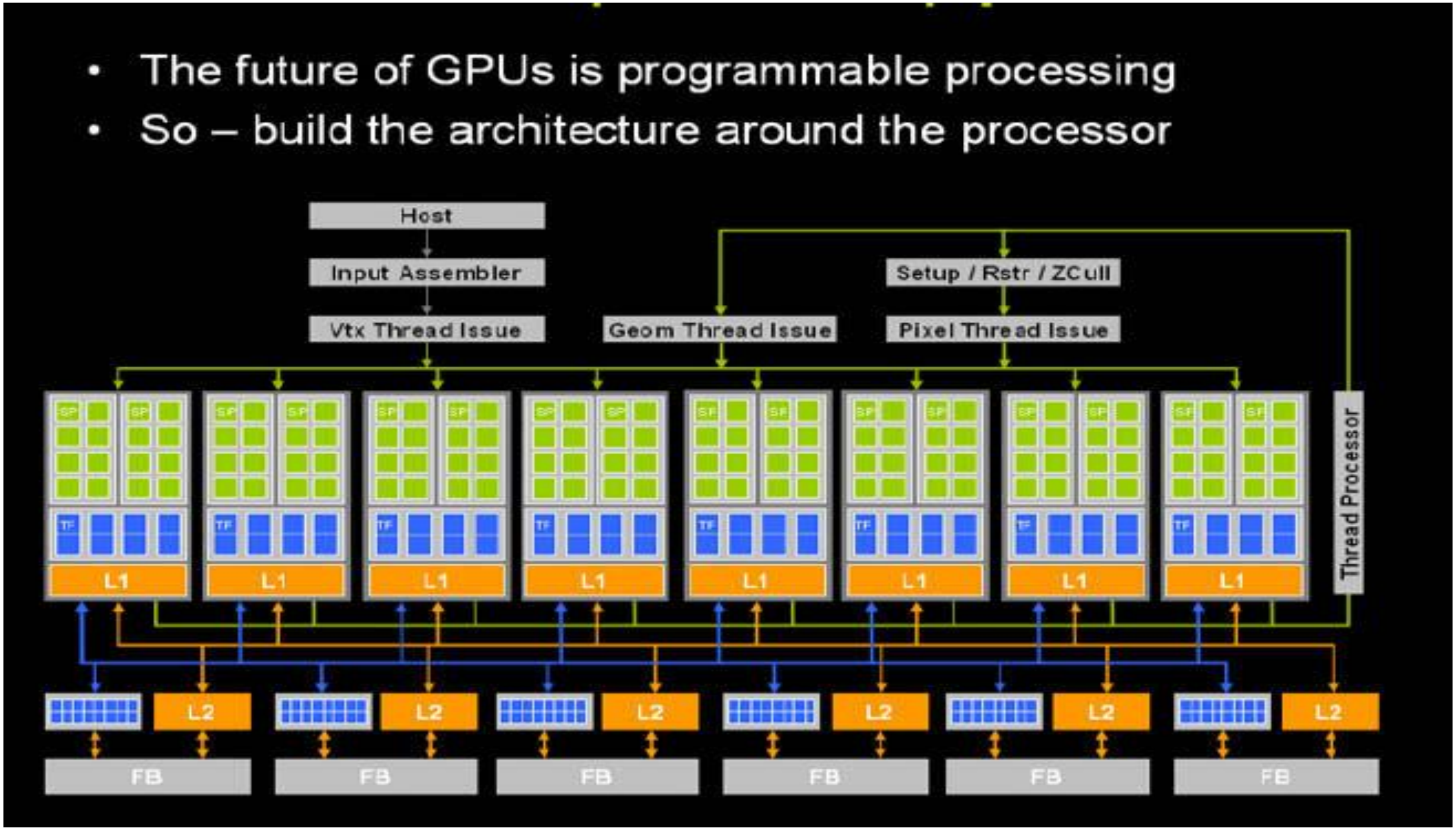
CUDA Performance Advantage



NVIDIA CUDA platform for parallel processing on Nvidia GPUs. Key elements are common C/C++ source code with different compiler forks for CPUs and GPUs; function libraries that simplify programming; and a hardware-abstraction mechanism that hides the details of the GPU architecture from programmers.

Source : NVIDIA, References

NVIDIA GeForce GPU



Source & Acknowledgements : NVIDIA, References

CUDA PROGRAM STRCUTURE

❖ ***A CUDA*** program consists of one or more phases that are executed on either the **host** (CPU) or a **device** such as **GPU**.

- The phases that exhibit **little** or **no data parallelism** are implemented in the **host** code.
- The phases **rich amount of data** parallelism are implemented in the **device** code.

❖ ***A CUDA*** program is a unified source code encompassing both **host** and **device** code.

❖ The NVIDIA C Compiler (**nvcc**) separates the two during the compilation process. The ***host-code i***s straight **ANSI C** code

❖ The ***device code*** is written using ANSCI key-words for labeling ***data-parallel*** functions called **kernels** and their associated data structures.

Source & Acknowledgements : NVIDIA, References

An approach to Writing CUDA Kernels

- ❖ Use algorithms that can expose substantial parallelism, you'll need thousands of threads…
- ❖ Identify ideal GPU memory system to use for kernel data for best performance
- ❖ Minimize host/GPU DMA transfers, use pinned memory buffers when appropriate
- ❖ Optimal kernels involve many trade-offs, easier to explore through experimentation with microbenchmarks based key components of the real science code, without the baggage
- ❖ Analyze the real-world use cases and select the kernel(s) that best match, by size, parameters, etc.

Source : NVIDIA, References

Processor Terminology

- SPA
  - ✓ Streaming Processor Array (variable across GeForce 8-series, 8 in GeForce8800)
- TPC
  - ✓ Texture Processor Cluster (2 SM + TEX)
- SM
  - ✓ Streaming Multiprocessor (8 SP)
  - ✓ Multi-threaded processor core
  - ✓ Fundamental processing unit for CUDA thread block
- SP
  - ✓ Streaming Processor
  - ✓ Scalar ALU for a single CUDA thread

Source : NVIDIA, References

NVIDA :CUDA – Data Parallelism

❖ ***Data Parallelism*** : It refers to the program property whereby many arithmetic operations can be safely performed on the data structure in a simultaneous manner

❖ Example : The concept of Data Parallelism is applied to typical matrix-matrix computation.

❖ Each element of the product matrix P is generated by performing a dot product between a row of input matrix M and a column of input matrix N as shown in figure.

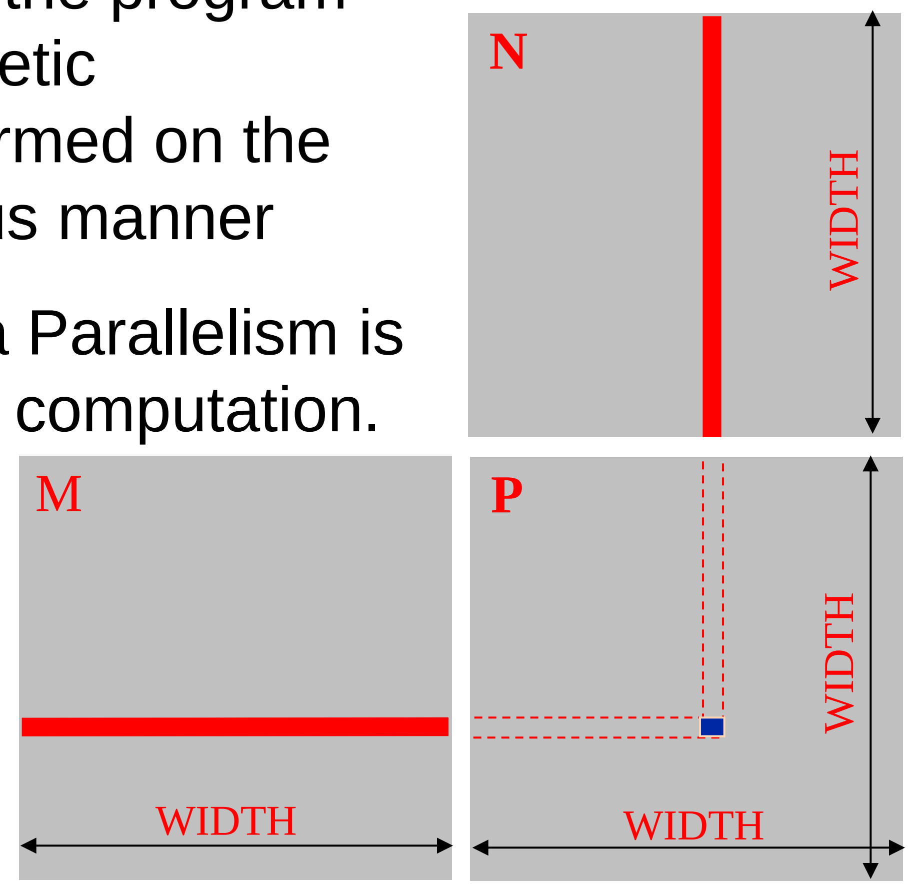


Figure Data parallelism in matrix multiplication.

Source & Acknowledgements : NVIDIA, References

NVIDA :CUDA – Data Parallelism

- In figure, highlighted elements of a matrix **P** is generated by taking the dot product of the highlighted row of matrix **M** and the highlighted column of matrix **N**

- **Note :** Dot product operations for computing different matrix **P** elements can be simultaneously performed.
  - None of these dot products will affect the results of each other.

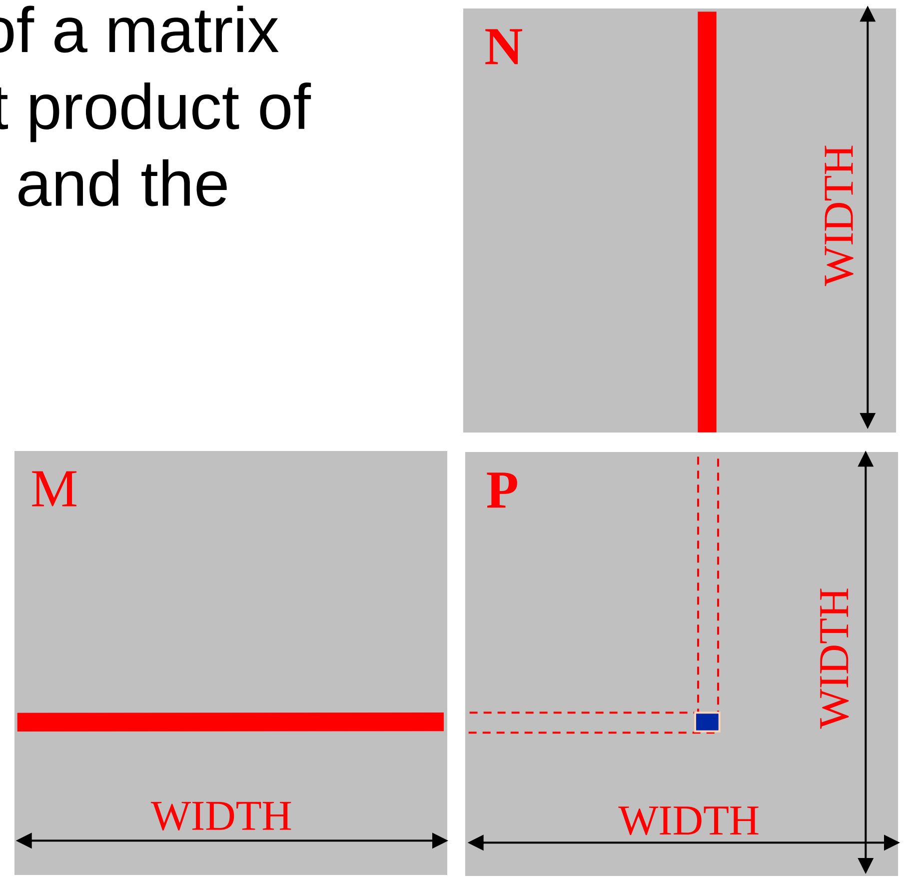


Figure Data parallelism in matrix multiplication.

Source & Acknowledgements : NVIDIA, References

NVIDA :CUDA – Data Parallelism

- For **P** = (1000 X 1000); **M** = (1000 X 1000) & **N** = (1000 X 1000)
- The number of dot products : 1,000,000
- Each dot product involves 1000 multiply and 1000 accumulate arithmetic operations

Note :

1. Data Parallelism in real application is not as simple as matrix-matrix multiplication.
2. Different forms of Data parallelism exists in several applications

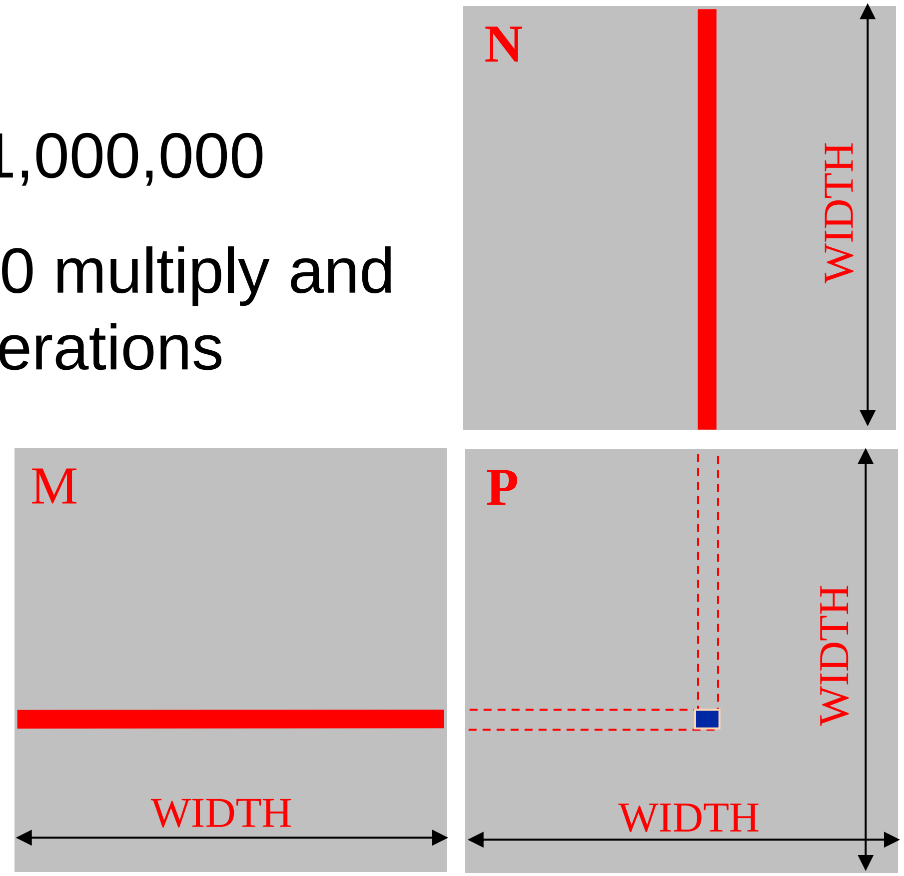


Figure : Data parallelism in matrix Multiplication.

Source: NVIDIA

CUDA PROGRAM STRCUTURE

❖ The device code is complied by the ***nvcc*** and executed on a **GPU** device.

- Refer CUDA Software Development Kit **(SDK)** are implemented in the **host** code.

❖ ***About Kernel function*** :

- Generate a large number of threads to exploit parallelism

- In Matrix into Matrix Multiplication algorithm, the kernel that uses one thread to compute one element of output matrix **P** would generate ***1,000,000 threads*** when it is invoked.

Source & Acknowledgements : NVIDIA, References

CUDA PROGRAM STRCUTURE

Remarks :

- ❖ CUDA threads are of much lighter weight than the CPU threads

- ❖ It can be assumed that these threads take **very few cycles** to generate and schedule due to efficient hardware support.

  - Note : CPU threads that typically require thousands of clock cycles to generate and schedule.

  - When kernel function is invoked or launched, all the **threads** that are generated take advantage of **data parallelism.**

  - All the threads that are generated by a kernel during an invocation are collectively called a **grid.**

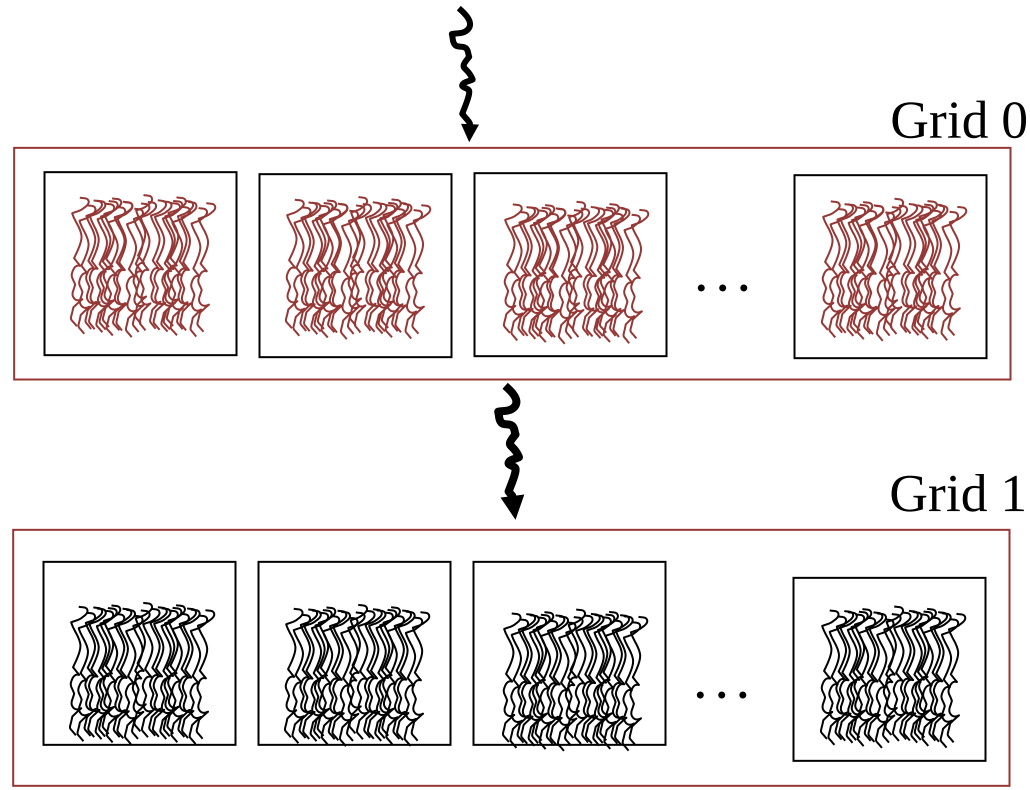
Source & Acknowledgements : NVIDIA, References

CPU serial code

GPU parallel kernel
Kernel<<<nBIK, nTid>>>(args);

CPU serial code

GPU parallel kernel
Kernel<<<nBIK, nTid>>>(args);



Execution of a CUDA program.

- ❖ Figure shows the execution of **two grids** of threads. When all the threads of a kernel complete their execution, the corresponding grid terminates, and the execution continues on the host until another kernel is invoked.

Source & Acknowledgements : NVIDIA, References

NVIDA :CUDA STRUCTURE

Example 1. : Matrix Multiplication

```
int main (void) {
  Step 1 : // allocate and the initialize the matrices M,N, P
           // I/O read the input matrices M & N
           ………….

  Step 2 : // M * N  on the device
            MatrixMultiplication (M,N,P, Width)

  Step 3 : // I/O  to write the  Output matrix P
           // Free matrices M,N, P
      ………

 return 0;

 }
```


Source: NVIDIA

Example : Matrix Multiplication

```
Void MatrixMultiplication(float*  ,float*        *  ,int     )
{
  for (int i = 0; i < Width; ++i)
    for (int j = 0; j < Width; ++j) {
       float sum = 0;
       for (int k = 0; k < Width: ++k) {
          float a = M[i * Width + k];
          float b = N[k * Width + j];
          sum += a = b;
       }
       P[i * width + j] = sum;
    }
}
```

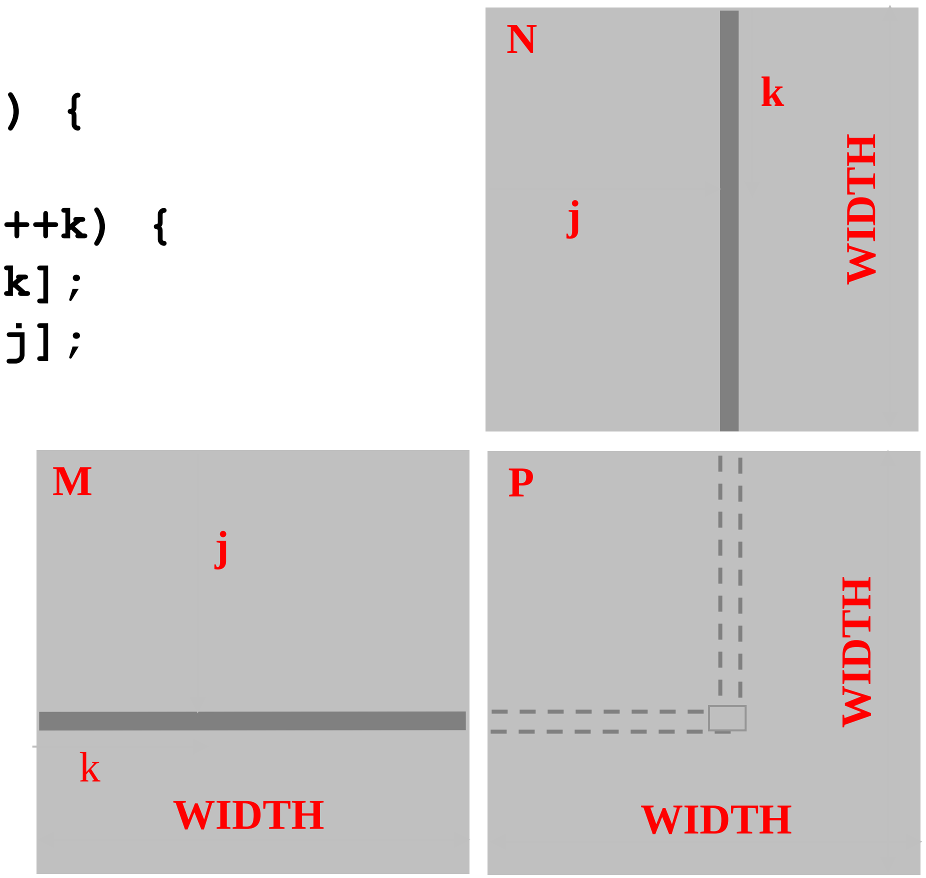


Figure A simple matrix multiplication function with only host code.

Example : Matrix Multiplication

Note : 4 x 4 matrix is placed into 16 consecutive memory locations (Simple code can be written using Standard C language.)

$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$
$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$
$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$
$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$

M

$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$	$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$	$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$	$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$

Placement of two-dimensional array elements into the linear address system memory.

NVIDIA :CUDA STRUCTURE

Example 2: Matrix Multiplication

Revised host code simple matrix multiplication that moves the matrix multiplication to a device

```
Void MatrixMultiplication(float* M,float* N,float* P,int Width)
{
   int size = Width * Width *sizeof(float);
   float* Md, Nd, Pd;
   ....................
   Step 1: // Allocate device memory for M, N, and P
           // copy M and N to allocate device memory locations

   Step 2: // Kernel invocation code – to have the device to
           // perform the actual matrix multiplication

   Step 3:  // copy P from the device memory
           // free device matrices
}
```

Source & Acknowledgements : NVIDIA, References

CUDA Architecture

CUDA Device Memories and Data Transfer

– **Processor:**

– Set of Multi-Processors (MP)

– Set of Scalar Processor (SP)

– **Memory:**

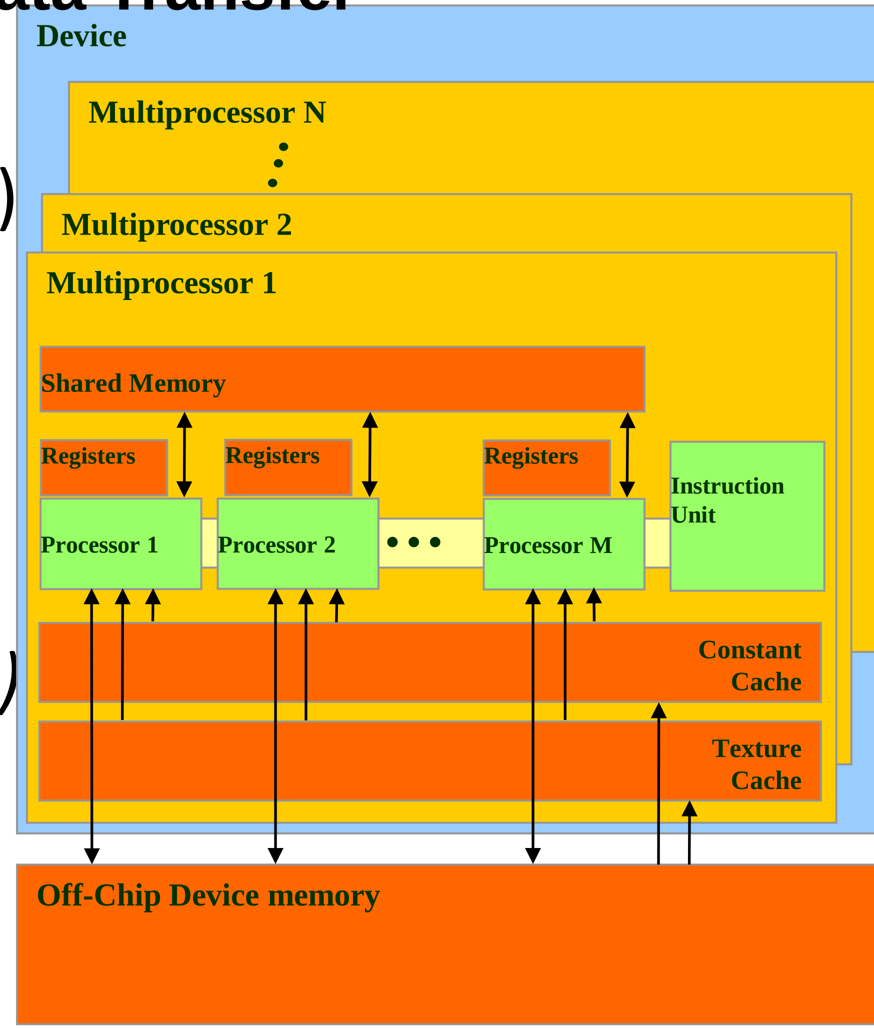
– High b/w global memory

– Fast shared memory *(per SP)*

– **Execution:**

– *Kernel program* on GPU

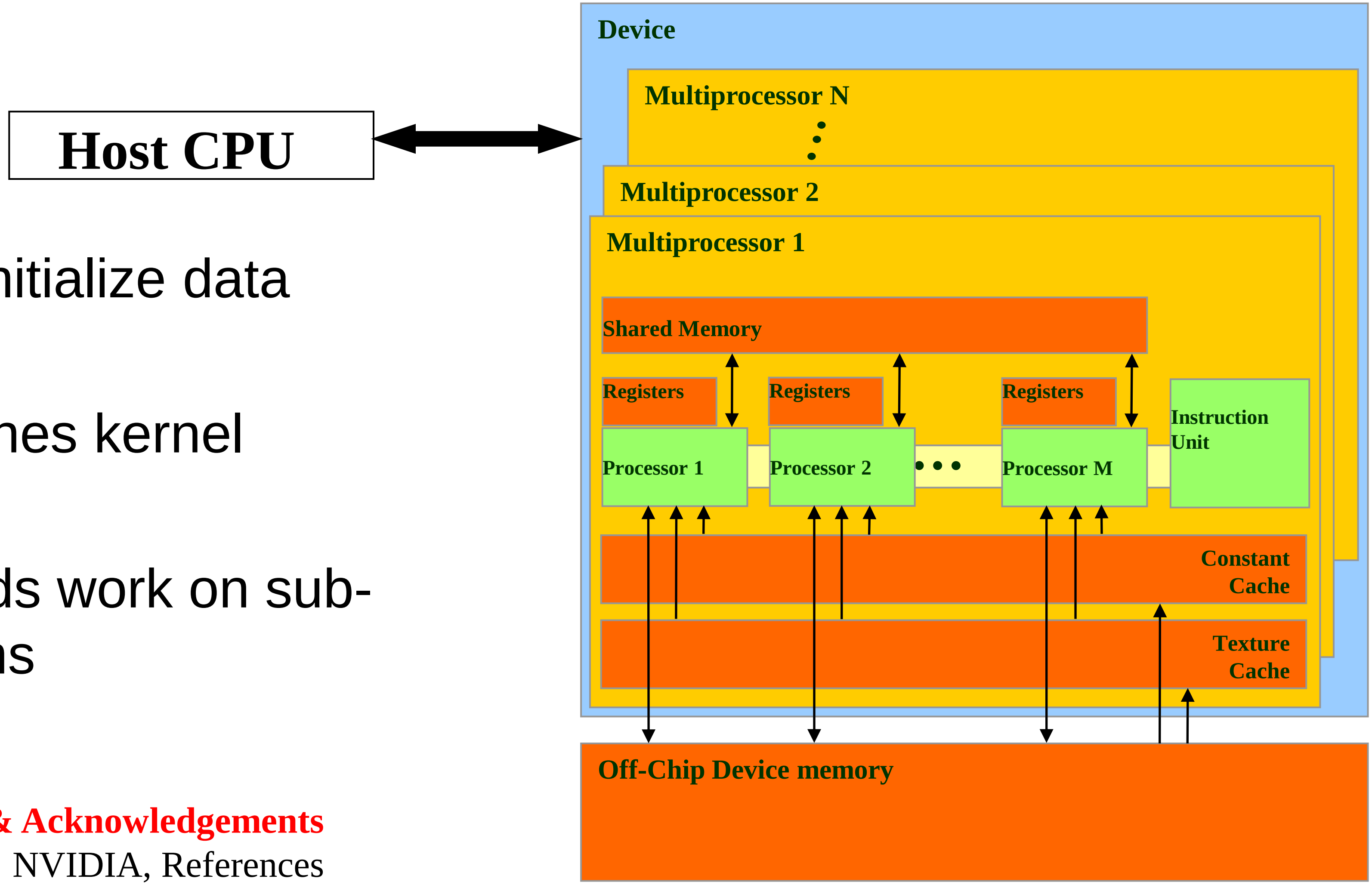
– Threads scheduling in warps



Source & Acknowledgements : NVIDIA, References

Basic Implementation on GPU

CUDA Device Memories and Data Transfer



CPU initialize data

Launches kernel

Threads work on sub-streams

Source & Acknowledgements : NVIDIA, References

Source & Acknowledgements : NVIDIA, References

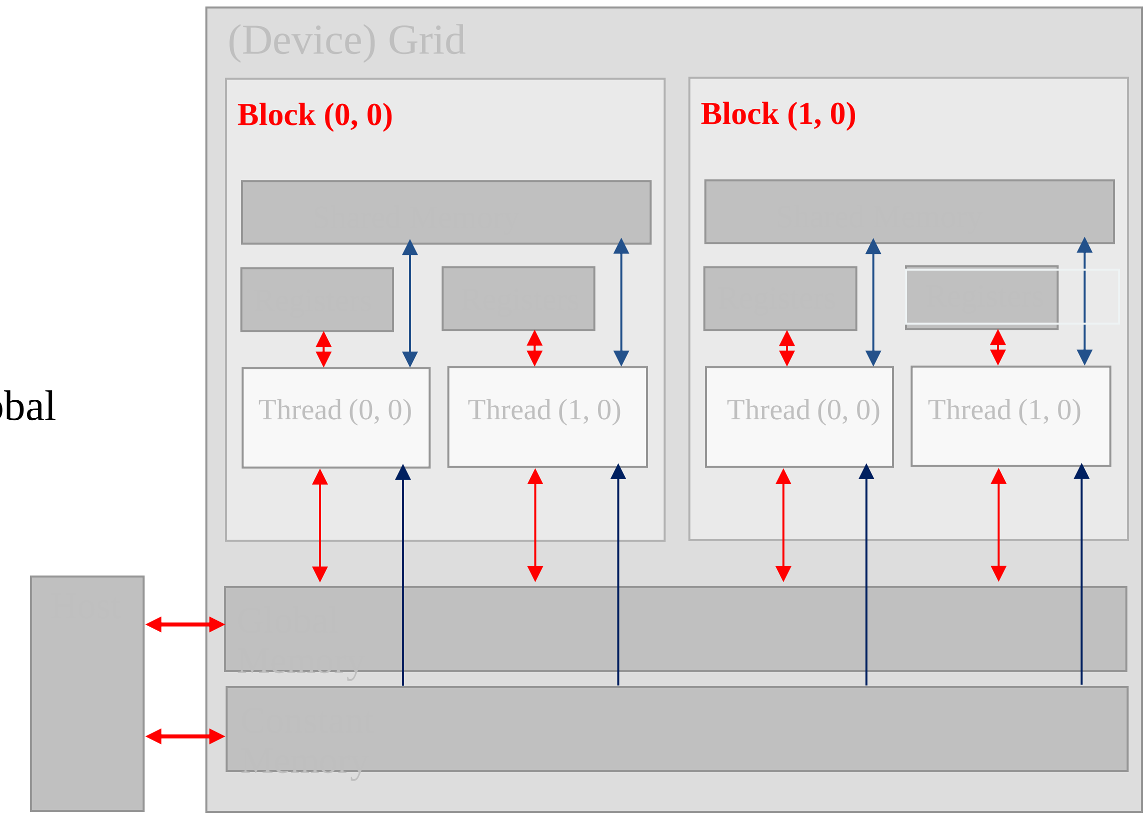
NVIDA :CUDA DEVICE MEMORIES & DATA TRANSFER

CUDA device memory model & Data transfer

- **Device code can:**
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-gold constant

- **Host code can**
  - Transfer data to/from per-grid global and constant memories

❖ global memory & constant memory -devices host code can transfer to and from the device, as illustrated by the bi-directional arrows between these memories and host



Host memory is not shown in the figure

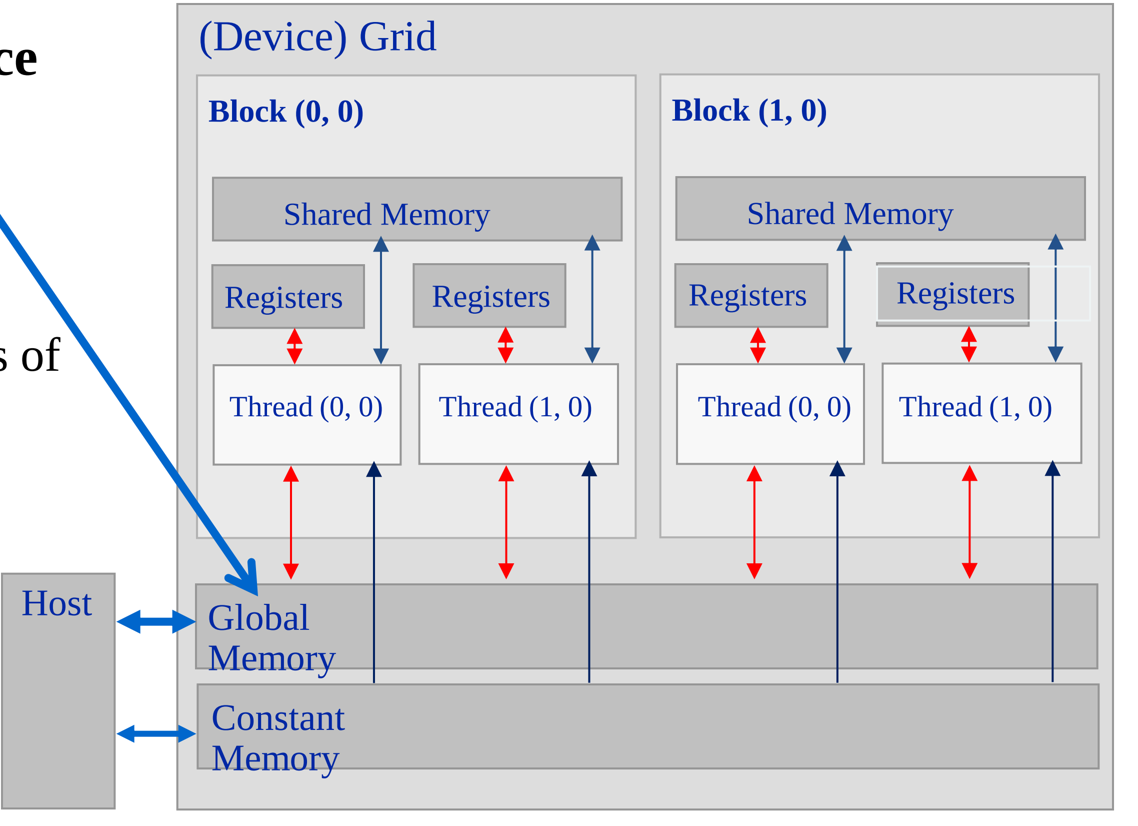
Source & Acknowledgements : NVIDIA, References

NVIDA :CUDA DEVICE MEMORIES & DATA TRANSFER

CUDA device memory model & data transfer

- **cudaMalloc()**
  - **Allocates object in the device global memory**
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** allocated object terms of bytes

- **cudaFree ()**
  - Frees object from device global memory
    - Pointer to freed object



CUDA API functions for device global memory management

Source & Acknowledgements : NVIDIA, References

Example : Matrix Multiplication

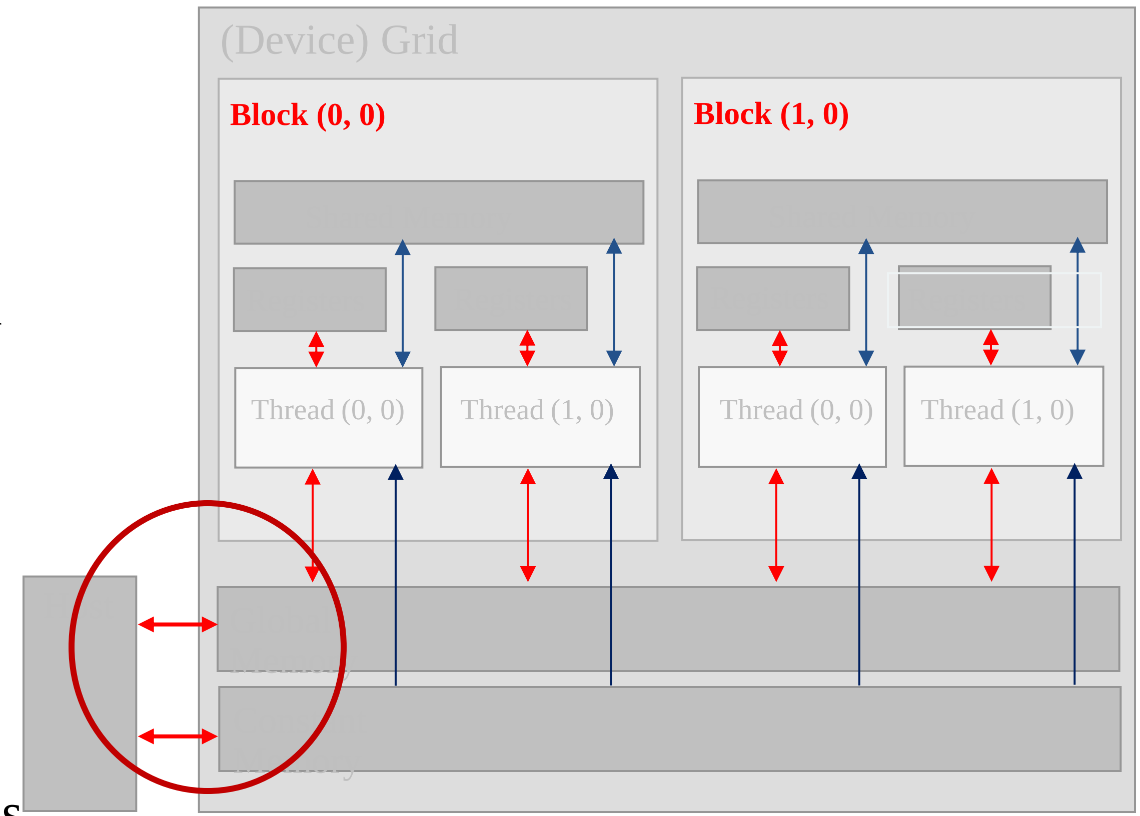
```
Void MatrixMultiplication(float* M,float* N,float* P,int Width)
{
   int size = Width * Width *sizeof(float);
   float* Md, Nd, Pd;
   ......................
   Step 1: // Allocate device memory for M, N, and P
           // copy M and N to allocate device memory locations

   Step 2: // Kernel invocation code - to have the device to
           // perform the actual matrix multiplication

   Step 3:  // copy P from the device memory
           // free device matrices
}
```

Source & Acknowledgements : NVIDIA, References

NVIDA :CUDA DEVICE MEMORIES & DATA TRANSFER

CUDA device memory model & data transfer

- **cudaMemcpy()**
  - **Memory data transfer**
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
    - Transfer is asynchronous



CUDA API functions for data transfer between memories

Source & Acknowledgements : NVIDIA, References

Device Memory & Data transfer

`cudaMalloc()` : Called from the host code to allocate a piece of global memory for an object.

```
float* Md
int size = Width * Width *sizeof(float);
cudaMalloc( (void**)&Md, size);
…………….
cudaFree(Md);
…………………
```

1. The first parameter of the **`cudaMalloc()`** function is the address of a pointer variable that must point to the allocated object after allocation
2. The second parameter of **`cudaMalloc()`** function gives size of the obejct to be allocated.
3. After the computation, **`cudaFree()`** is called with pointer **`Md`** as input to free the storage space for the Matrix from the device global memory.

Device Memory & Data transfer

CUDA Programming Environment : Two symbolic constants

```
cudaMemcpy(Md,M,size, cudaMemcpyHostToDevice);

cudaMemcpy(P,Pd,size, cudaMemcpyDeviceToHost);
```

are predefined constants of the CUDA Programming Environment.

Note : The `cudaMemcpy()` function takes four parameters

1. The first parameter is a pointer destination location for the copy operation
2. The second parameter points to the source data object to be copied
3. The third parameter specifies the number of bytes to be copied
4. The fourth parameter indicates the types of memory involved in the copy: from the *host memory to host memory; from host memory to device memory; from device memory to host memory*

Note : Please note that **cudaMemcpy() cannot** be used to copy between different GPUs to multi-GPU systems.

Source & Acknowledgements : NVIDIA, References

Device Memory & Data transfer

The revised MatrixMultiplication() function Code

```
Void MatrixMultiplication(float* M,float* N,float* P,int Width)
{
   int size = Width * Width *sizeof(float);
   float* Md, Nd, Pd;
  Step 1. // Transfer of M and N to device memory
          cudaMalloc( (void**)&Md, size);
          cudaMemcpy(Md,M,size, cudaMemcpyHostToDevice);
          cudaMalloc( (void**)&Nd, size);
          cudaMemcpy(Md,M,size, cudaMemcpyHostToDevice);
          // Allocate P on the device
          cudaMalloc ( (void**) &Pd, size)
  Step 2. // Kernel Invocation code
          …………………….
  Step 3. // Transfer P from device to host
          cudaMemcpy(P,Pd,size, cudaMemcpyDeviceToHost);
          // free device matrices
          cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
}
```

Source & Acknowledgements : NVIDIA, References

KERNEL FUNCTIONS AND THREADING

- ***CUDA*** kernel function is declared by **"__global__"** keyword

  This function will be executed on the device and can only called from the host to generate a **grid of threads** on a device.

- Besides **"__global__" ,** there are two other keywords tha can be used in front of a function declaration**.**

  __device__ float DeviceFun( )

  __global__ void KernelFun( )

  __host__ float HostFunc( )

KERNEL FUNCTIONS AND THREADING

❖ ***CUDA*** extensions to C function declaration

__device__ float DeviceFun( ) : Declared as a **CUDA device** function)

__global__ void KernelFun( ) :Declared as a **CUDA kernel** function)

__host__ float HostFunc( ) :Declared as a **CUDA host** function)

	Executed on the :	**Only calling from the :**
__device__ float DeviceFun( )	device	device
__global__ void KernelFun( )	device	host
__host__ float HostFunc( )	host	host

Source & Acknowledgements : NVIDIA, References

KERNEL FUNCTIONS AND THREADING

The MatrixMultiplication() Kernel function

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd,
int Width)
{
    // 2D Thread ID
    Int tx = threadId.x;
    Int ty = threadId.y;
   // P value stores the Pd element that is computed by the
   // thread
   float Pvalue = 0;
   for (int k = 0; k < width; ++k) {
          float Mdelement  = Md[ty * width + k];
          float Ndelement  = Nd[k * width + tx];
          Pvalue  += Mdelement  * Ndelement;
       }
   // Write the matrix to device memory each thread writes one
   // element
   Pd[ty*Width + tx ] = Pvalue;
  }   // Limitation : Can handle only matrices of 16 elements in
each dimension
```

KERNEL FUNCTIONS AND THREADING

The MatrixMultiplication() Kernel function

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd,
int Width)
```

- ❖ Dot product loop uses `threadIdx.x` and `threadIdx.y` to identify the row of **Md** and column of **Nd** to work on

Limitations

- ❖ Can handle only matrices of 16 elements in each dimension (Due to fact that the kernel function does not use `blockIdx`)
- ❖ Limited to using only one block of threads
- ❖ It is assumed that each block can have upto 512 threads, we can limit to 16 X 16 because 32 X 32 requires more than 512 threads per block.
- ❖ **Question :** How to accommodate larger matrices ? (**Hint** : Use multiple thread blocks)

Source & Acknowledgements : NVIDIA, References

KERNEL FUNCTIONS AND THREADING

- **threadIdx.x** & **threadIdx.y**
  - Refer to the thread indices of a thread (Different threads will see different values in their **threadIdx.x** and **threadIdx.y** variables)
  - Refer thread as **Thread***threadIdx.x, threadIdx,y* Coordinates reflect a multi-dimensional organization for the threads.
  - CUDA threading hardware generates all of the **threadIdx.x** and t**hreadIdx.y** variables for each thread.
  - These work on particular part of data structure of the designed code and with these thread indices allow a thread to access the hardware registers at runtime that provides the identifying coordinates to the thread.

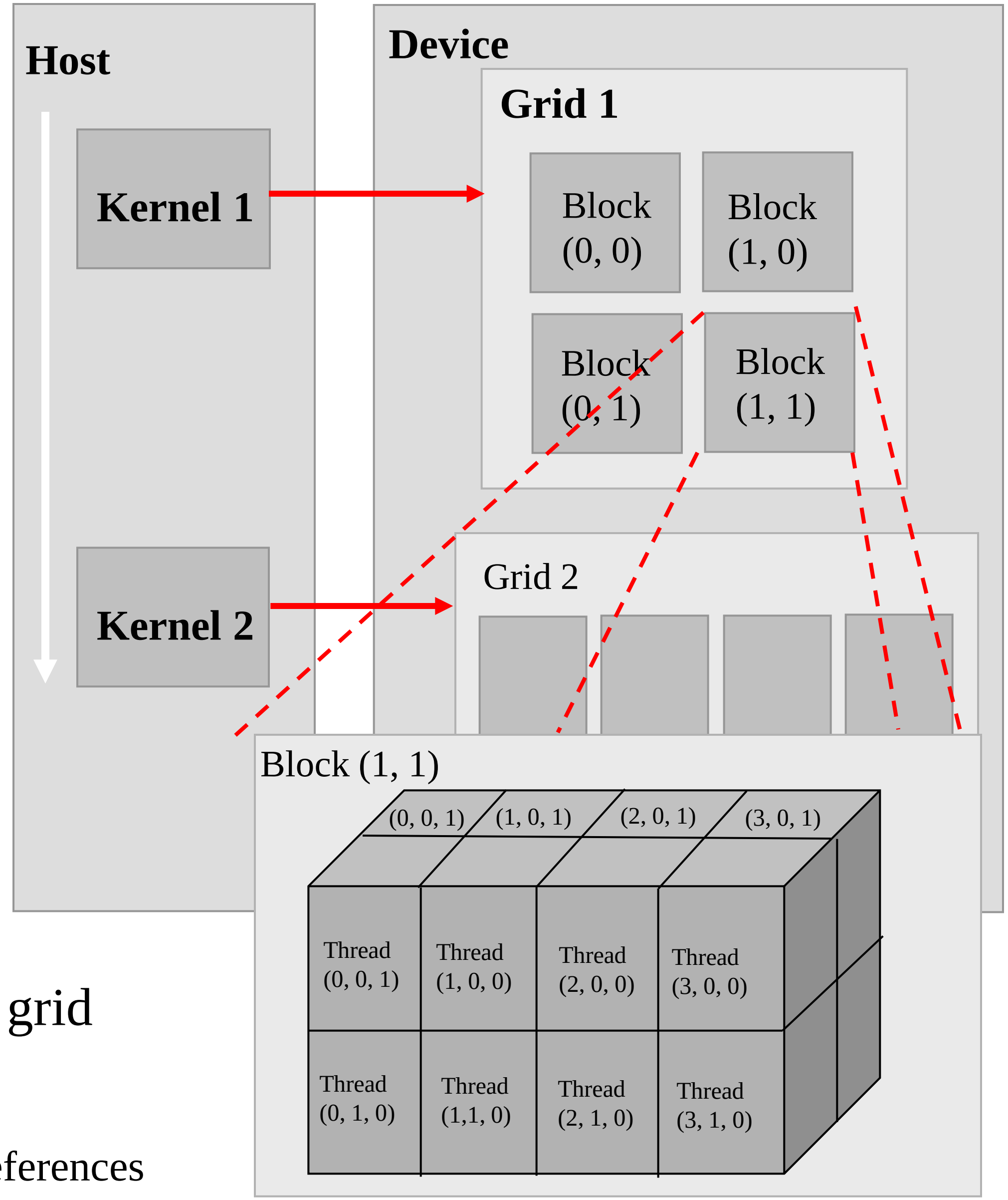
Source & Acknowledgements : NVIDIA, References

KERNEL FUNCTIONS AND THREADING

`threadIdx.x; threadIdx.y` in CUDA matrix multiplication

- Each thread uses its **`threadIdx.x`** and **`threadIdx.y`** to identify the row of **Md** and the column of **Nd** to perform the dot product operation.
- Each thread also uses its ***`threadIdx.x`*** and ***`threadIdx.y`*** values to select the **Pd** element that it is responsible for; for example ***threadId***$_{2,2}$ will perform a dot product between column 2 of **Nd** and row 3 of **Md** and write the result into element (**2,3**) of **Pd**. This way, the threads collectively generate all the elements of the **Pd** matrix.
- When a kernel is invoked or launched, it is executed as ***`grid`*** of parallel threads & each CUDA thread grid typically is comprised of thousands to millions of lightweight GPU threads per kernel invocation.

Source & Acknowledgements : NVIDIA, References



❖ **A Thread block**

- A thread block is a batch of threads that can co-operate with other by
  - Synchronizing their execution
    - ➢ For hazard-free shared memory accesses
- Efficiently sharing data through a low-latency shared memory

❖ **Cop-operation - thread blocks**

- Two threads from two different blocks can not cooperate

A multidimensional example of CUDA grid organization.

Source & Acknowledgements : NVIDIA, References

Ex : Vector Vector Addition

```
// Kernel definition
_global_ void VecAdd(float* A, float* B, float* C)
{
   int i = threadIdx.x;
   c(i) = A[i] + B[i];
}

int main ()
{
   ...
   // Kernel invocation with N Threads
   VecAdd<<<1, N>>>(A, B, C);
   ...
}
```

Kernel

KERNEL FUNCTIONS AND THREADING

Organization of Threads in a grid – CUDA

❖ Threads in a grid are organized into a two-level hierarchy, as illustrated in figure (Refer earlier slide)

❖ At the top level, each grid consists of one or more thread blocks. All blocks in a grid have the same number of threads

- Example : In figure (Refer earlier slide), **Grid 1** is organized as a 2 X 2 array of 4 blocks.
  - Each block has a unique two-dimensional co-ordinate given by the CUDA specific keywords **blockIdx.x** and **blockId.y**
  - All thread blocks must have the **same** number of threads organized in the same manner

Source : NVIDIA

NVIDIA :CUDA THREAD ORGANIZATION

KERNEL FUNCTIONS AND THREADING

Organization of Each Thread block in a grid

- Each **thread block** is, in turn, organized as a **three** dimensional array of threads with a total size up to **512 threads**
- The coordinates of threads in a block are uniquely defined **three** thread indices : **threadIdx.x**, **threadIdx.y** and **threadIdx.z**
- **Note :** Not all applications will use all three (3) dimensions of a thread block
- **Example :** (Refer earlier slide)
  - Each **thread block** is organized into a 4 x 2 x 2 three-dimensional array of threads
  - This gives a **Grid** one (1) a total of 4 x 16 = 64 threads

Source : NVIDIA

KERNEL FUNCTIONS AND THREADING

Organization of Each Thread block in a grid

Example of host code that launches a kernel

```
//Setup the execution configuration
dim3 dimBlock(Width, Width);
dim3 dimGrid(1,1);

// Launch the device computation threads !
MatixmultKernel<<< dimGrid, dimBlock>>> (Md, Nd, Pd, Width);
```

Source & Acknowledgements : NVIDIA, References

NVIDIA :CUDA THREAD ORGANIZATION

KERNEL FUNCTIONS AND THREADING

`Observations - Example 4:` (Refer earlier slide 40)

- Code does not use any block index in accessing input and output data.
- Threads with the same **`threadIdx`** values from different blocks would end-up accessing the same input and output data elements.
- As a result, the kernel can use only one thread block.
- The **`theadIdx.x`** and **`threadIdx.y`** values are used to organize the block into a row-dimensional array of threads.

Source & Acknowledgements : NVIDIA, References

KERNEL FUNCTIONS AND THREADING

Observations - Example 4: (Refer earlier slide 40)

- Because a thread block can have only up to 512 threads, each thread calculates one element of the product matrix in Example 4, the code can only calculate a product matrix upto 512 elements.

- **Conclusions :**

  1. The solution is not scalable & not acceptable due to choice of one thread block
  2. To have a sufficient amount of data parallelism to benefit from execution on a device use of multiple blocks is required.

- **Question to be addressed**

  How to set the grid and thread block dimensions ?

  How to specify execution configuration parameters ?

Source : NVIDIA

KERNEL FUNCTIONS AND THREADING

```
Organization of Each Thread block in a grid
  //Setup the execution configuration
  dim3 dimBlock(Width, Width);
  dim3 dimGrid(1,1);
 // Launch the device computation threads !
 MatixmultKernel<<< dimGrid, dimBlock>>> (Md, Nd, Pd, Width);
```

- Two **`struct`** variable of type **`dim3`** are declared
  - The **first** is for describing the configuration of blocks, which are defined as 16 x 16 groups of threads.
  - The second variable, **`dimGrid,`** describes the configuration of the grid.

In this example, we have only (1 X 1) block in each grid.

Source & Acknowledgements : NVIDIA, References

Part-II(B)

An Overview of CUDA enabled NVIDIA GPUs:

CUDA Threads

Source & Acknowledgements : NVIDIA, References

NVIDIA :CUDA – Thread Organization

CUDA Thread Organization

- ❖ All threads in a grid execute the same kernel
  - ➢ Rely on unique coordinates to distinguish themselves from each other and to identity the appropriate portion of the data to process.
- ❖ The threads are organized into a two-level hierarchy using unique coordinates
  - ➢ **`blockIdx`** (for block index) and
  - ➢ **`threadIdx`** (for thread index)

    (Assigned to them by the CUDA runtime system)
  - ➢ The **`gridDim`** and **`blockDim`** are additional built−in, pre-initialized variables that can be accessed within kernel functions

Source & Acknowledgements : NVIDIA, References

NVIDIA :CUDA – Thread Organization

CUDA Thread Organization

- All threads in a grid execute the same kernel
  - Rely on unique coordinates to distinguish themselves from each other and to identity the appropriate portion of the data to process.
- Size /Dimension of Grid or Block
  - The `blockIdx` and `threadIdx` appear as built-in, preinitialized variables that can be accessed within kernel functions

CUDA Thread Organization

- The **yellow** color box of each threads block in Figure shows a fragment of the kernel code
  - Part of the input data is **read** and
  - Part of the output data is **write**

NVIDIA :CUDA – Thread Organization

CUDA Thread Organization

- ❖ The example figure consists of **`N`** thread blocks, each with a **`blockIdx.x`** value ranges from **`0`** to ***`N-1`***
  - ➢ Each block in-turn consists of `M` threads, each with a **`threadIDx.x`** value ranges from **`0`** to ***`M-1`***.
- ❖ All blocks at each grid level are organized as a **one-dimensional (1D) array**
- ❖ All threads within each block level are organized as a **1D array and** each grid has a total of **`N*M`** threads

Example : The black box of each thread block in figure 6 shows a fragment of the kernel code.

- The code fragment uses the

```
Int threadI =  blockId.x + blockDim.x + threadIdx.x;
```

to identify the part of (a) input data to read from and (b) the part of the (b) output data structure to write to.

NVIDIA :CUDA – Thread Organization

```
Dim3 dimGrid(128, 1,1);
Dim3 dimBlock(32,1,1,);

Kernel Function <<< dimGrid, dimBlock >>> (…);
```

You can also use

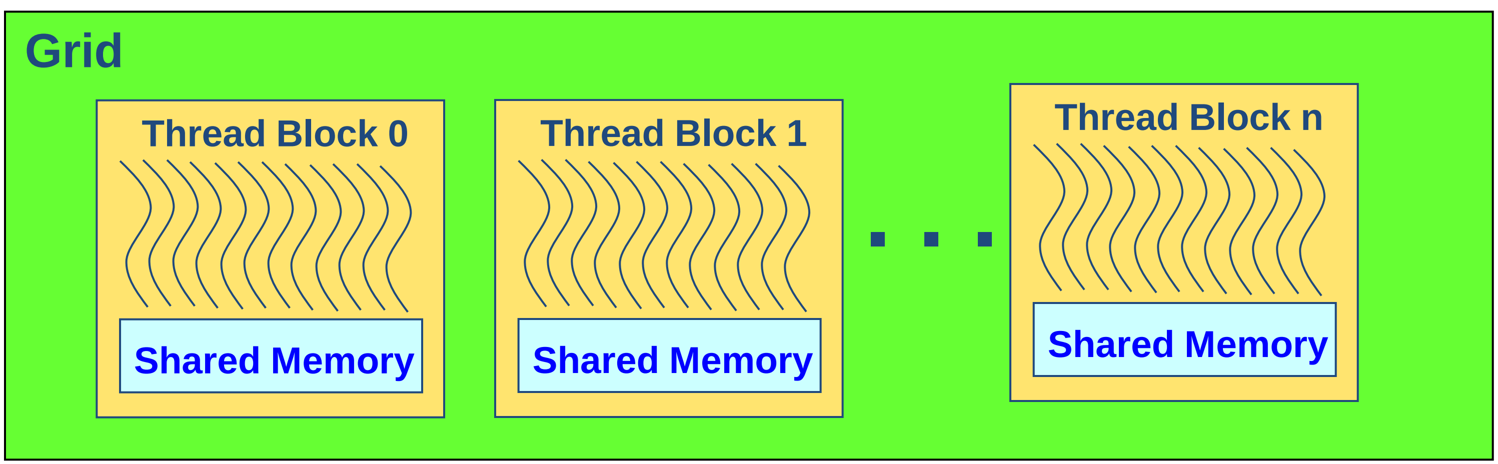
```
Kernel Function << 128, 32 >>> (…);
```

- The values of **gridDim.x** and **gridDim.y** can range from **1** to **65535**
- The values of **gridDim.x** and **gridDim.y** can be calculated based on other variables at kernel launch time.

Source & Acknowledgements : NVIDIA, References

Thread Batching

- ❖Kernel launches a grid of thread blocks
  - Threads within a block cooperate via shared memory
  - Threads within a block can synchronize
  - Threads in different blocks cannot cooperate
- ❖Allows programs to transparently scale to different GPUs



Source: NVIDIA

NVIDIA :CUDA – Thread Organization

CUDA Thread Organization

- The example figure consists of **`N`** thread blocks, each with a **`blockIdx.x`** value ranges from **`0`** to ***`N-1`***
  - Each block in-turn consists of `M` threads, each with a **`threadIDx.x`** value ranges from **`0`** to ***`M-1`***.

Example : The code fragment uses the

```
Int threadI =  blockId.x + blockDim.x + threadIdx.x;
```

to identify the part of (a) input data to read from and (b) the part of the (b) output data structure to write to.

Thread **`3`** of Block **`0`** has a **`threadId`** value of **`0*M + 3`**
Thread **`3`** of Block **`1`** has a **`threadId`** value of **`1*M + 3`**
Thread **`3`** of Block **`2`** has a **`threadId`** value of **`2*M + 3`**
Thread **`3`** of Block **`3`** has a **`threadId`** value of **`3*M + 3`**
Thread **`3`** of Block **`4`** has a **`threadId`** value of **`4*M + 3`**
Thread **`3`** of Block **`5`** has a **`threadId`** value of **`5*M + 3`**

NVIDA :CUDA – Thread Organization

CUDA Thread Organization

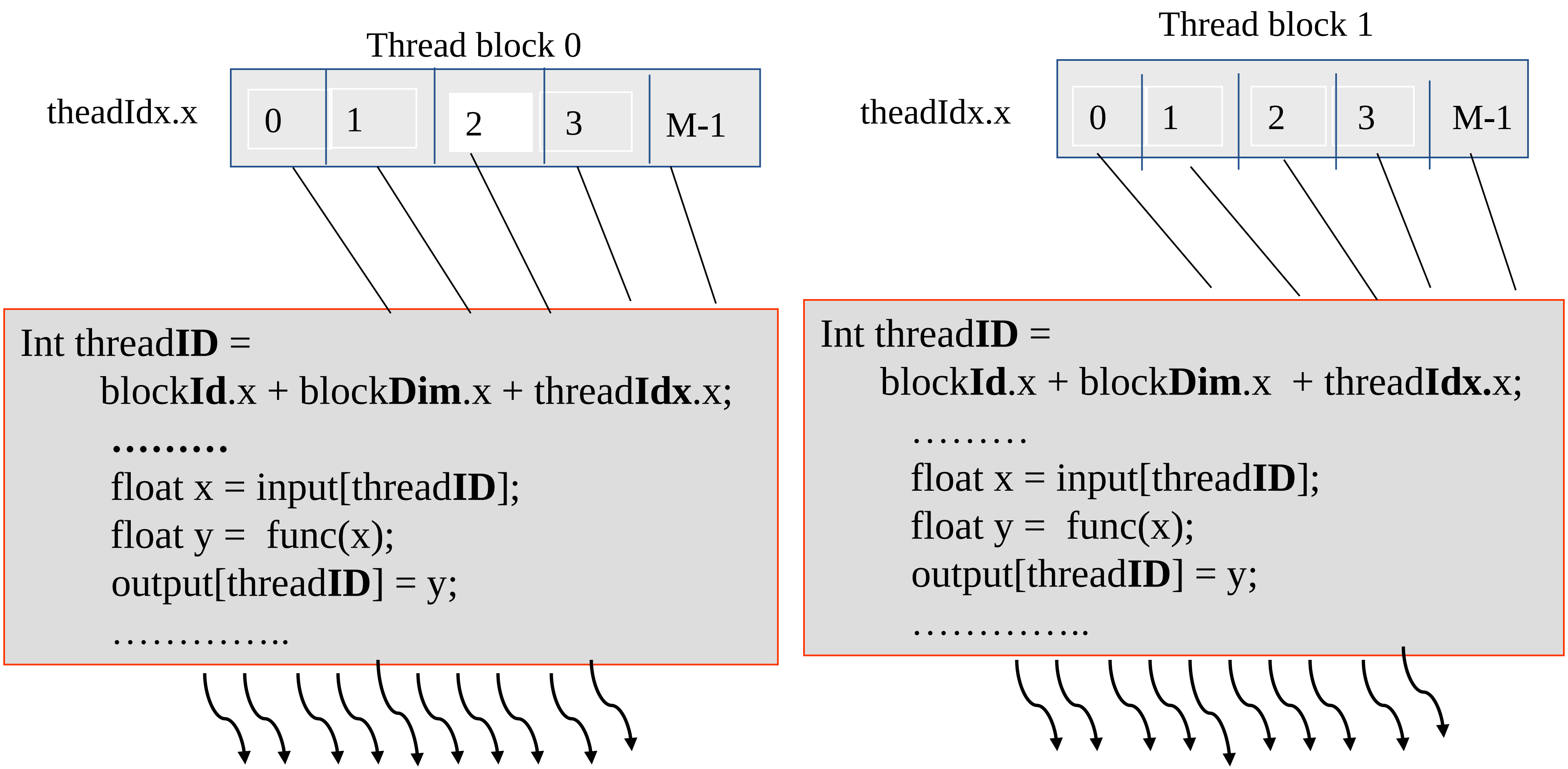
- The example figure consists of **N** thread blocks, each with a **blockIdx.x** value ranges from **0** to ***N-1***
  - Each block in-turn consists of M threads, each with a **threadIDx.x** value ranges from **0** to **M-1**.
- Each grid has a total of **N*M** threads

Example : Assume a each grid **128** blocks (**N = 128**) and each block has 32 (**M=32**) threads and a total of **128*32 = 4096** threads in the grid.

- Access to **blockDim** in the kernel function returns 32

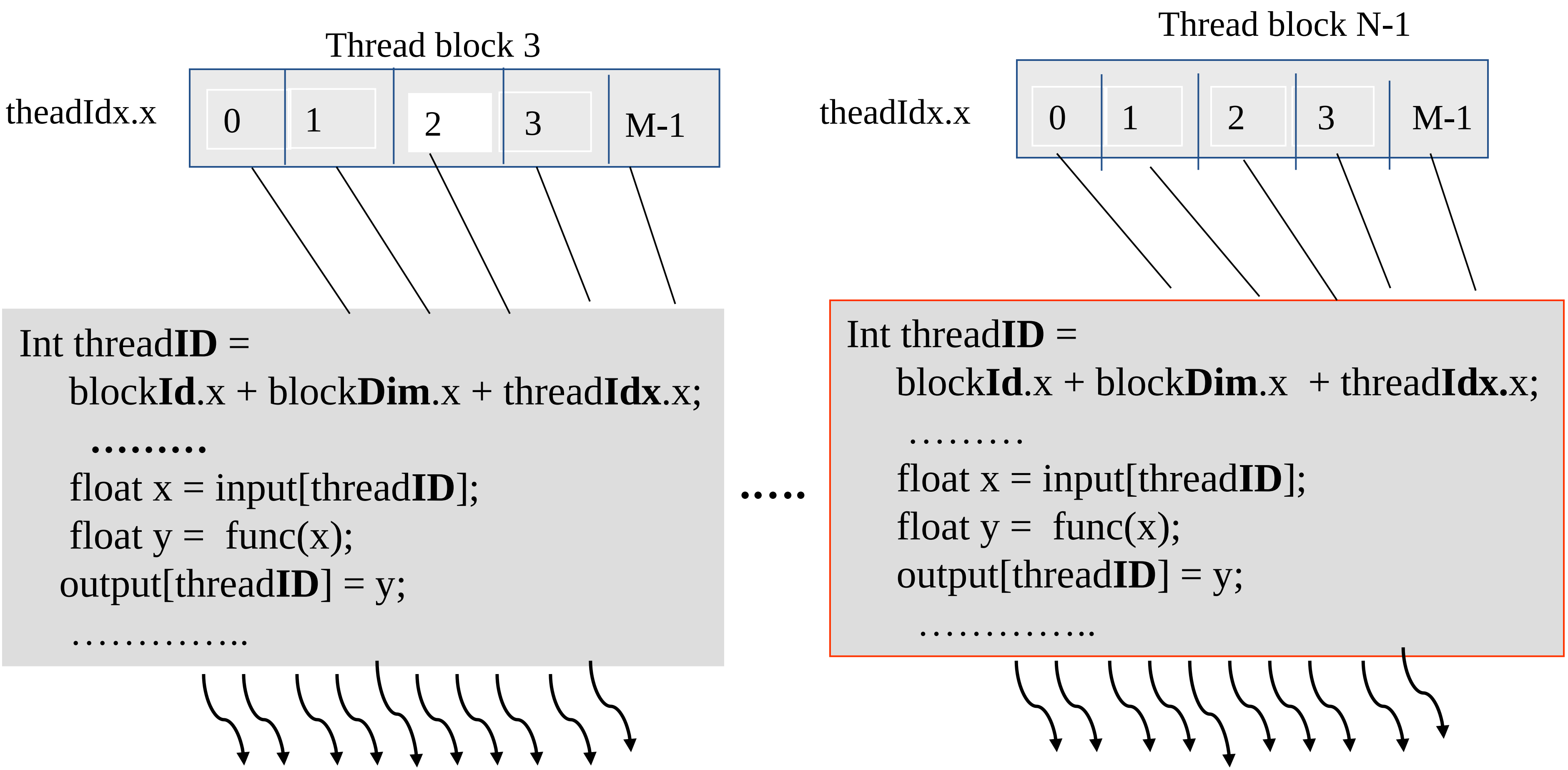
Thread **3** of Block **0** has a **threadId** value of **0*32 + 3 = 3**
Thread **3** of Block **4** has a **threadId** value of **4*32 + 3 = 131**
Thread **3** of Block **20** has a **threadId** value of **20*32 + 3 = 643**
Thread **3** of Block **40** has a **threadId** value of **40*32 +3 = 1283**
Thread **10** of Block **80** has a **threadId** value of **80*32+10 = 2570**
Thread **3** of Block **100** has a **threadId** value of **100*32+3 = 3203**
Thread **15** of Block **102** has a **threadId** value of **102*32+15 = 3279**
Thread **16** of Block **120** has a **threadId** value of **120*32+16 = 3856**

NVIDIA :CUDA THREAD ORGANIZATION



CUDA Thread Management – An Overview

NVIDIA :CUDA THREAD ORGANIZATION



CUDA Thread Management – An Overview

NVIDIA :CUDA – Thread Organization

- Each thread of the 4096 threads has its own unique threaded value

- Kernel code uses threadID variable to index into the **input[ ]** array and **output[ ]** arrays.

- If we assume that both arrays are declared with 4096 elements, then each thread may take one of the **input[ ]** of elements and produce one of the **output[ ]** elements

- Performance depends upon **input[ ]** array and **output[ ]** arrays

CUDA – Grid ; Host Code to launch the kernel

```
Dim3 dimGrid(128, 1,1);
Dim3 dimBlock(32,1,1,);
Kernel Function <<< dimGrid, dimBlock >>> (…);
```

The execution configuration parameters are between **<<<** and **>>>**

❖ The Scalar values can also be used for the execution configuration parameters if a gird or a block has only one dimension. For example

```
Kernel Function << 128, 32 >>> (…);
```

Source & Acknowledgements : NVIDIA, References

NVIDIA :CUDA – Thread Organization

CUDA – Grid

- ❖ In CUDA, a **grid** is organized as a **2D array or blocks**.
- ❖ Grid Organization is determined by the execution of configuration provided at kernel launch)

  `dim3 dimGrid(128, 1,1);`

  - The **first** parameter - specifies the dimensions of each block in terms of number of blocks
  - The **second** parameter specifies the dimensions of each block in terms of number of threads
    - ➢ Each such parameter is a **`dim3`** type, which is essentially a **`C struct`** with three unsigned integer filed : **`x,y,`**and **`z`**.
  - The **third** parameter –grid dimension parameter is set to 1 for clarity. (Because of grids are 2D array of blocks dimensions)
- ❖ The exact organization of a grid is determined by the execution configuration provided at kernel launch.

NVIDIA :CUDA – Thread Organization

CUDA – Grid ; Host Code to launch the kernel

```
Dim3 dimGrid(128, 1,1);
Dim3 dimBlock(32,1,1,);
Kernel Function <<< dimGrid, dimBlock >>> (...);
```

- ❖ The values of **gridDim.x** and **gridDim.y** can range from **1** to **65535**
- ❖ The values of **gridDim.x** and **gridDim.y** can be calculated based on other variables at kernel launch time.

- ❖ All threads in a block share the same **blockIdx** value.
  - **blockIdx.x** value ranges between **0** and **gridDim.x-1**
  - **blockIdx.y** value ranges between **0** and **gridDim.y-1**
- ❖ Remark : Once a kernel is launched, its dimensions can not change.

NVIDIA :CUDA – Thread Organization

CUDA - Grid- thread blocks

- In CUDA, a **each thread block** is organized into a **3D array of threads**
- All blocks in a grid have the **same** dimensions.

- Each **`threadIdx`** consists of three components : the
  x-coordinate **`threadIdx.x`**,
  y-coordinate **`threadIdx.y`**, and
  z-coordinate **`threadIdx.z`**

- The exact organization *of a thread block is determined by the execution configuration provided at kernel launch.*

NVIDIA :CUDA – Thread Organization

CUDA - Grid- thread blocks

```
dim3 dimBlock(32, 1, 1);
```

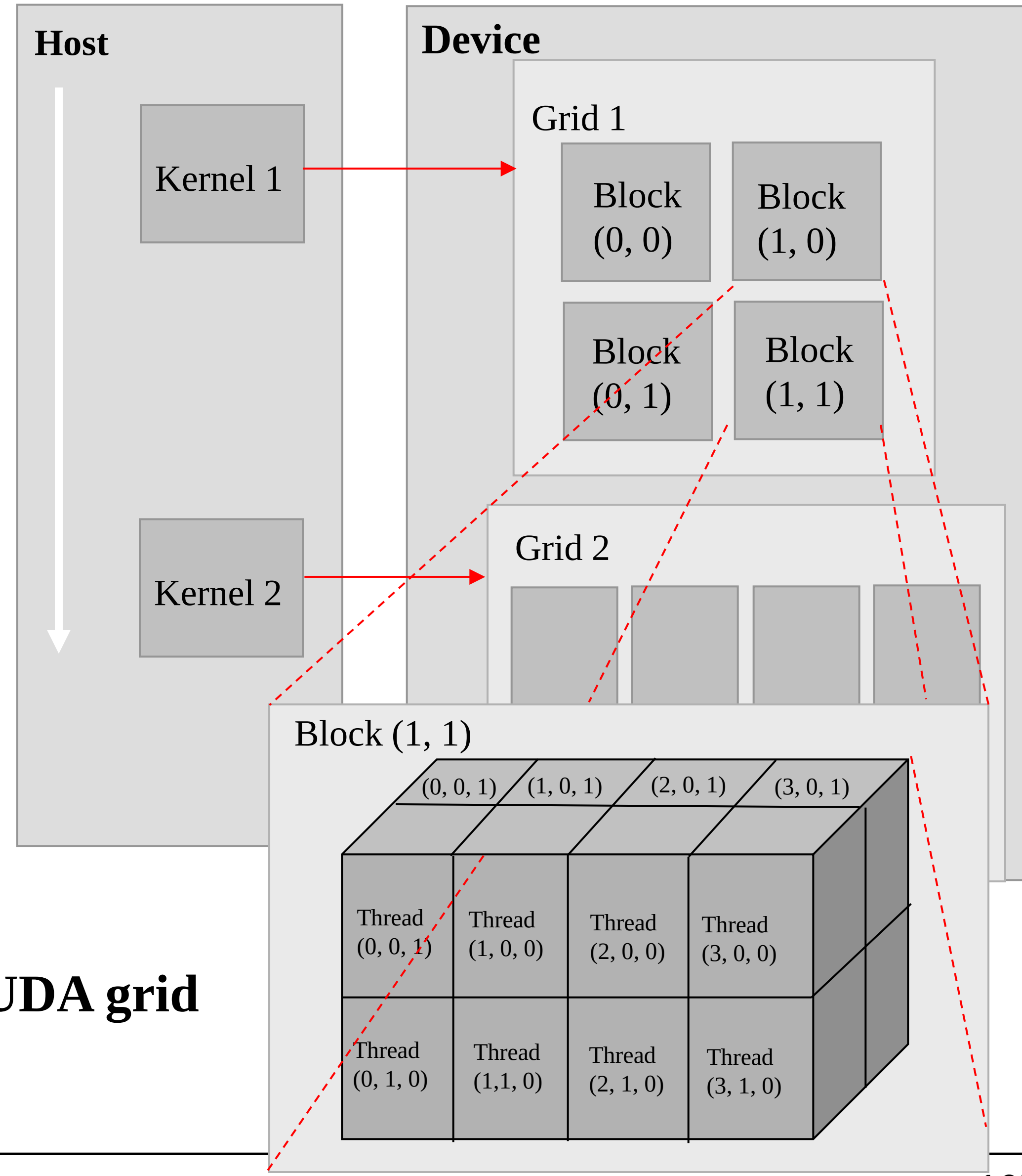
- The **first** parameter - specifies the total terms of number of blocks
- The **second** and **third** parameter specifies the number of threads in each dimension
- The configuration parameter can be accessed as a pre-defined **`C struct`** variable, **`blockDim`**

- Remark : The total size of a block is limited to 512 threads, with flexibility in distribution these elements into the three dimensions as long as the total number of threads does not exceed 512.

Source & Acknowledgements : NVIDIA, References

NVIDIA :CUDA – Thread Organization

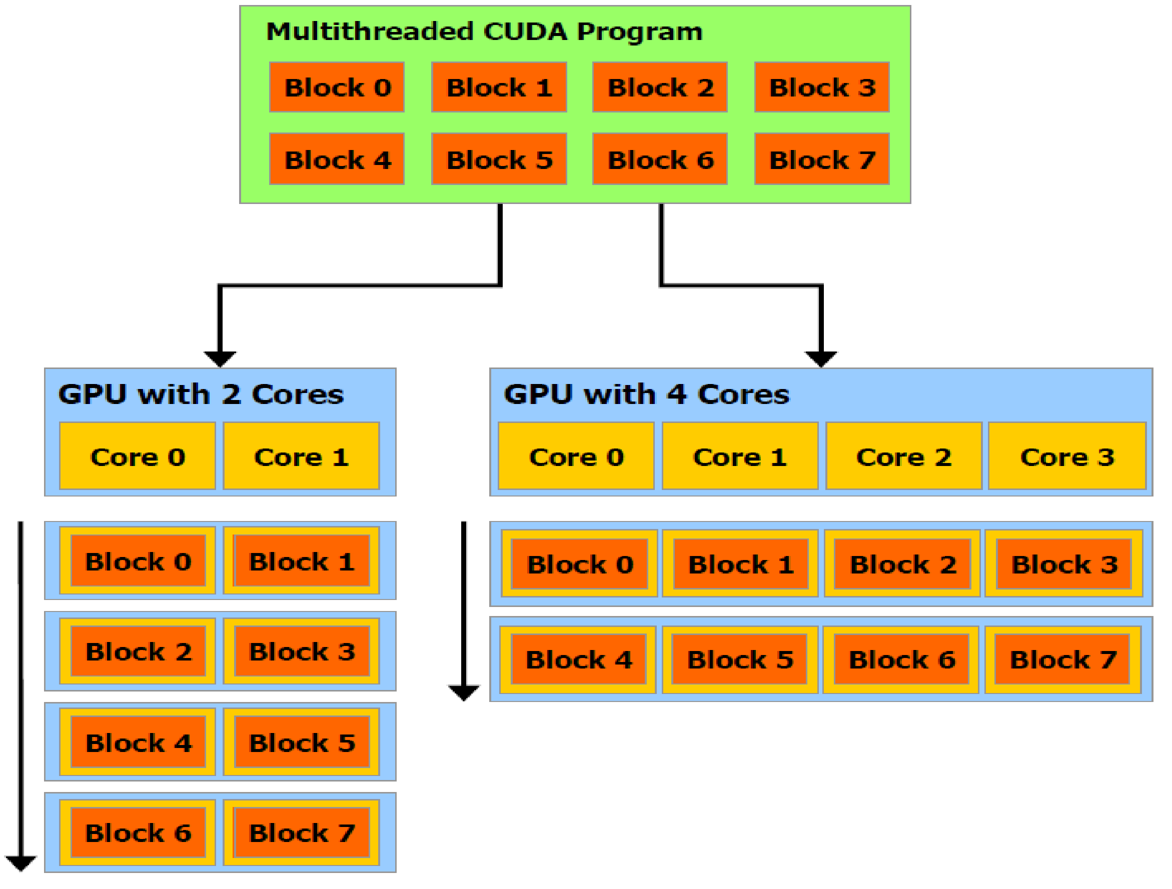
```
Dim3 dimGrid(2, 1,1);
Dim3 dimBlock(4,2,1,);
Kernel Function
 <<<
   dimGrid, dimBlock
    >>>
 (......);
```



A **multidimensional example of CUDA grid organization.**

NVIDIA :CUDA – Thread Organization

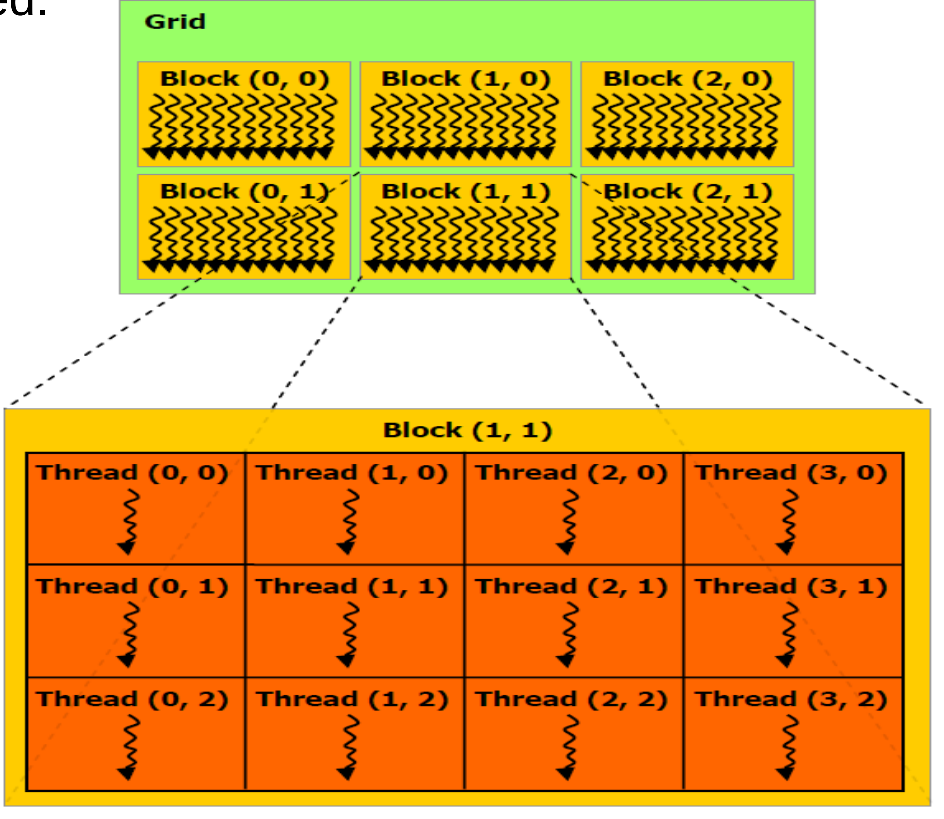
Automatic Scalability : A multi-threaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more cores will automatically execute the program in less time than a GPU with fewer cores.



Source & Acknowledgements : NVIDIA, References

NVIDIA :CUDA – Thread Organization

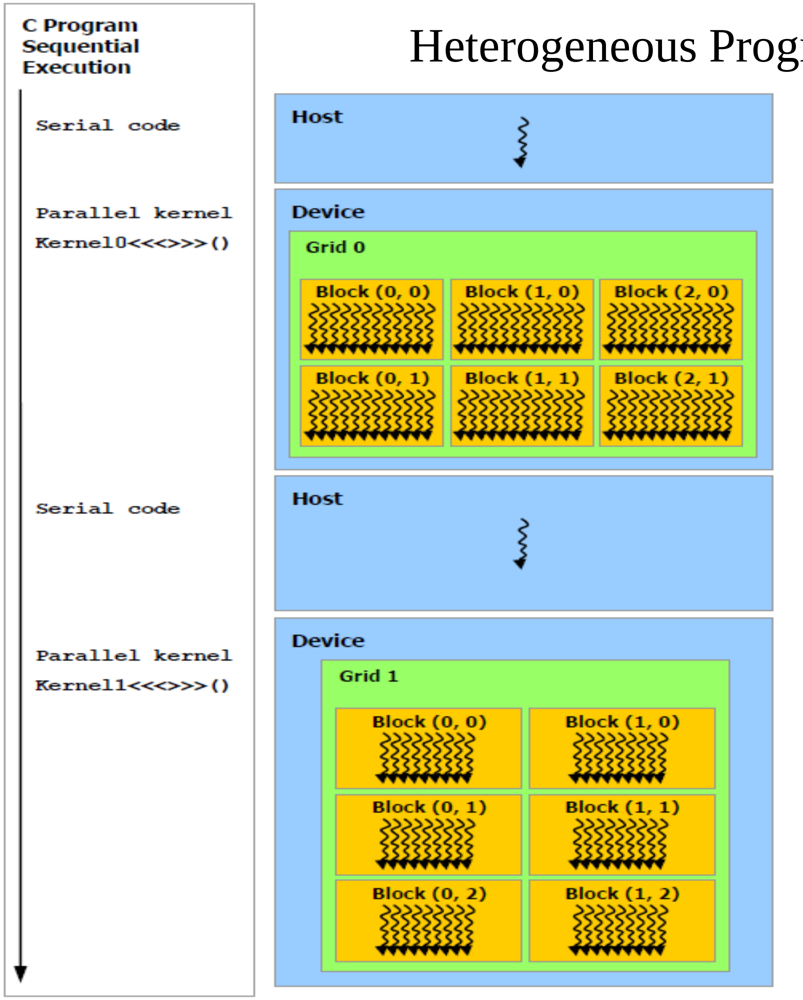
Grid of Thread Blocks : Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks as illustrated by Figure. The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system, which it can greatly exceed.



Source & Acknowledgements : NVIDIA, References

NVIDIA :CUDA – Structure

Heterogeneous Programming



Serial code executes on the host while parallel code executes on the device.

Source & Acknowledgements : NVIDIA, References

Part-II(C)

An Overview of CUDA enabled NVIDIA GPUs:

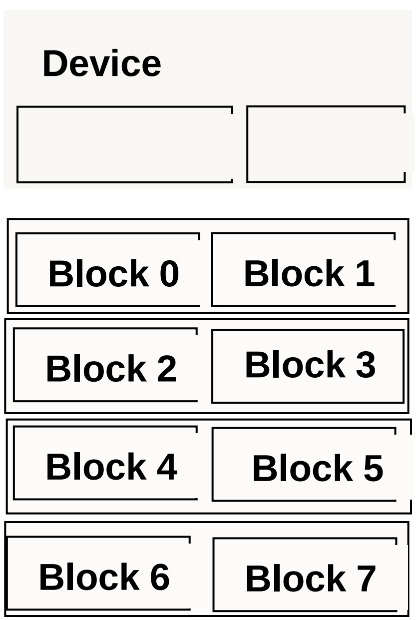
CUDA Synchronization

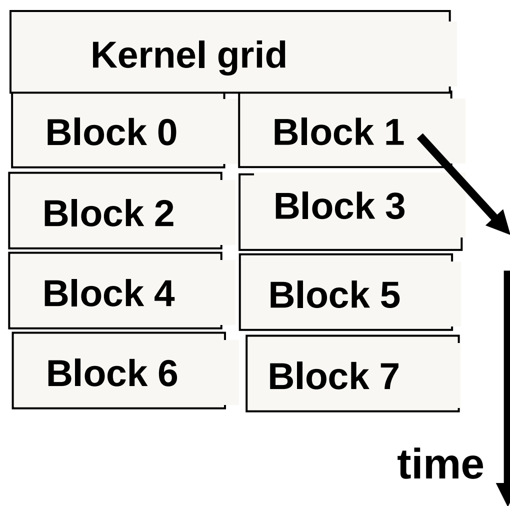
Source & Acknowledgements : NVIDIA, References

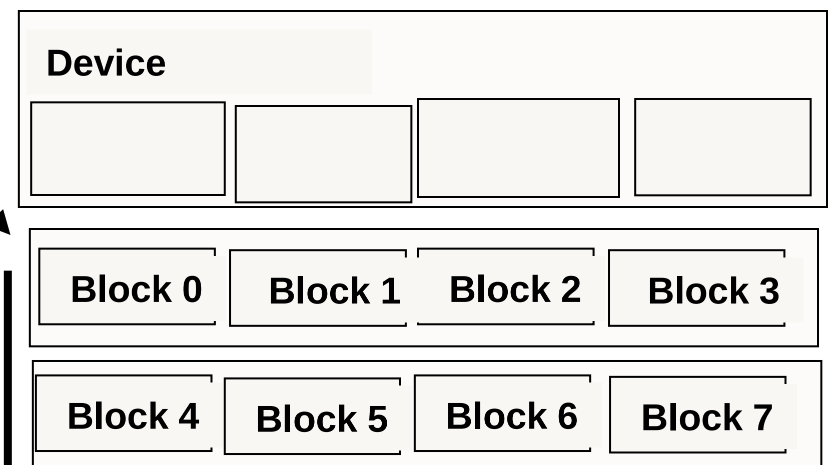
NVIDIA : CUDA Threads Organisation

Synchronization and transparent scalability

- ❖ CUDA allows threads in the same block to coordinate their activities using barrier synchronization function __syncthreads().
- ❖ Call to _synchtreads(), ensures that all threads in a block have completed a phase of their execution of the kernel before any moves on to the next phase.







Each block can execute in any order relative to other blocks

Transparent Scalability for CUDA programs allowed by the lack of synchronization constraints between locks

Synchronization and transparent scalability

- ❖ In CUDA a __syncthreads() statement must be executed by all threads in a block.
- ❖ Call to __syncthreads(), ensures that all threads in a block have completed a phase of their execution of the kernel before any moves on to the next phase.

Issues in CUDA Barrier Synchronization

- ❖ Use of `__synthread()` statement in “**`if`**” statement
- ❖ Use of `__synthread()` statement in “**`if-then-else`**” statement

- ❖ thread may perform execution of “*then*” path OR “*if*” path OR “*else”* path, and this leads to waiting of threads at barrier synchronization points. This results waiting for each other thread.

- ❖ The ability to synchronize also imposes execution constraints on threads within a block.

NVIDIA : CUDA Threads Organisation

Synchronization and transparent scalability

Issues in CUDA Barrier Synchronization : *How to avoid excessive long waiting time ?*

- ❖ The threads in a each block should execute close time proximity with each other.

- ❖ CUDA runtime systems satisfy this constraint by assigning execution resources to all threads in a block as a unit, that is when a thread o a block is assigned to an execution resources.

  - This ensures the time proximity of all threads in a block an prevents excessive waiting time during synchronization

Source & Acknowledgements : NVIDIA, References

NVIDIA : CUDA Threads Organisation

Synchronization and transparent scalability

Issues in CUDA Barrier Synchronization : *How to avoid excessive long waiting time ?*

❖ CUDA runtime can execute blocks in any order relative to each other because none of them must wait for each other.

❖ **Remark :** The ability to execute the same application ode at a wide range of speeds allows the production of a wide range of implementation according to the cost, power, and performance requirements of particular market segment.

❖ In **CUDA** one can execute large number of blocks at the same time, subject to more resources exist for typical high-end implementation

Source & Acknowledgements : NVIDIA, References

NVIDIA : CUDA Threads Organisation

Thread Assignment :

❖ Once the kernel is launched, CUDA runtime system generates the corresponding grid of threads.

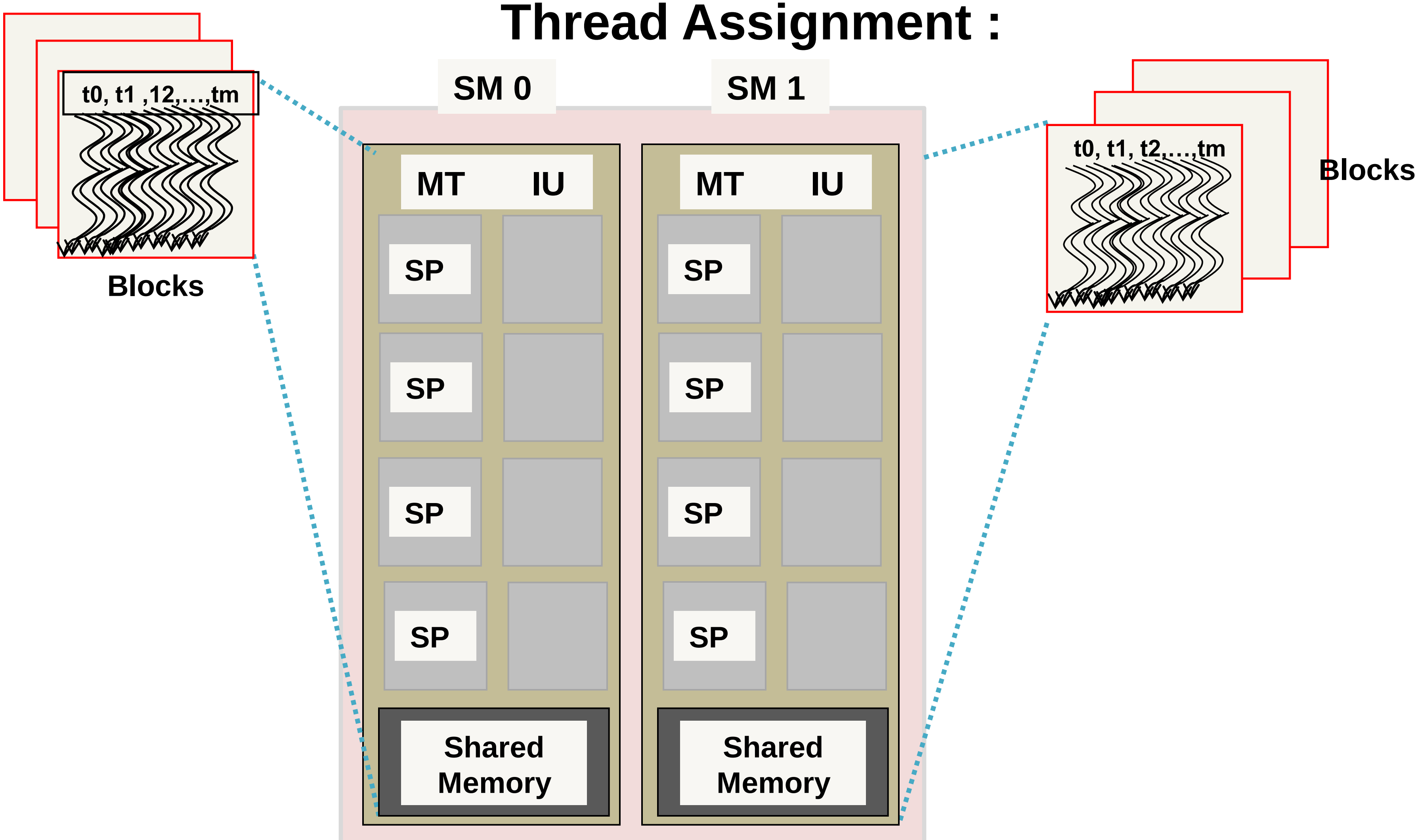
❖ These threads are assigned to execution resources on a block-by-block basis.

❖ Thread block assignment to streaming multiprocessors (SMs)

Source & Acknowledgements : NVIDIA, References

NVIDIA : CUDA Threads Organisation

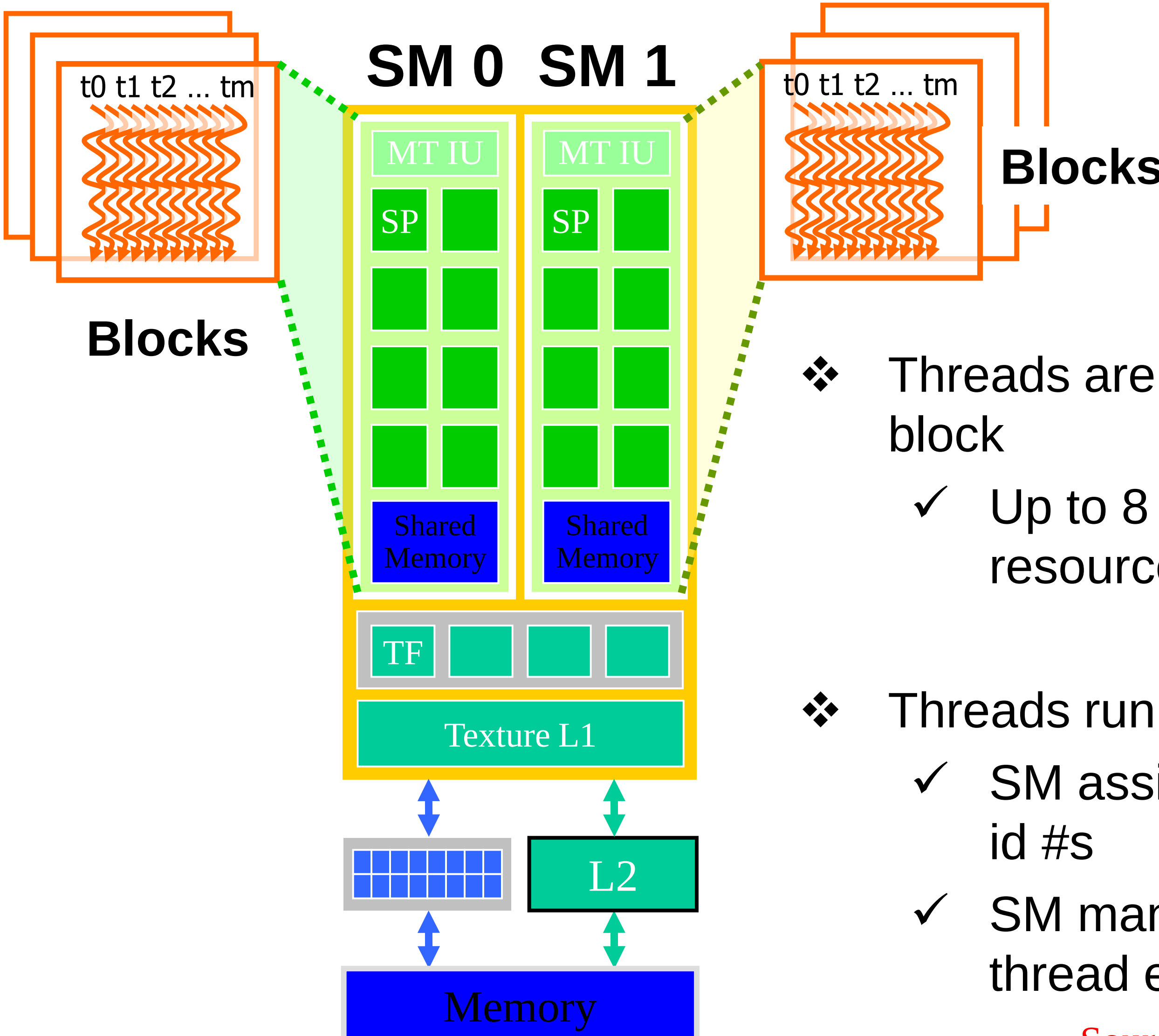
Thread Assignment :



Thread block assignment to streaming multiprocessors (SMs)

Source & Acknowledgements : NVIDIA, References

NVIDIA : CUDA Threads Organisation

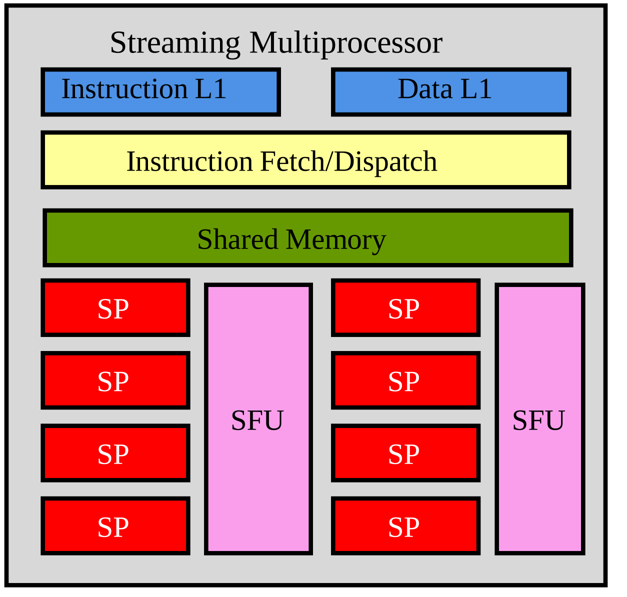


- ❖ Threads are assigned to SMs in block
  - ✓ Up to 8 Blocks to each SM as resource allows
- ❖ Threads run concurrently
  - ✓ SM assigns/maintains thread id #s
  - ✓ SM manages/schedules thread execution

Source : NVIDIA, References

Streaming Multiprocessor (SM)

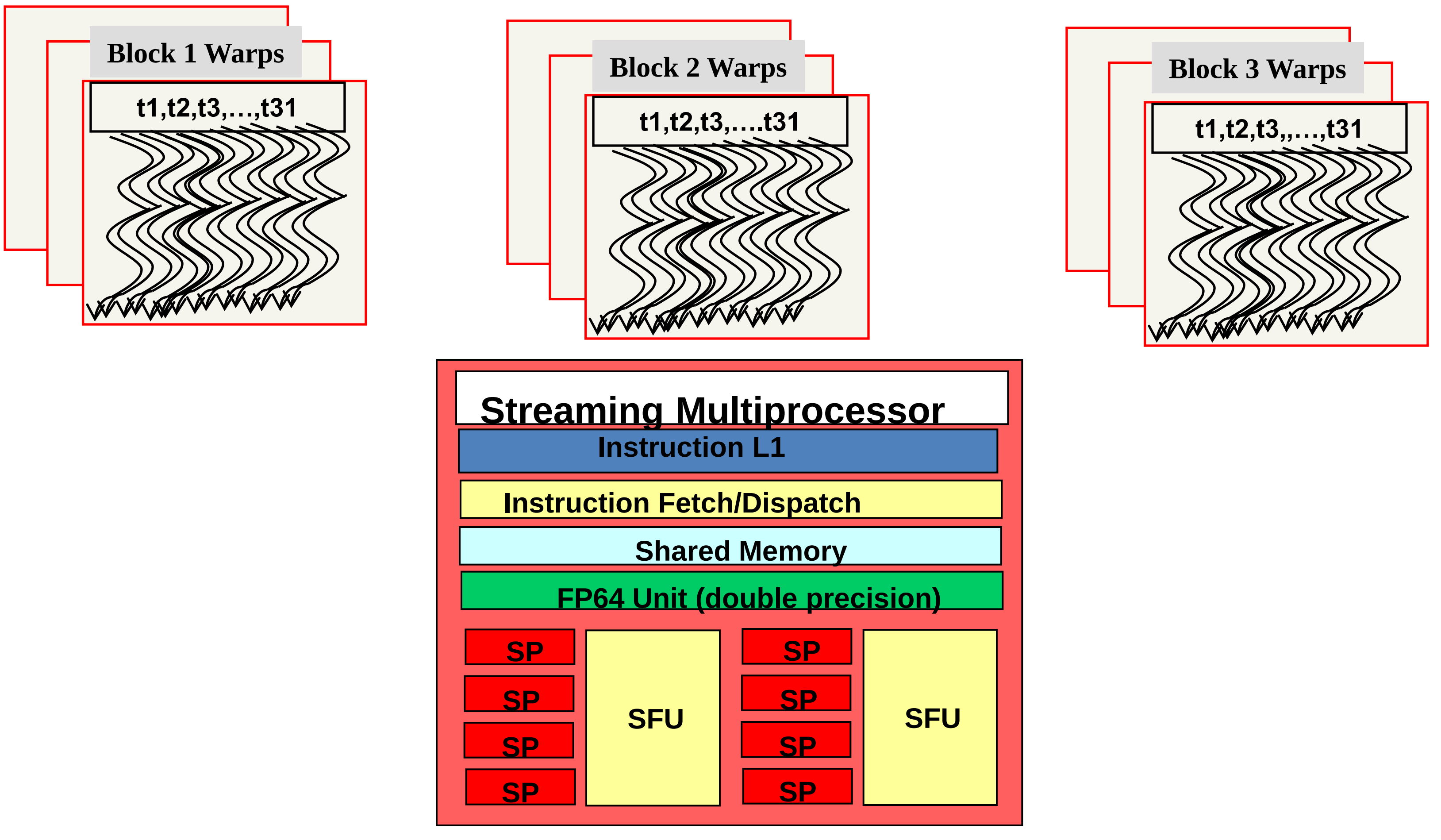
- Streaming Multiprocessor (SM)
  - ✓ 8 Streaming Processors (SP)
  - ✓ 2 Super Function Units (SFU)
- Multi-threaded instruction dispatch
  - ✓ 1 to 512 threads active
  - ✓ Shared instruction fetch per 32 threads
  - ✓ Cover latency of texture/memory loads
- 20+ GFLOPS (24 GFLOPS in G92)
- 16 KB shared memory
- DRAM texture and memory access



Source : NVIDIA, References

NVIDIA : CUDA Thread Scheduling & Latency Tolerance

NVIDIA GT200 GPU Block Diagram GT200 : Tesla C1060/ S1070
Blocks partitioned into *warp* for thread scheduling



Source & Acknowledgements : NVIDIA, References

NVIDIA : CUDA Threads Organisation

Thread Assignment

- ❖ Execution resources are organized into streaming multiprocessors **NVIDIA GT200** implementation features

  - **30** Streaming Multi-Processors (**SMs**)
  - **8** Threading blocks can be assigned to each **SM** as long as there are enough execution resources to satisfy the needs of all the blocks.
  - Each threading block can have atmost **512** threads
  - **240** thread blocks can be simultaneously assigned to **SM**s
  - **Upto 1024** threads can be assigned to each **SM**
  - Maximum of **30720** threads can be simultaneously residing in the **SM**

- ❖ Most grids contain many more than **240** blocks.
- ❖ The runtime system maintains a list of blocks that need to execute and assign new blocks to SMs as they complete execution of blocks previously assigned to them.
- ❖ **Note :** In situations with an insufficient amount if any one or more types of resources needed for the simultaneous execution of 8 blocks , the CUDA runtime automatically reduces the number of blocks assigned to each SM until the resource usage is under the limit.

NVIDIA : CUDA Threads Organisation

Thread Assignment

- ❖ Three thread blocks assigned to each SM.
- ❖ One of the SM resource limitations is the number of threads that can be simultaneously tracked and scheduled.
- ❖ Hardware resources are required for SMs to maintain the thread, block IDs, and track their execution status.
- ❖ **Upto 1024** threads can be assigned to each SM.
  - 4 blocks of 256 threads each, 8 blocks of 128 threads each .. *(16 blocks of 64 threads each is not possible.)*

❖ Execution resources are organized into streaming multiprocessors

NVIDIA GT80 implementation features

- **16** Streaming Multi-Processors (**SMs**)
- **8** Threading blocks can be assigned to each **SM** as long as there are enough execution resources to satisfy the needs of all the blocks.
- Each threading block can have atmost **256** threads
- **Upto 768** threads can be assigned to each **SM** (3 blocks of 256 each; 6 blocks of 128 threads each)
- Maximum of **12288** threads can be simultaneously residing in the **SM**

Source & Acknowledgements : NVIDIA, References

NVIDIA :CUDA – Thread Organization

CUDA - Grid- thread blocks

Ex : A multi-dimensional example of CUDA grid organization

❖ The grid consists of four blocks organized into a 2 X 2 array

- Each block is in figure is labeled with (**`blockIdx.x`**, **`blockIdx.y`**)
- Ex : Block (1,0) has **`blockIdx.x`** = 1, and **`blockIdx.y`** = 0

❖ In CUDA, total size of block is limited to **512** threads, with flexibility in distributing these elements into the three dimensions as long as the total number of threads does not exceed 512 threads. (****)

❖ **Ex :** (512,1,1,), (8,16,2) and (16,16,2) are allowable blockDim values, but (32,32,1) is not allowable because the total number of threads would be 1024.

NVIDIA :CUDA – Thread Organization

CUDA - Grid- thread blocks

Ex : A multi-dimensional example of CUDA grid organization

- ❖ Grid consists of 4 blocks of 16 threads each, with a grand total of 64 threads in the grid.

- ❖ Each thread block is organized into 4 X 2 X 2 arrays of threads (16 threads). (Only **one** block is shown because of all thread blocks in the grid have **same** dimension.)

- ❖ block (1,10) to show its 16 threads;
  - ➢ thread (2,1,0) has

    `blockIdx.x` = 2, **`blockIdx.y`** = 1, **`blockIdx.z`** = 0
- ❖ CUDA grid contain thousands to million of threads

Source & Acknowledgements : NVIDIA, References

NVIDIA :CUDA – Thread Organization

KERNEL FUNCTIONS AND THREADING

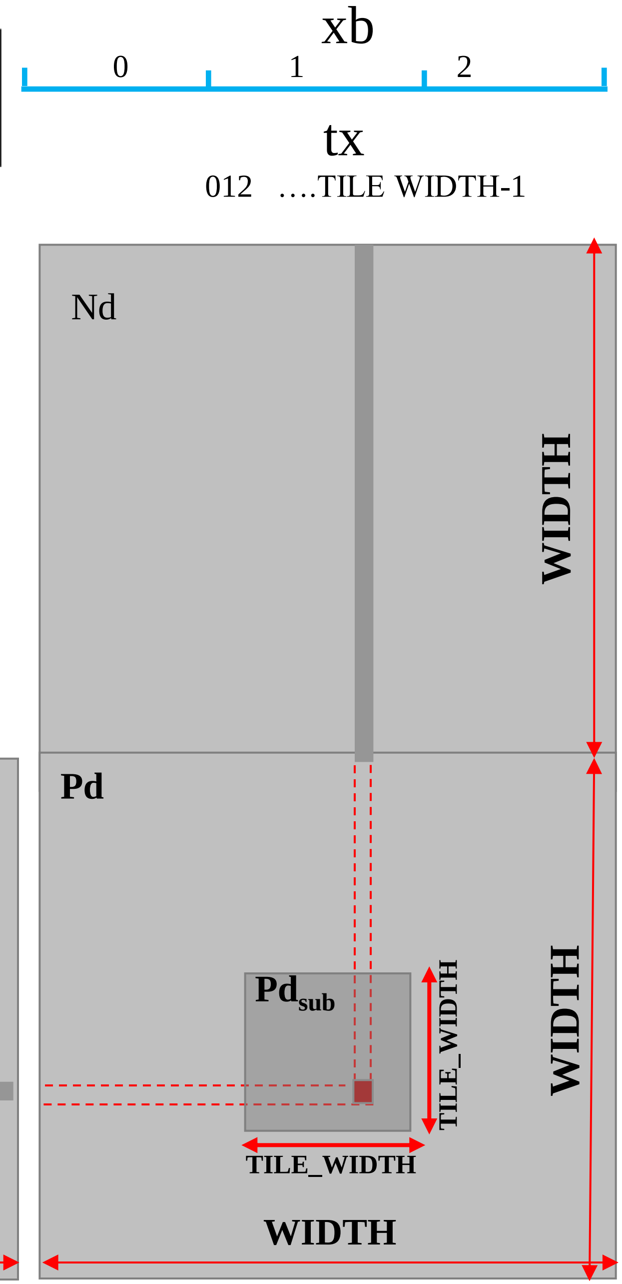
❖ **`threadIdx.x & threadIdx.y`**

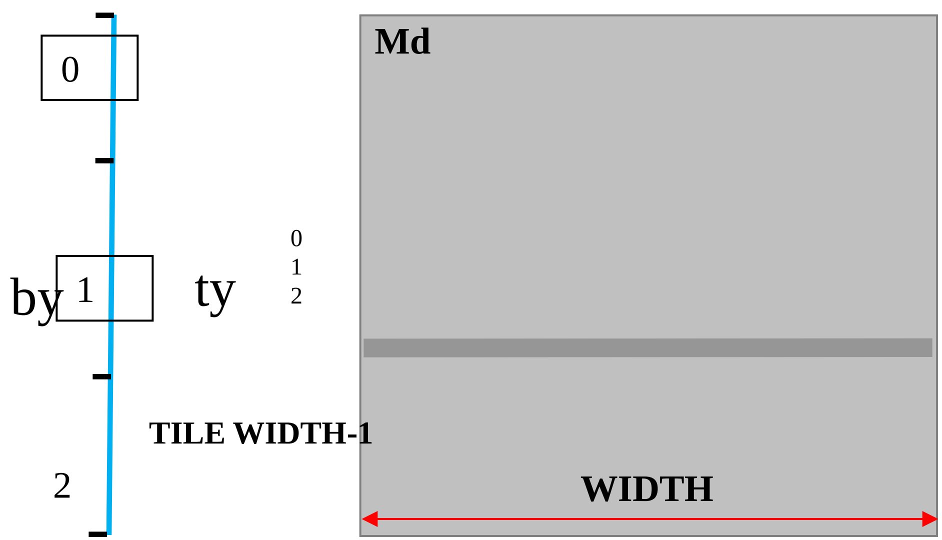
- Refer to the thread indices of a thread (Different threads will see different values in their **`threadIdx.x`** and t**`hreadIdx.y`** variables)
- Refer thread as **`Thread`***threadIdx.x, threadIdx,y* Coordinates reflect a multi-dimensional organization for the threads.
- CUDA threading hardware generates all of the **`threadIdx.x`** and t**`hreadIdx.y`** variables for each thread.
- These work on particular part of data structure of the designed code and with these thread indices allow a thread to access the hardware registers at runtime that provides the identifying coordinates to the thread.

Source : NVIDIA

NVIDIA :CUDA – Thread Organization

- **USING blockIdx AND threadIdx**
  - **Break Pd into square tiles**
  - All the **Pd** element s of a **tile** are computed by a block of threads
    - Keep dimensions of these **Pd tiles** small, we can increase the total number of threads in each block to 512 which is maximum allowable block size.





Matrix Multiplication using multiple blocks by tiling **Pd**

NVIDIA :CUDA – Thread Organization

USING blockIdx AND threadIdx

❖ For convenience sake ,

`threadIdx.x` and **`threadIdx.y`** as **`tx and ty`**; and **`blockIdx.x`** and **`blockIdx.y`** as **`bx`** and **`by`**.

- Each thread calculates one **`Pd`** element. The difference is that it must uses its **`blockIdx.x`** values to identify its element inside the tile.
- Each thread uses both **`threadIdx`** and **`blockIdx`** to identify the **`Pd`** element to work on.
- All threads calculating the **`Pd`** elements within a **`tile`** have the same **`blockIdx`** values

Source : NVIDIA

NVIDIA :CUDA – Thread Organization

USING blockIdx AND threadIdx

❖ Assume that the dimensions of a block are square and are specified by the variable **`TILE_WIDTH`**

❖ Each dimensions of **`Pd`** is now divided into section s of **`TILE_WIDTH`** elements each and each block handles such a section.

- Thread can find **`x`** index and **`y`** index of **`Pd`** element i.e.

  `x = bx + TILE_WIDTH + tx`

  `y = by + TILE_WIDTH + ty`

`Pd` element at respective column & row can be computed.

Source : NVIDIA

NVIDIA :CUDA – Thread Organization

USING blockIdx AND threadIdx

❖ Assume that the dimensions of a block are square and are specified by the variable **`TILE_WIDTH`**

❖ Each dimensions of **`Pd`** is now divided into section s of **`TILE_WIDTH`** elements each and each block handles such a section.

- Thread can find **`x`** index and **`y`** index of **`Pd`** element i.e.

  `x = bx + TILE_WIDTH + tx`

  `y = by + TILE_WIDTH + ty`

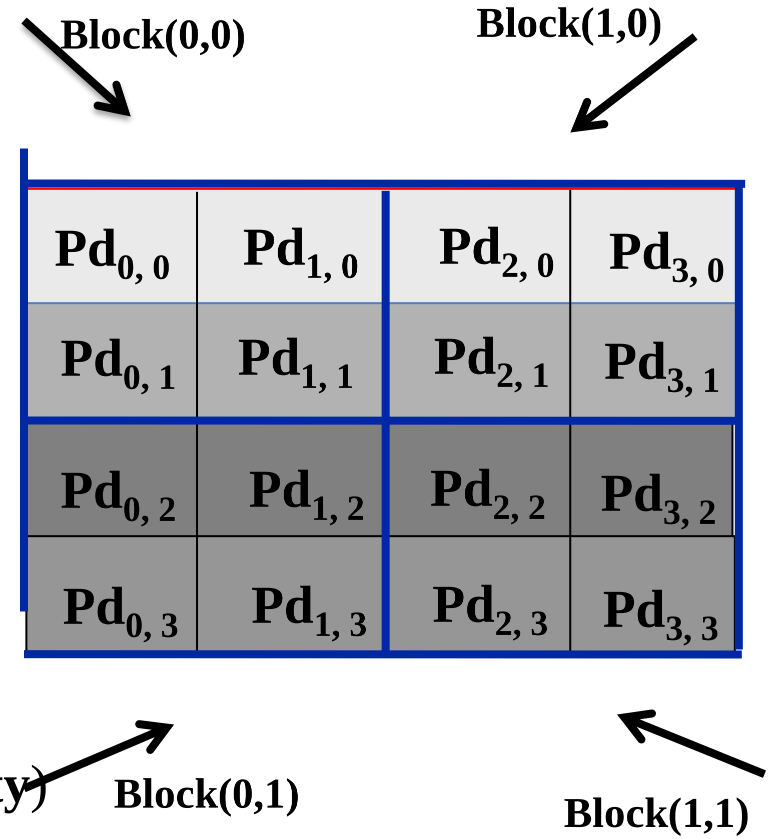
`Pd` element at respective column & row can be computed.

Source & Acknowledgements : NVIDIA, References

NVIDIA :CUDA THREAD ORGANIZATION

USING blockIdx AND threadIdx Ex : Matrix Multiplication

- **Using Multiple blocks to calculate Pd.**
  - **Break Pd into 4 tiles**
  - Each *dimension of* ***Pd*** is now divided into sections of **2** elements
  - Each block needs to calculate **4 Pd** elements
- **Identify the indices for the Pd element**
  Thread (0,0) of block (0,0) calculates $Pd_{0,0}$ whereas thread (0,0) of block (1,0) calculates $Pd_{2,0}$
- Identify the row (**y**) of **Md** and the column (**x**) of index of **Nd** for input values using **TILE_WIDTH**



Using Multiple blocks to calculate **Pd**.

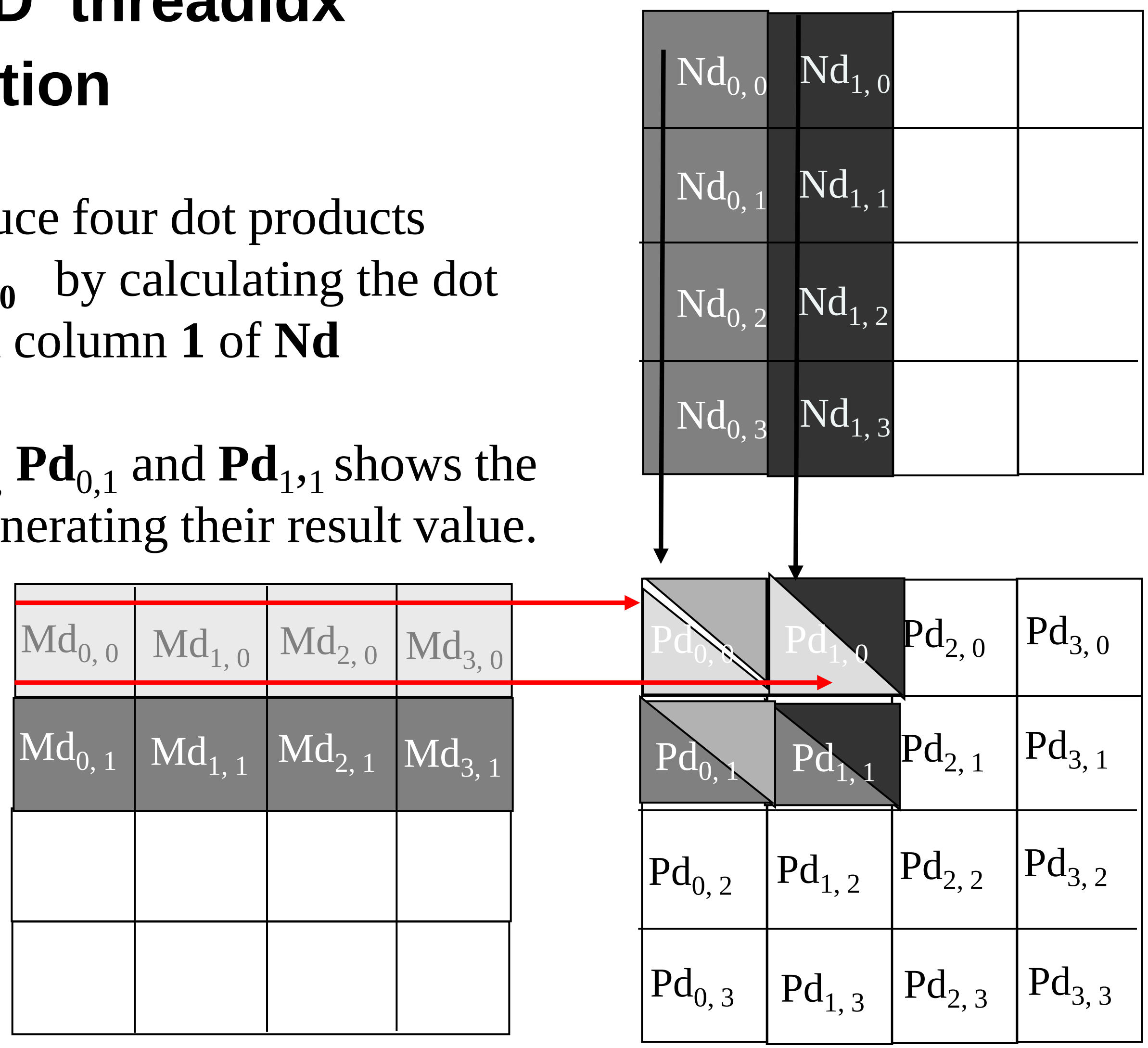
- For the row index of **Md** used by thread (**tx,ty**) of block (**bx,by**) is **(by*TILE_WIDTH + ty)**
- For the clumn index of **Nd** used by the same is **(bx*TILE_WIDTH + tx)**

USING blockIdx AND threadIdx

Ex : Matrix Multiplication

- Threads in block **(0,0)** produce four dot products
- Thread (**0,0**) generates **Pd$_{0,0}$** by calculating the dot product of row **0** of **Md** and column **1** of **Nd**

- The arrows of **Pd**$_{0,0}$, **Pd**$_{1,0,}$ **Pd**$_{0,1}$ and **Pd**$_{1,1}$ shows the row and column used for generating their result value.



Matrix multiplication actions of one thread block

Ex : Matrix Matrix Addition

```
// Kernel definition
_global_ void MatAdd(float A[N][N], float B[N][N],
                     float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x
    int j = blockIdx.y * blockDim.y + threadIdx.y
    if (i < N && j < N)
        c[i][j] = A[i][j] + B[i][j];
}
int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N/ threadPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    …
}
```

Ex : Matrix Matrix Addition

```
// Kernel definition
_global_ void MatAdd(float A[N][N], float B[N][N],
                     float C[N][N])
{
   int i = threadIdx.x;
   int j = threadIdx.y;
   c[i][j] = A[i][j] + B[i][j];
}
int main()
{
   ...
   // Kernel invocation with one block of N * N * 1 threads
   int numBlocks = 1;
   dim3 threadsPerBlock(N, N);
   MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
   ...
}
```

Thread Hierarchy

Revised matrix multiplication kernel using multiple blocks

```
__global__ void MatrixMulKernel(float* Md, float* Nd,float* Pd,
int Width)
{
   // Calculate the row index of the Pd element and M
   int Row  = blockIdx.y *TILE_WIDTH + threadIdx.y;

   // Calculate the column index of the Pd element and N
   int Col  = blockIdx.x *TILE_WIDTH + threadIdx.x;

   float Pvalue = 0;
   // each thread computes one element of the block sub-matrix
   for(int k = 0; k < Width; ++k)
      Pvalue +-  Md[Row*Width+k] * Nd[k*Width+Col)];

      Pd[Row*Width_col] = Pvalue;
}
```

NVIDIA :CUDA – Thread Organization

Summary of matrix multiplication kernel using multiple-blocks:

❖ **Step 1 :** Each thread uses its **`blockIdx`** and **`threadIdx`** values to identity the row index (**`Row`**) and the column index (**`Col`**) of the **`Pd`** element that is responsible for.

❖ **Step 2 :** Performs a dot product on the row of **`Md`** and column of **`Nd`** to generate the value of the **`Pd`** element. It eventually writes the **`Pd`** value to the appropriate global memory locations.

Note : This kernel can handle matrices upto 16 X 65,535 elements in each dimension.

❖ For large matrices, one can divide the **`Pd`** matrix into sub-matrices of a size permitted by the kernel

Source : NVIDIA

NVIDIA :CUDA – Thread Organization

Summary of matrix multiplication kernel using multiple-blocks:

- For large matrices, one can divide the Pd matrix into sub-matrices of a size permitted by the kernel
- Each submatrix can be processed by an ample number of blocks (65,535 X 65,535). All of these blocks can run in parallel provided new design of GPUs which can accommodate large number of execution resources.

Source & Acknowledgements : NVIDIA, References

NVIDIA : CUDA Thread Scheduling & Latency Tolerance

Thread Scheduling : In **CUDA** it is an specific hardware implementation

- **Case Study :**

- **G200 :** Number of warps per **SM** may increased up to 32.

- The ***warp*** scheduling is used for long-latency hiding (long latency operations) refers to access of global memory access

- Zero-overhead thread scheduling takes place in CUDA, in which selection of ready warps for execution does not introduce any idle time into the execution timeline.

Revised matrix multiplication kernel using multiple blocks

Revised Host code for launching the revised kernel

```
// Setup the execution configuration
   dim3 dimGrid(Width/TILE_WIDTH, Width/TILE_WIDTH);
   dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);

 // Launch the device computation threads;
   MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

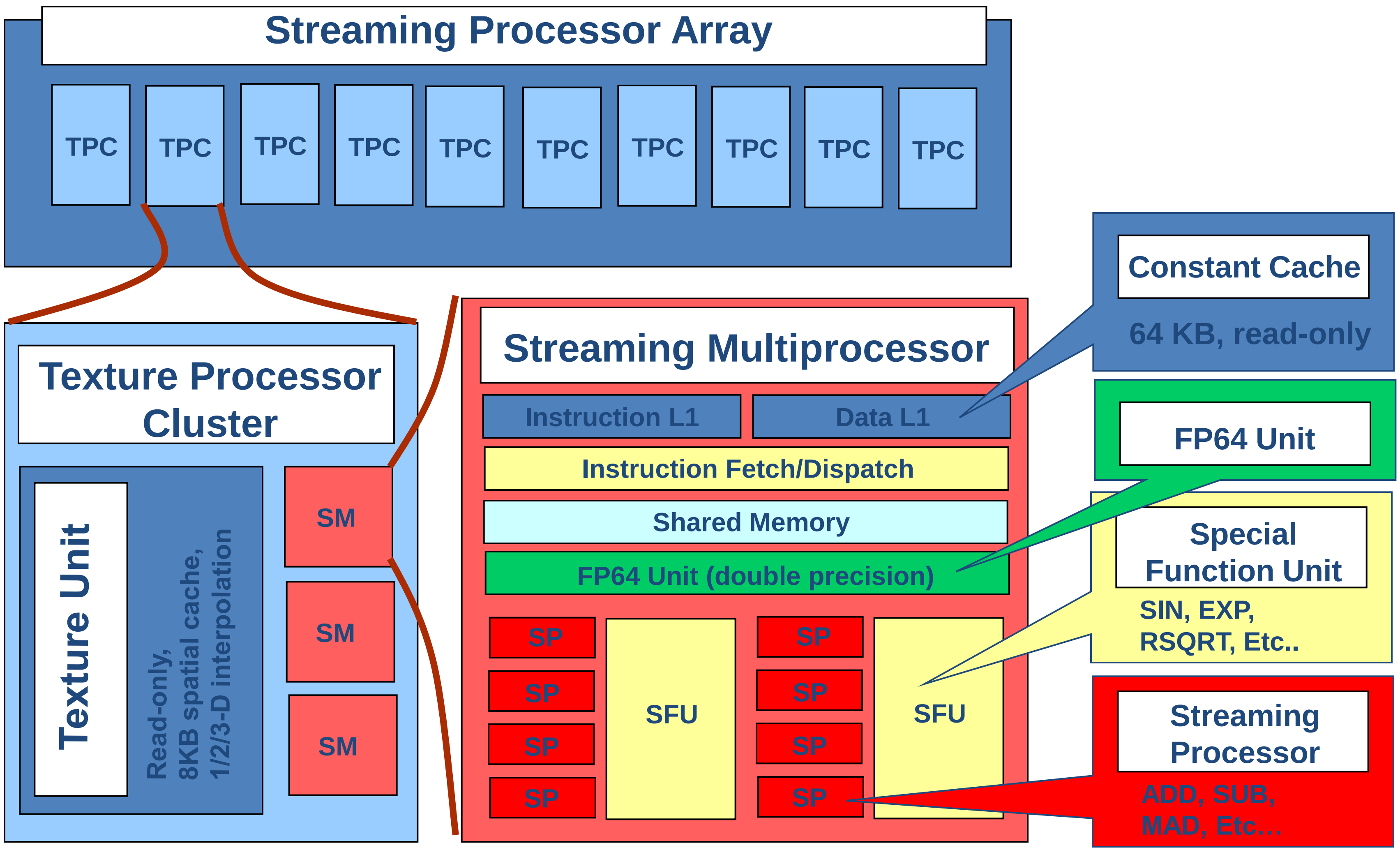
`Note : dimGrid` receives the value of **`Width/TILE_WIDTH`** for both the **`x`** dimension and **`y`** dimension.

`Md`, **`Nd`**, and **`Pd`** array as 1D array with row major layout

The calculation of indices used to access **`Md`**, **`Nd`** and **`Pd`** is the same

NVIDIA – GPU Computing Products - History

NVIDIA GT200 GPU Block Diagram GT200 :
incorporated in Tesla C1060 & S1070 products.



Source : NVIDIA, References

NVIDIA : CUDA Thread Scheduling & Latency Tolerance

Thread Scheduling : In **CUDA** it is an specific hardware implementation

- ❖ Once a thread block is assigned to each SM, it is further divided into 32-thread units called ***warps.***
  (Knowledge of ***warps*** can be helpful in understanding and optimizing the performance of **CUDA** applications on particular generations of CUDA devices.

- ❖ The ***warp*** is the unit of thread scheduling in SMs

- ❖ Each ***warp*** consists of 32 threads of consecutive `threadIx` values
  - Threads 0 through 31 from the first warp, threads 32 through 63 second warp, and so on…..
    Ex : Three blocks (Block 1, Block2, & Block 3) are assigned to an SM and each block is further divided into warps for scheduling.
  - If each block has 256 threads, then we can determine that each block has 256/32 or 8 warps.
  - With 4 blocks in each SM, we have 8 x 3 = 24 warps in each SM

Thread Scheduling : In CUDA it is an specific hardware implementation

- **G80 :** In each **SM** maximum number of threads is 768, equivalent to 24 ***warps.***
- **G200 :** Number of warps per **SM** may increased up to 32.
- The ***warp*** scheduling is used for long-latency hiding (long latency operations) refers to access of global memory access
- Zero-overhead thread scheduling takes place in CUDA, in which selection of ready warps for execution does not introduce any idle time into the execution timeline.

Source & Acknowledgements : NVIDIA, References

NVIDIA : CUDA Thread Scheduling & Latency Tolerance

Thread Scheduling : In **CUDA** it is an specific hardware implementation

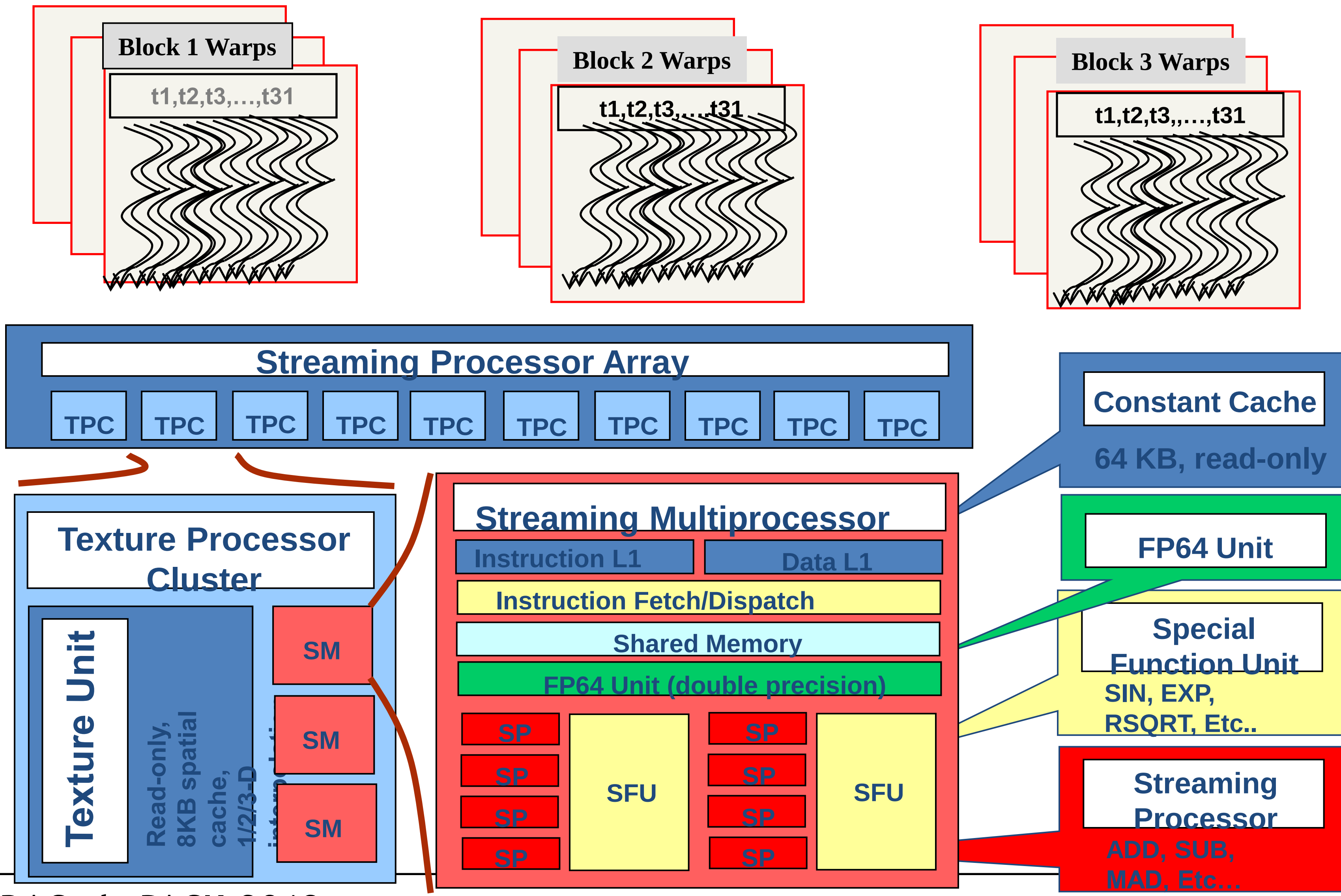
Matrix – Matrix Multiplication; G200 : Number of warps per **SM** is 32 and the number of threads that can be assigned to each SM is **1024 &** the number of threads assigned to each thread block is **512**

Pros & Cons of choice of "different thread blocks" for the GT200

- **Case Study -1 : 8 X 8 thread blocks :** Each block has 64 threads, & 12 (1024/64) blocks fully occupy an SM (8 blocks in each SM are limited and hence 64x 8 = 512 threads in each SM is possible.
  - This shows SM execution resources will likely to be under utilized as there will be fewer **warps**
- **Case Study -2 : 16 X 16 thread blocks :** Each block has 256 threads, & 4 (1024/256) blocks fully occupy an SM (8 blocks in each SM are limited and it s well within the limits. Good choice for performance.
- **Case Study -3 : 32 X 32 thread blocks :** Each block has 1024 thread which exceeds the limitation of up to 512 threads per block

NVIDIA : CUDA Thread Scheduling & Latency Tolerance

NVIDIA GT200 GPU Block Diagram GT200 : Tesla C1060/ S1070
Blocks partitioned into for thread scheduling



Source : NVIDIA, References

NVIDIA : CUDA Thread Scheduling & Latency Tolerance

Thread Scheduling : In **CUDA** it is an specific hardware implementation

Matrix – Matrix Multiplication; G200 : Number of warps per **SM** is 32 and the number of threads that can be assigned to each SM is **1024 &** the number of threads assigned to each thread block is **512**

Pros & Cons of choice of "different thread blocks" for the GT200

- ❖ **Case Study -1 : 8 X 8 thread blocks :** Each block has 64 threads, & 12 (1024/64) blocks fully occupy an SM (8 blocks in each SM are limited and hence 64x 8 = 512 threads in each SM is possible.
  - This shows SM execution resources will likely to be under utilized as there will be fewer **warps**
- ❖ **Case Study -2 : 16 X 16 thread blocks :** Each block has 256 threads, & 4 (1024/256) blocks fully occupy an SM (8 blocks in each SM are limited and it s well within the limits. Good choice for performance.
- ❖ **Case Study -3 : 32 X 32 thread blocks :** Each block has 1024 thread which exceeds the limitation of up to 512 threads per block

Part-II(D)

An Overview of CUDA enabled NVIDIA GPUs:

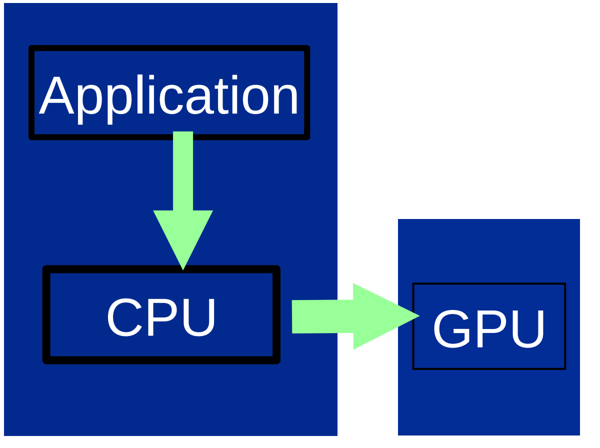
CUDA Memories

Source & Acknowledgements : NVIDIA, References

GPU Computing : Think in Parallel

GPU Computing : Take Advantage of Shared Memory

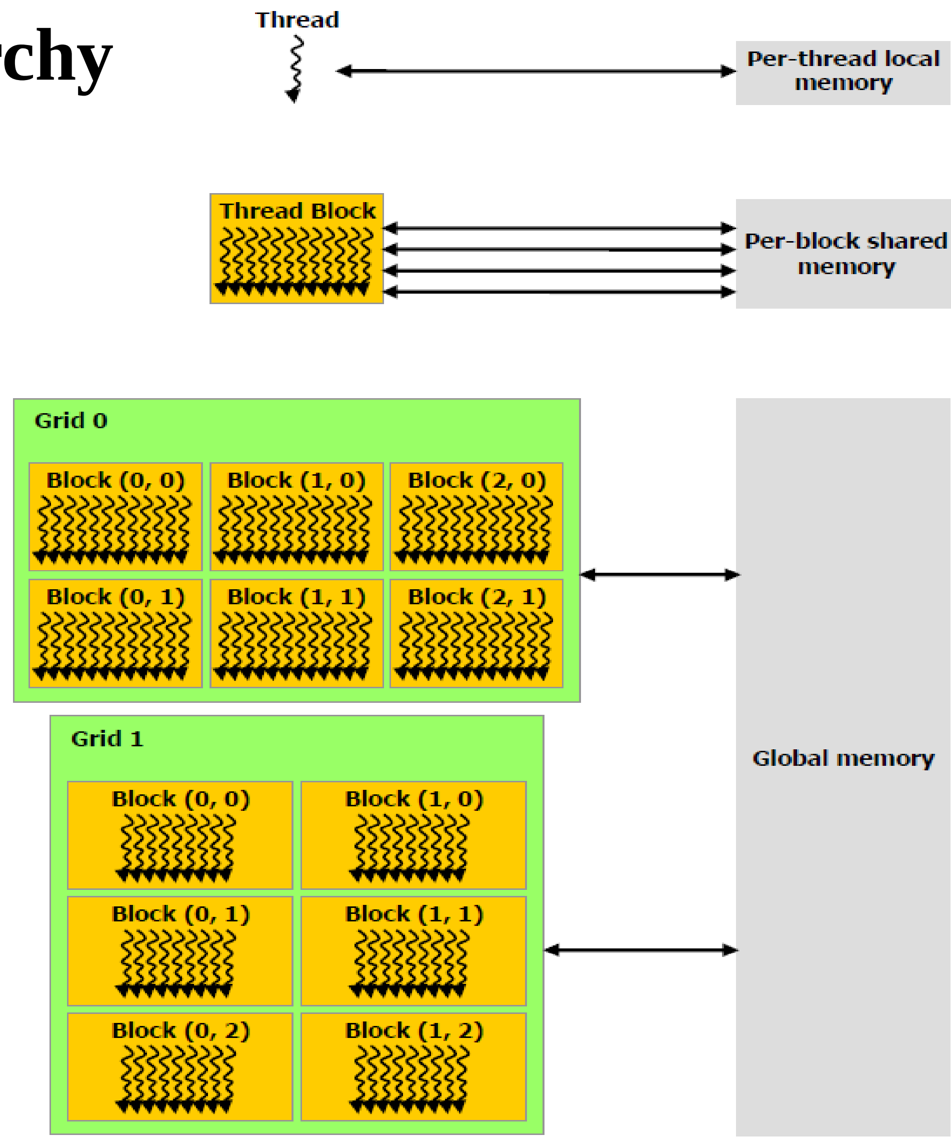
- ❖ Hundreds of times faster than global memory
- ❖ Threads can cooperate via shared memory
- ❖ Use one/ a few threads to load/computer data shared by all threads
- ❖ Use it to avoid non-coalesced access
  - Stage loads and stores in shared memory to re-order non-coalesceable addressing
  - Matrix transpose example later



Source & Acknowledgements : NVIDIA, References

NVIDIA :CUDA – Memory Hierarchy

Memory Hierarchy



NVIDA :CUDA - Quick terminology review

- ❖ *CUDA* exposes the memory hierarchy to developers, allowing them to maximize application performance by optimizing data access

- ❖ The *GPU* is implemented on a graphics card with video memory, called ***device memory***

  - The video memory (*off-chip*) memory is separated from the GPU, and it takes at least 400 clock-cycles to fetch data from that memory.

  - Two groups of memory on a graphics card.

    - ➢ On-chip (shared) memory is almost fast as **registers**.

    - ➢ Off-chip (device) memory takes **400-600 clock** cycles /store data.

Source : NVIDIA

CUDA : Importance of Memory Access Efficiency

Ex : Matrix – Matrix Multiplication : **Memory access calculation for matrix-matrix commutations – “for**” loop based on **CGMA**

- ❖ **Compute to Global Memory Access (CGMA) ratio :** Number of floating point calculations performed for each access to the global memory within a region of a CUDA program
  - The ratio of floating-point calculation to the global memory access operations is **1** to **1.** or **1.0**
- ❖ **The CGMA ratio** has major implications on the performance of a CUDA kernel.
  - **Ex : NVIDIA G*80** supports **86.4** gigabytes per second (GB/s) of global memory access bandwidth.
  - The highest achievable floating-point calculation throughput is limited by the rate at which the input data can be loaded from the global memory.

Source & Acknowledgements : NVIDIA, References

CUDA : Importance of Memory Access Efficiency

Ex : Matrix – Matrix Multiplication : ** Memory access calculation for matrix-matrix computations – "for**" loop based on **CGMA**

- With **4 bytes** in each single precision floating-point datum**,** one can expect to load not more than 21.6 (86.4/4) giga single-precision data per second.

- With a **CGMA** ration of 1.0, the matrix multiplication kernel will execute at no more than 21.6 billion floating point operations per second (gigaflops), as each floating operation requires one single-precision global memory datum.

- The achieved is fraction of the peak performance of 367 gigaflops for the G80

How CGMA ratio is increased to achieve a higher level of performance for the kernel ?

Source & Acknowledgements : NVIDIA, References

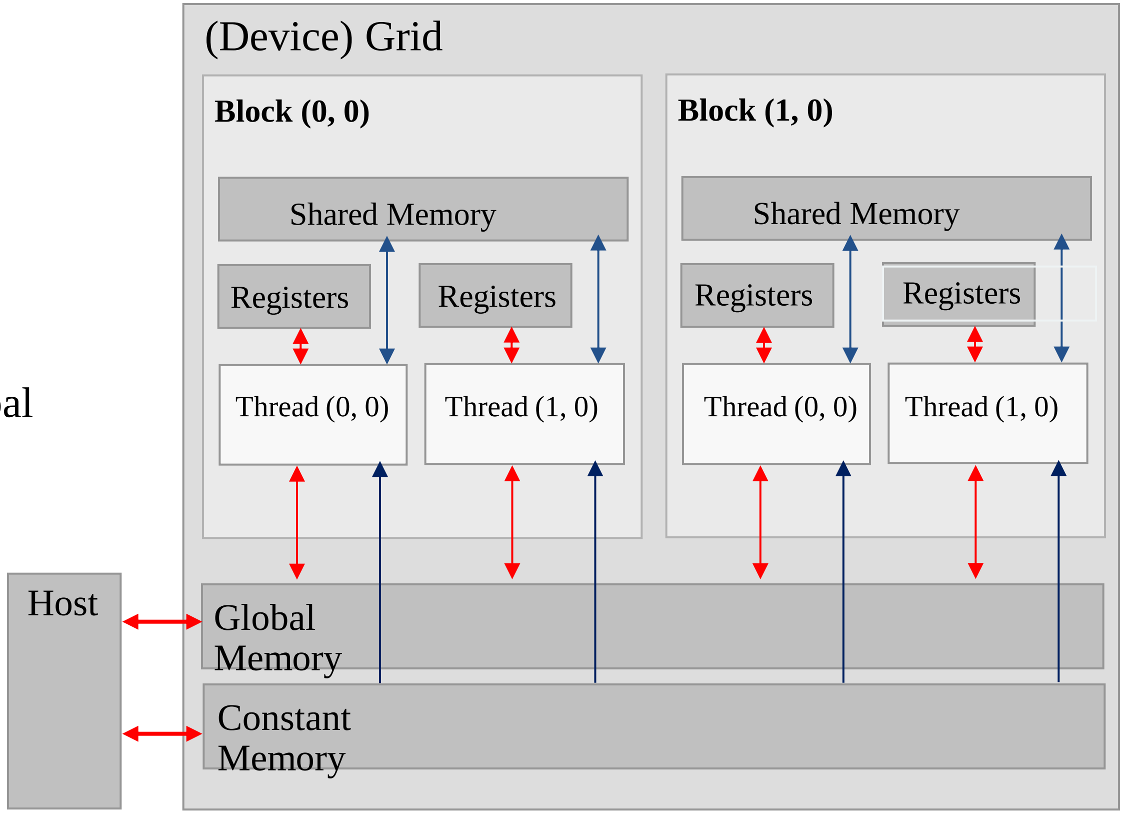
NVIDA :CUDA DEVICE MEMORIES & DATA TRANSFER

CUDA device memory model & Data transfer

- **Device code can:**
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-gold constant

- **Host code can**
  - Transfer data to/from per-grid global and constant memories

❖ global memory & constant memory -devices host code can transfer to and from the device, as illustrated by the bi-directional arrows between these memories and host



Host memory is not shown in the figure

Source & Acknowledgements : NVIDIA, References

CUDA Device Memory Types

- ❖ Global memory and constant memory can be written (**W**) and (**R**) by the host by calling application programming interface (**API**) functions.
- ❖ The constant memory supports short-latency, high-bandwidth, read-only access by the device when all threads simultaneously access the same location.
- ❖ Registers and shared memory are on-chip memories.
- ❖ Variables that reside in these types of memory can be accesses at very high speed in a highly parallel manner.
- ❖ Registers are allocated to individual threads; each thread can only access its own registers.
- ❖ A kernel function typically uses registers to hold frequently accesses variables that are private to each thread.

CUDA : Importance of Memory Access Efficiency

CUDA Device Memory Types - Shared Memory

- ❖ Shared memory is allocated to thread blocks ; all threads in a block can access variables in the shared memory locations allocated to the block.
- ❖ Shared memory is an efficient means for threads to co-operate by sharing their input data and the intermediate results of their work by declaring a CUDA variable in one of the CUDA memory types, A CUDA programmer dictate the visibility and access speed of the variable.
- ❖ CUDA syntax for declaring program variables into the various devices memory.

CUDA Variable Type Qualifiers			
Variable Declaration	**Memory**	**Scope**	**Lifetime**
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Threads	Kernel
__device__, __shared__, int SharedVar;	Shared	Block	Kernel
__device__, int GlobalVar;	Global	Grid	Application
__Device__, ___constant__, int ConstVar;	Constant	Grid	Application

CUDA Device Memory Types - Shared Memory

SCOPE :

- Each declaration gives its declared CUDA variable a scope and lifetime.

- Scope identifies the range of threads of a block, or by all threads of all grids.

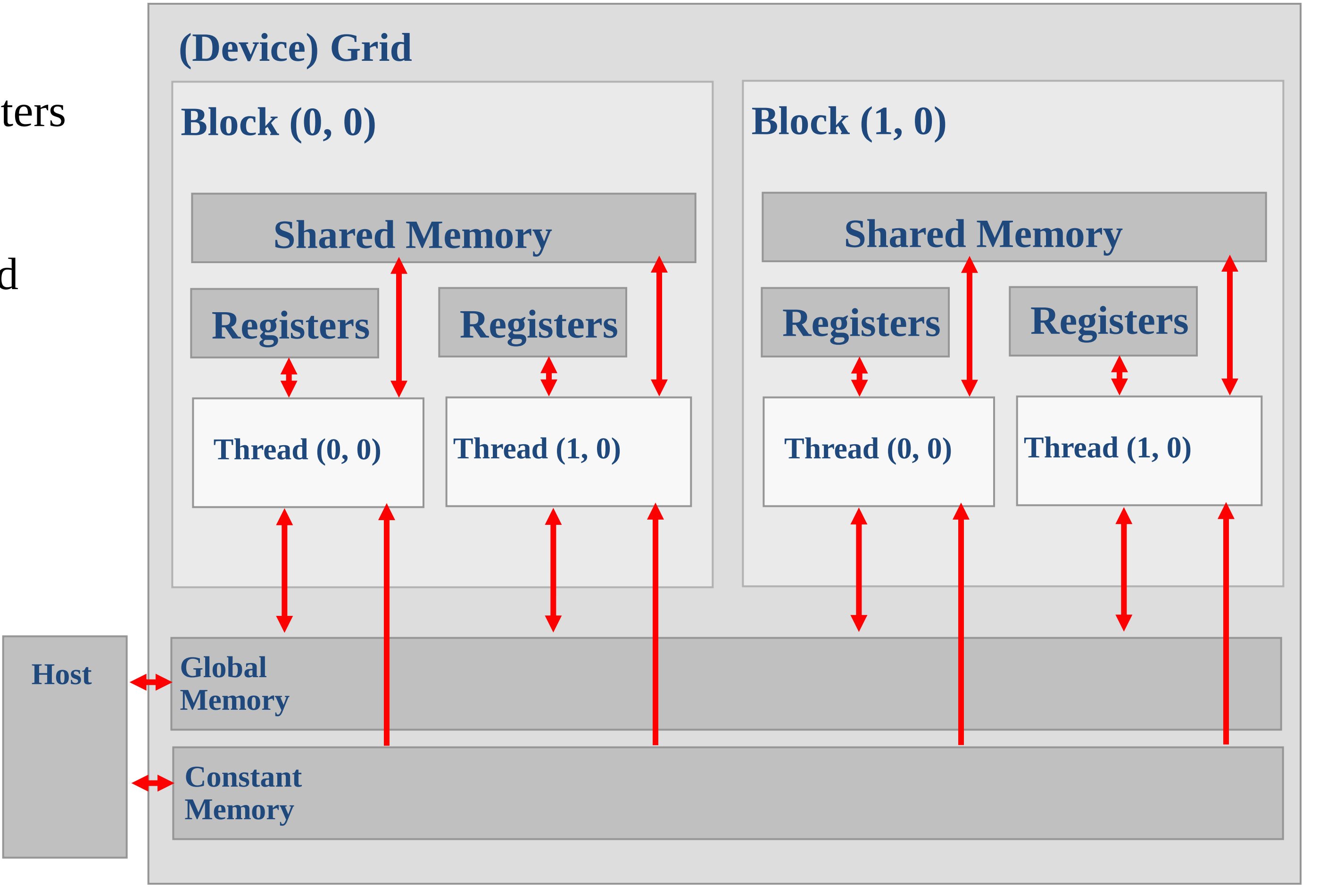
- If the scope of a variable is a single thread, a private version of the variable will be created for every thread; each thread can only access its private version of the variable.

- **For Example :** if a kernel declares a variable whose scope is a thread and it is launched with **1 million threads**, then **1 million versions** of the variable will be created so each thread initializes and used its own version of the variable.

CUDA : Importance of Memory Access Efficiency

CUDA Device Memory Types

- Device code can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-gold constant
- Host code can
  - Transfer data to/from per-grid global and constant memories



Overview of the CUDA device memory model .

Source & Acknowledgements : NVIDIA, References

- Device code can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-gold constant

- Host code can
  - Transfer data to/from per-grid global and constant memories

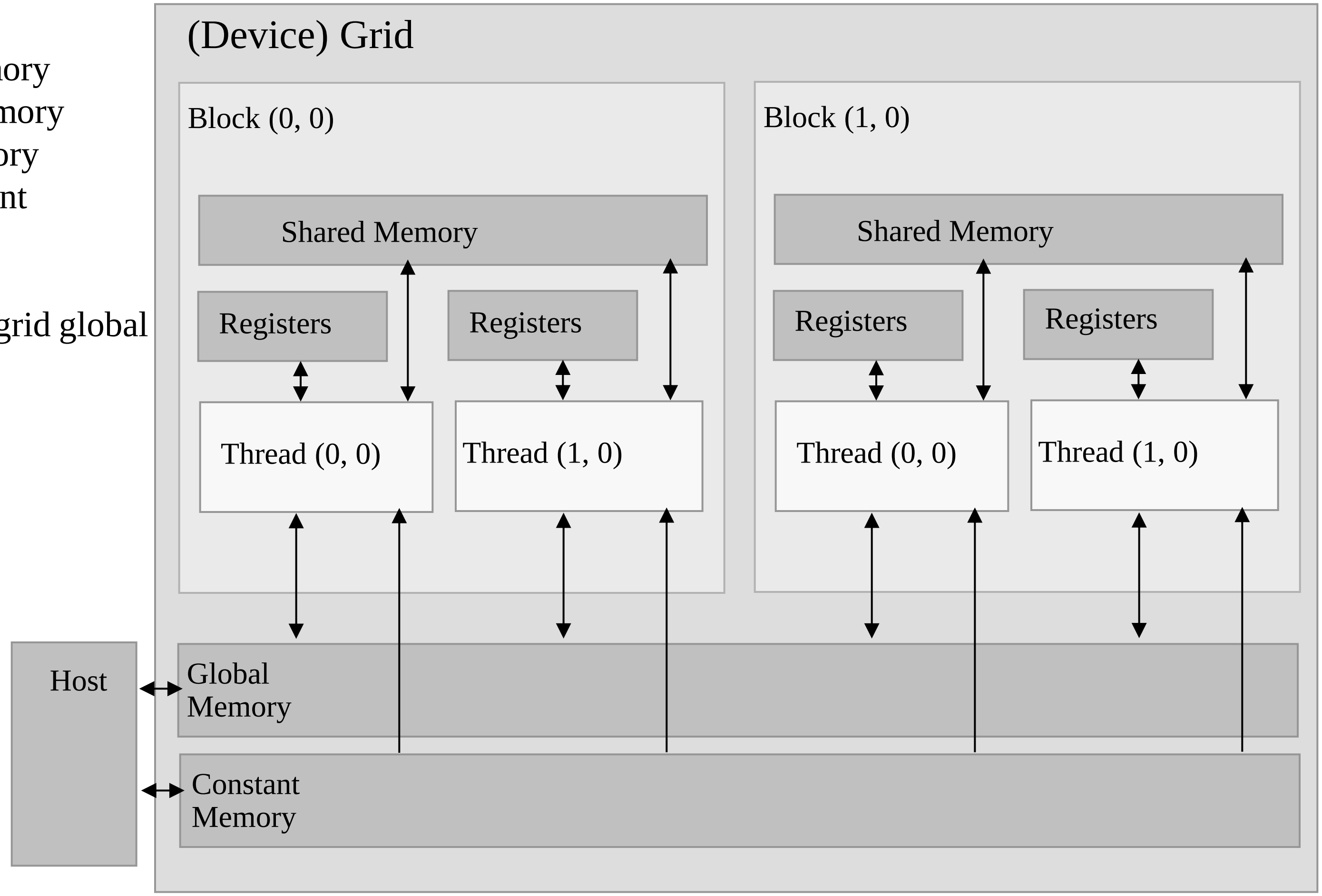
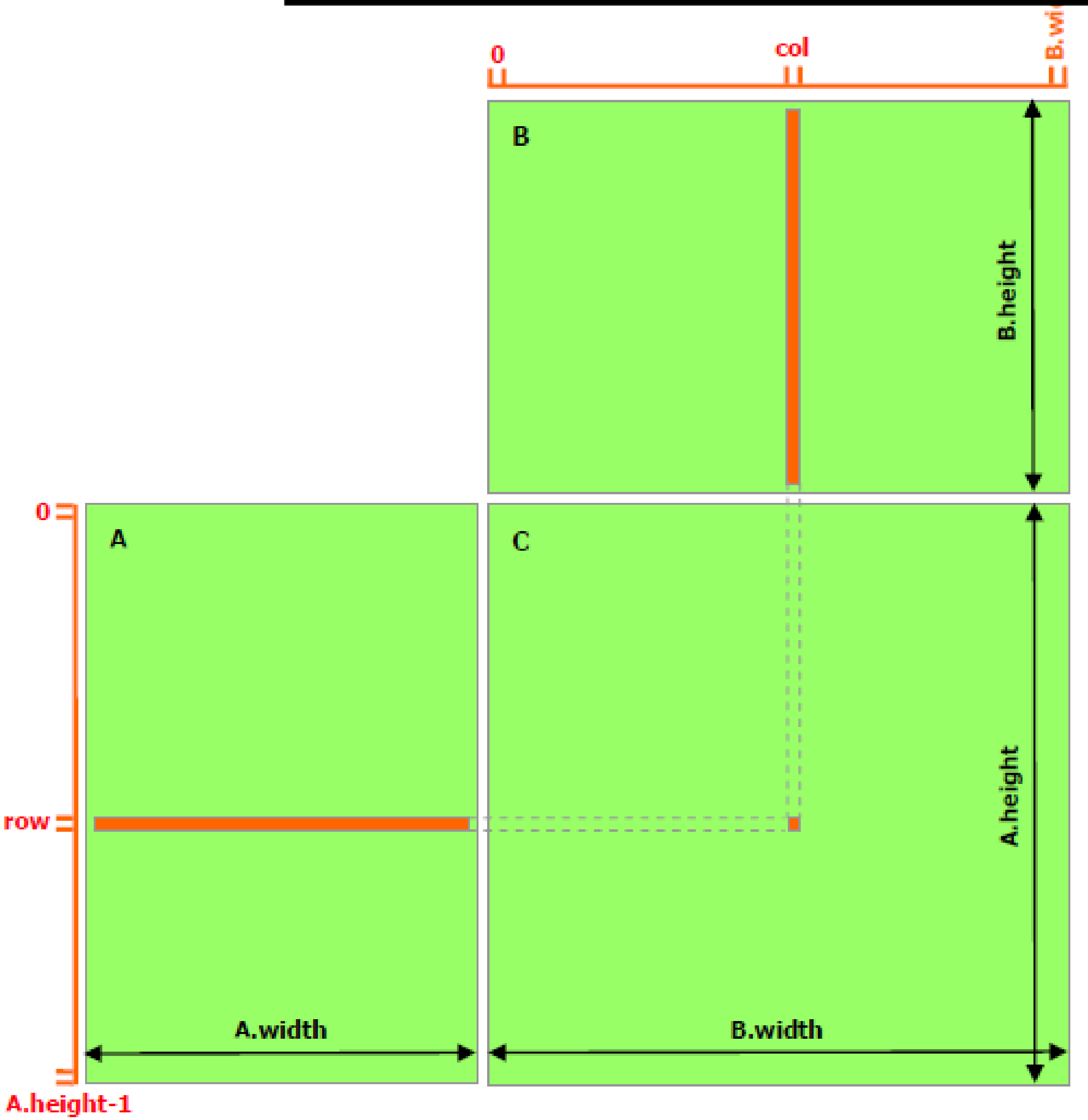


Figure 3.7 Overview of the CUDA device memory model .

Revised matrix multiplication kernel using multiple blocks

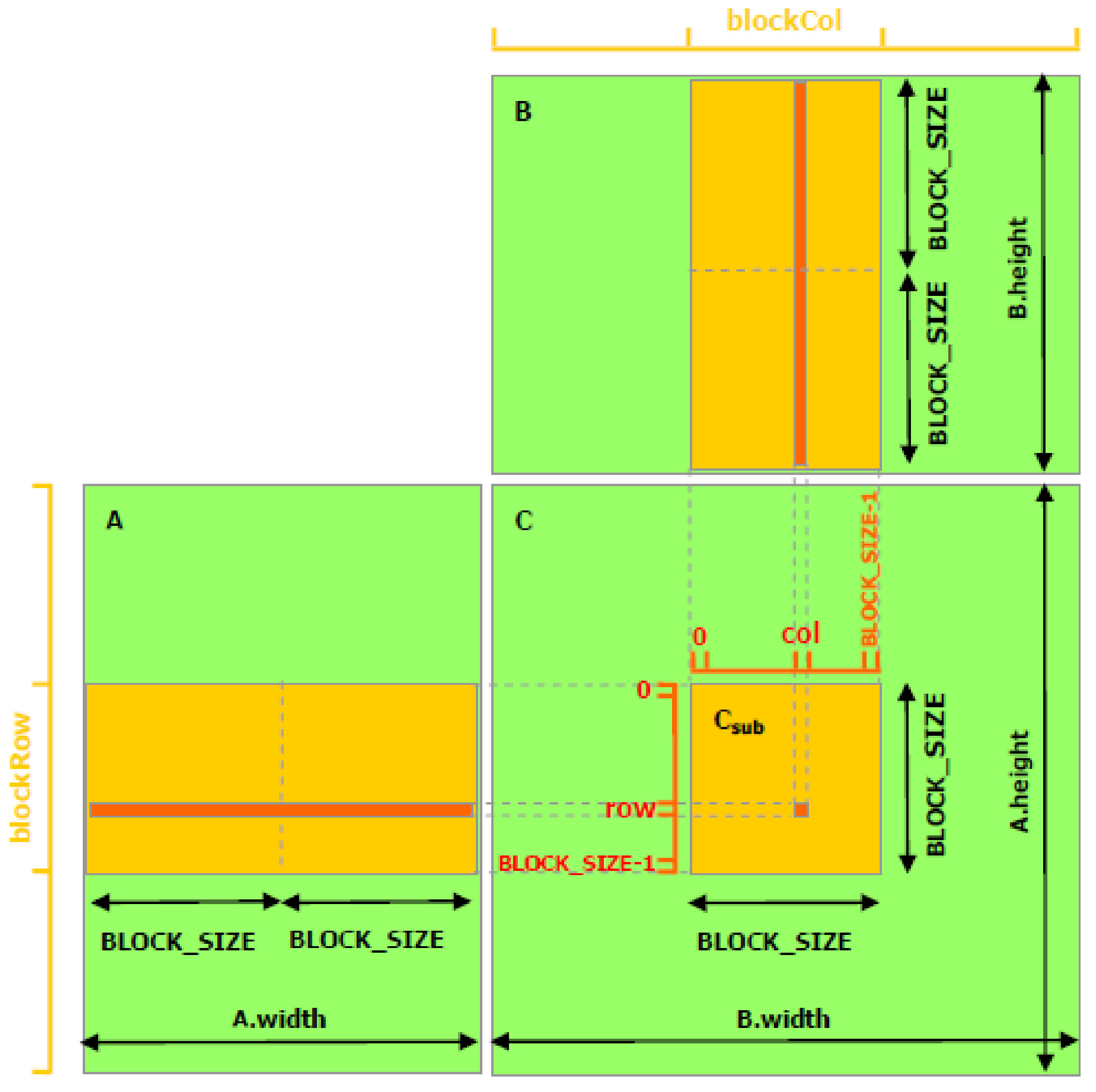
```
__global__ void MatrixMulKernel(float* Md, float* Nd,float* Pd,
int Width)
{
   // Calculate the row index of the Pd element and M
   int Row  = blockIdx.y *TILE_WIDTH + threadIdx.y;

   // Calculate the column index of the Pd element and N
   int Col  = blockIdx.x *TILE_WIDTH + threadIdx.x;

   float Pvalue = 0;
   // each thread computes one element of the block sub-matrix
   for(int k = 0; k < Width; ++k)
      Pvalue +-  Md[Row*Width+k] * Nd[k*Width+Col)];

      Pd[Row*Width_col] = Pvalue;
}
```

NVIDIA :CUDA – Use of Memory



Matrix Multiplication without Shared Memory

NVIDIA :CUDA – Use of Memory



Matrix Multiplication with Shared Memory

CUDA Programming Structure

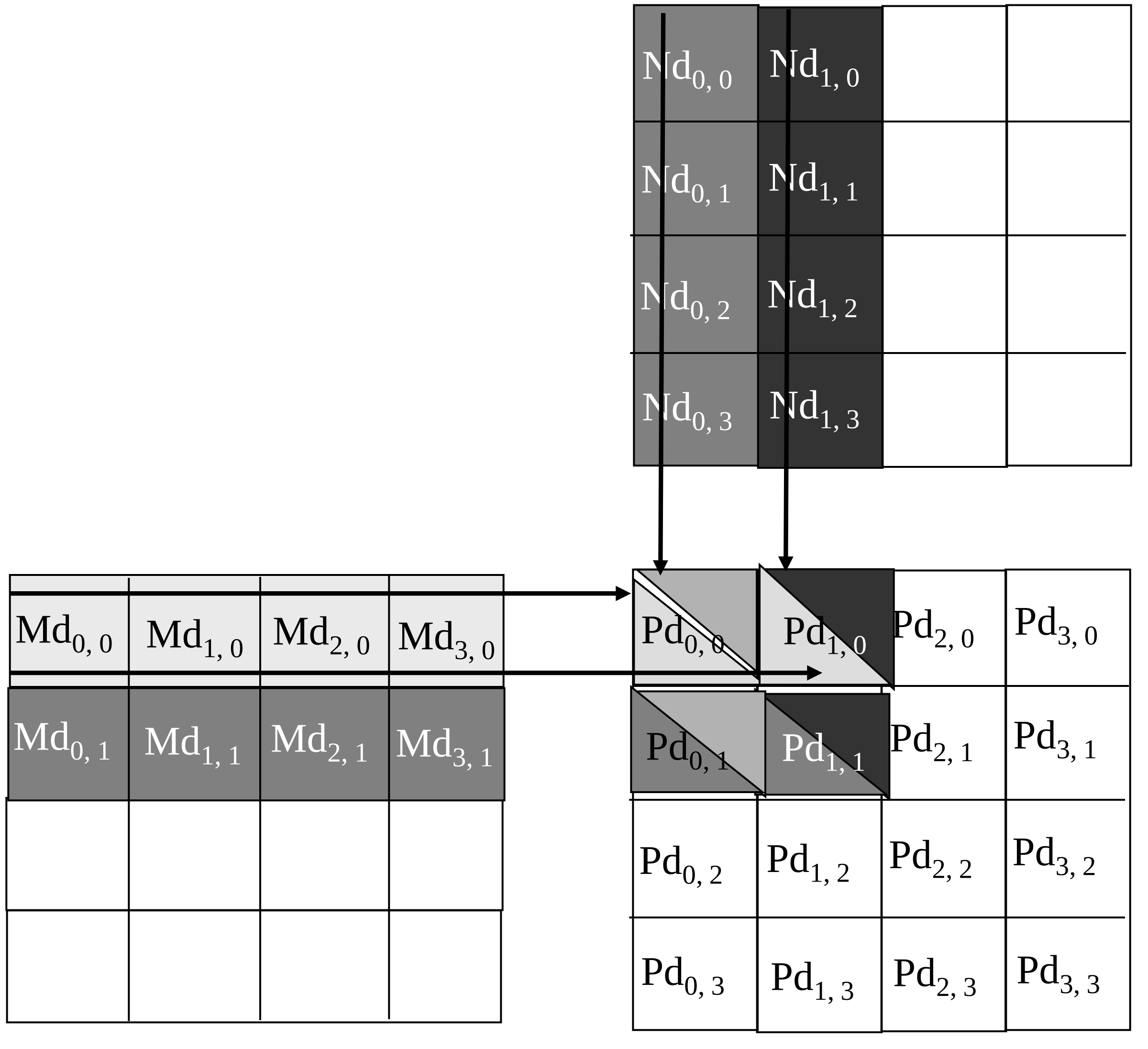


Figure Matrix multiplication actions of one thread block.

Part-II(D)

An Overview of CUDA enabled NVIDIA GPUs:

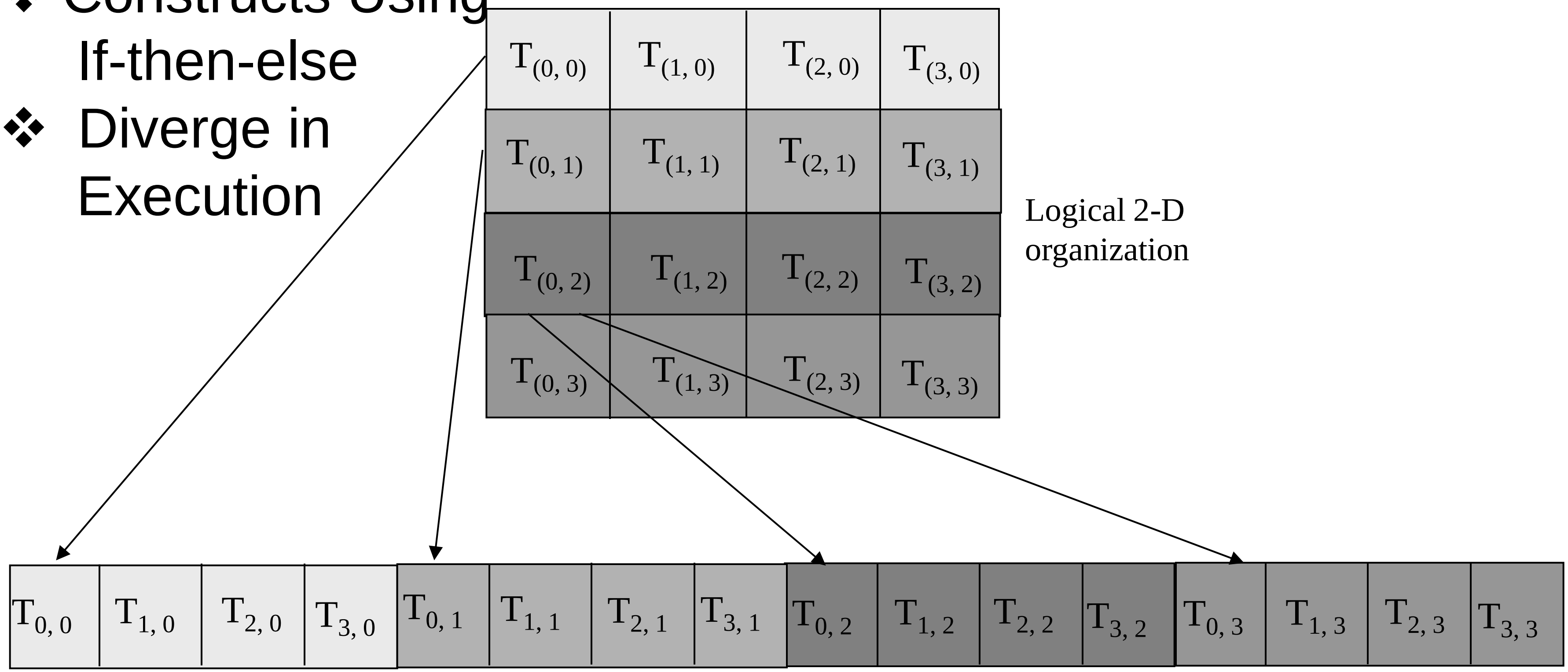
CUDA Execution

Source & Acknowledgements : NVIDIA, References

CUDA Thread Execution - Performance

Warp Parallelism

- Single Instruction – Multiple thread (SIMT)
- Constructs Using If-then-else
- Diverge in Execution



Placing threads into linear order

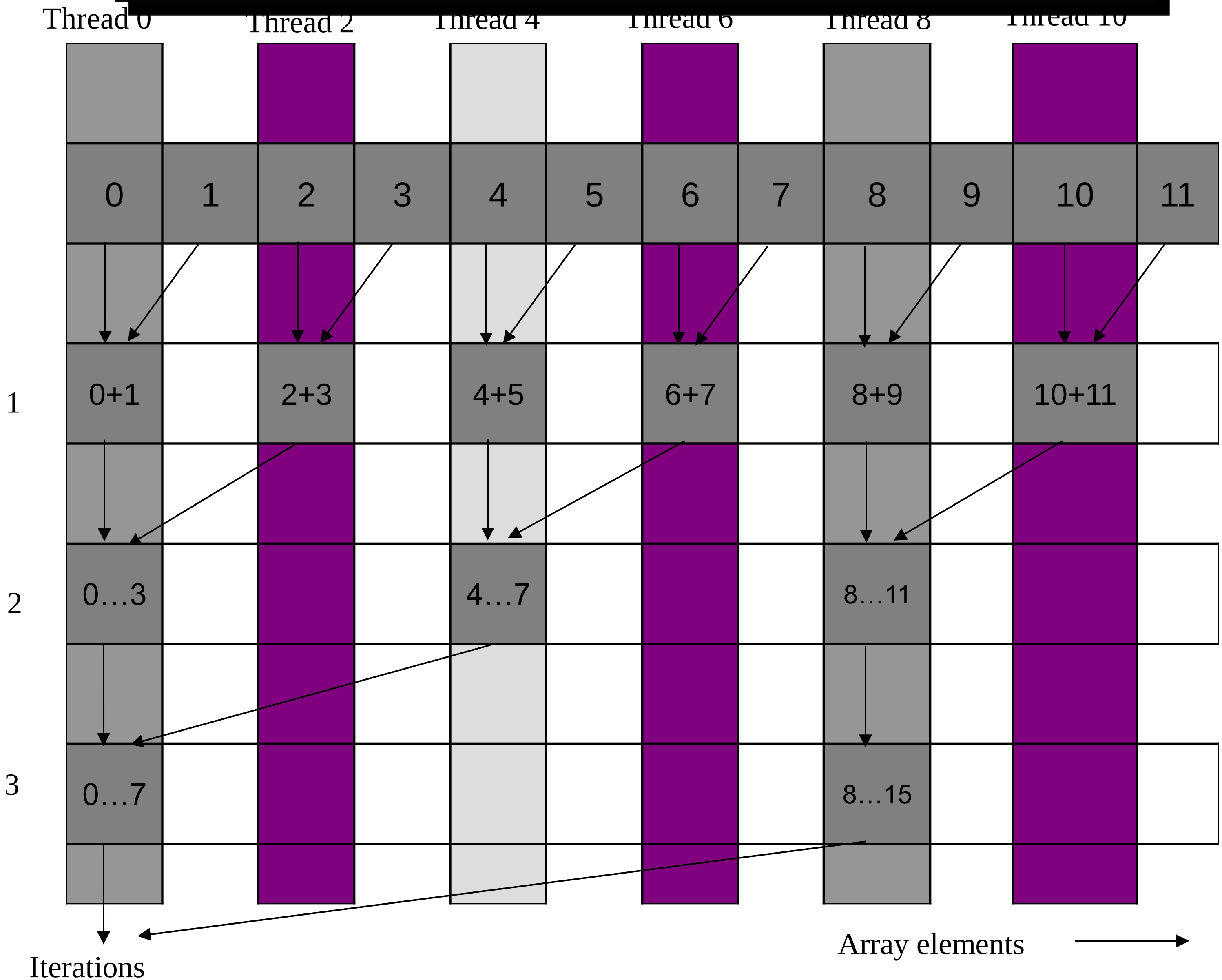
CUDA Thread Execution - Performance

```
_shared_float partialSum[ ]

Unsingned int t = threadsIdx.x;

for (unsigned int stride = 1;
        stride < blockDim.X; stride *=2)
 {
   __syncthreads ( );
   If (t % (2*stride) == 0)
  partialSum[t] + = partialSum[ t +stride];
}
```

A simple sum reduction kernel.

Source & Acknowledgements : NVIDIA, References

CUDA Thread Execution - Performance



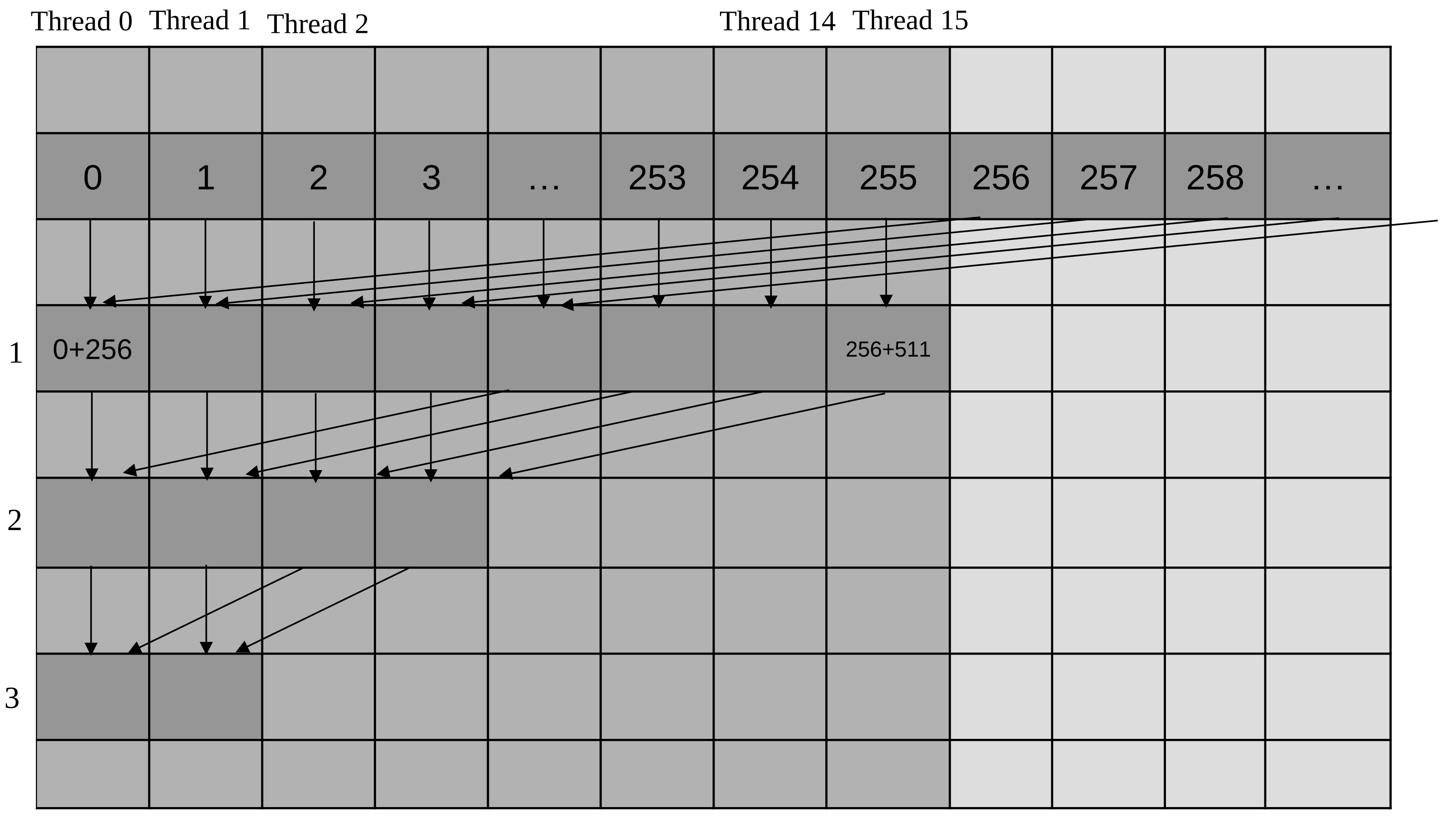
A Deduction of the sum reduction kernel.

CUDA Thread Execution - Performance

```
_shared_float partialSum[ ]

Unsingned int t = threadsIdx.x;

for (unsigned int stride = 1;
         stride < blockDim.X; stride *=2)
{
     __syncthreads ( );
     If (t  < stride)
     partialSum[t] + = partialSum[ t +stride];
}
```

A kernel with les thread divergence.

CUDA Thread Execution - Performance

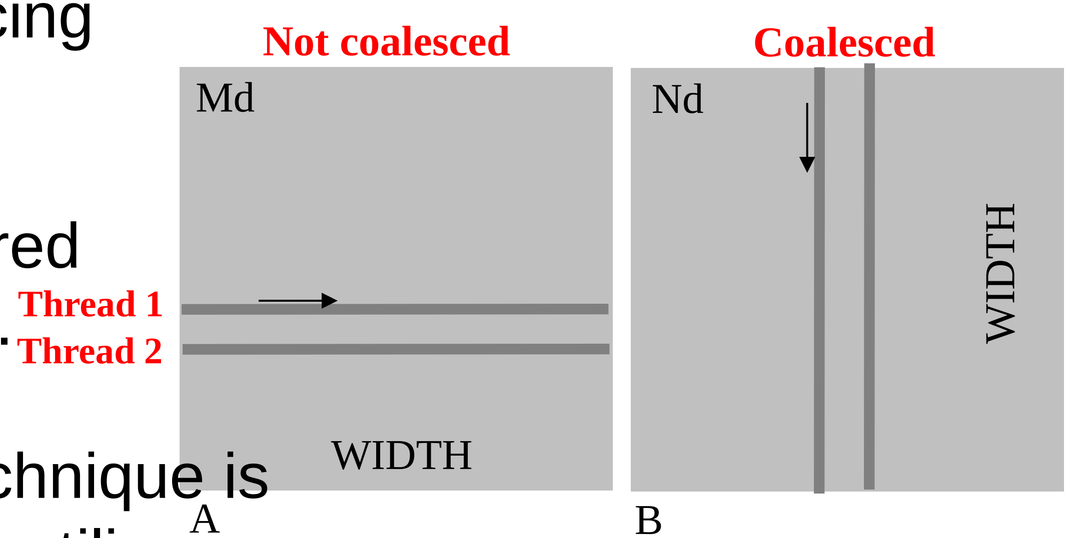


Execution of the revised algorithm.

Global Memory Bandwidth

- ❖ Kernel performance is related to accessing data in the global memory
- ❖ Use of Memory Coalescing

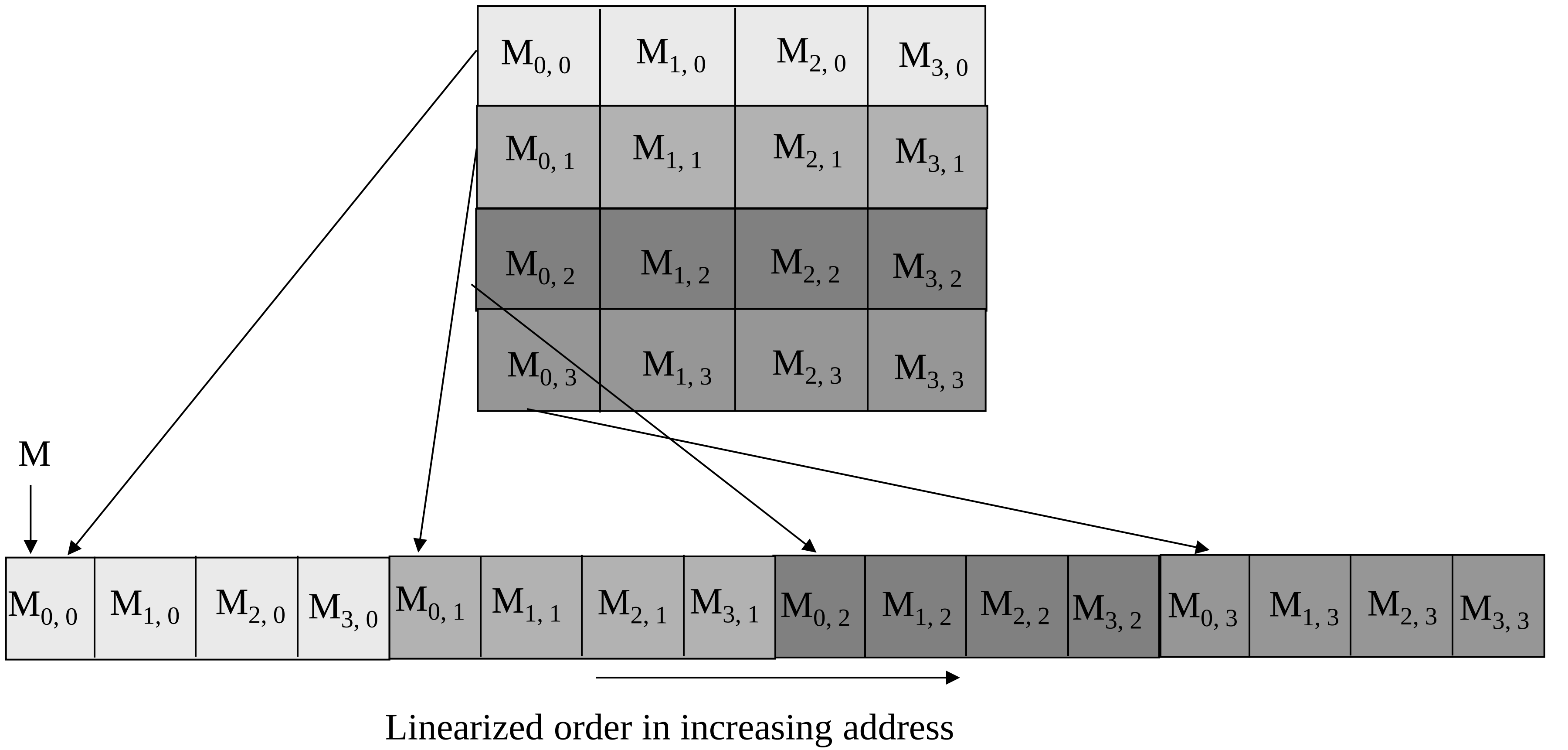
  Move the data from the global memory into shared memories and registers.



- ❖ Memory Coalescing technique is used in conjunction with tiling technique

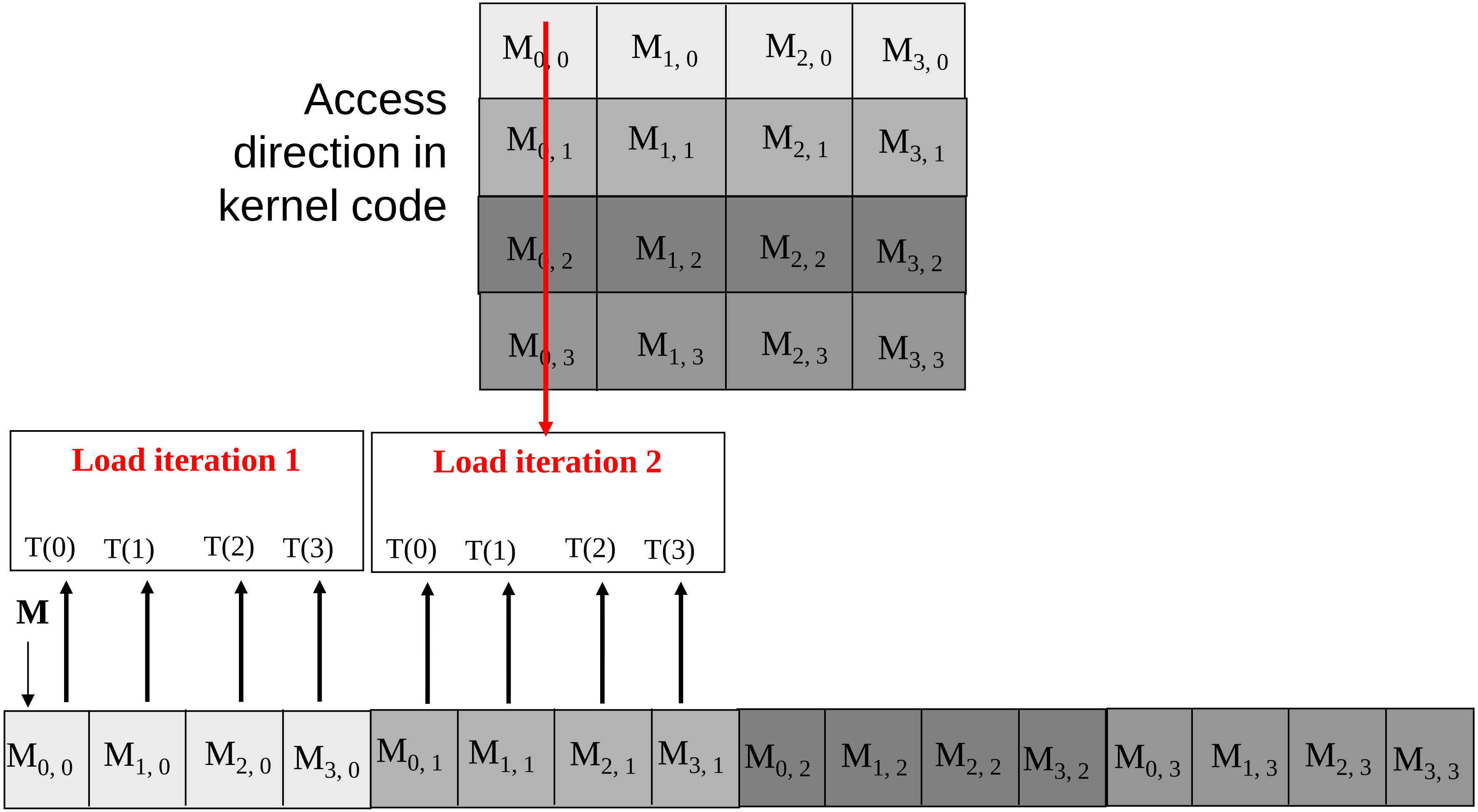
Memory access pattern for coalescing.

Global Memory Bandwidth



Placing matrix elements order into linear order.

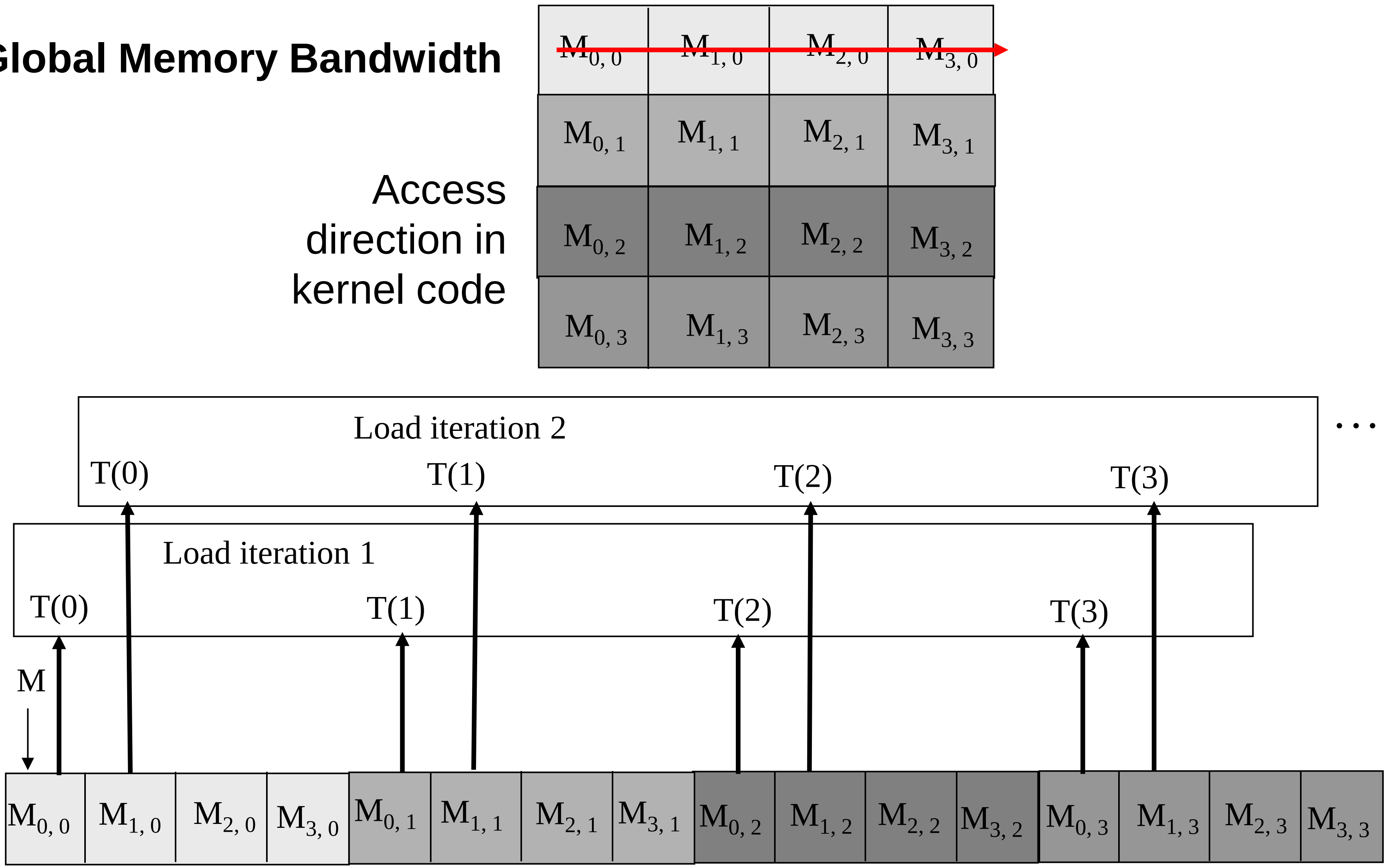
Global Memory Bandwidth



A coalesced access pattern.

CUDA Thread Execution - Performance

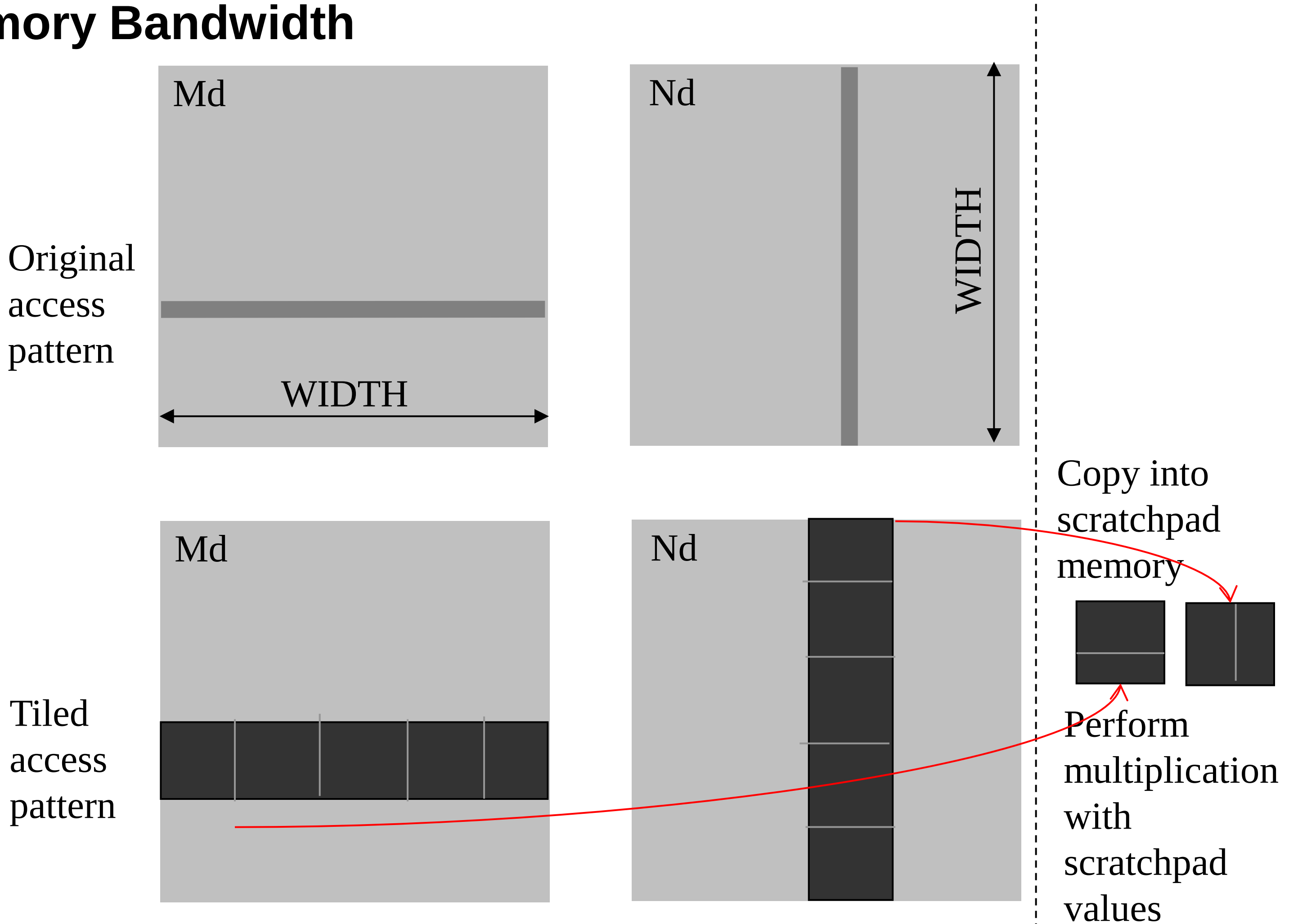
Global Memory Bandwidth



A uncoalesced access pattern.

CUDA Thread Execution - Performance

Global Memory Bandwidth



Using shared memory to enable coalescing.

```
_global_ void MatrixMulKernel(float*Md, float*Nd, gloat*Pd, int width)
{
1. _shared_float Mds[TILE_WIDTH][TILE_WIDTH];
2. _shared_float Nds{TILE_WIDTH][TIKE_WIDTH];

3. int bx = blockIdx .x; int by = blockidx.y;
4. int tx = threadIdx.x; int ty = threadIdx.y;

// Identify the row and column of the Pd element to work on
5. int Row = by * TILE_WIDTH + ty;
6. int Col = bx * TILE_WIDTH + tx;

7. Float Pvalue = 0;
// Loop over the Md and Nd tiles required to computer the Pd element
8. For (int m = 0; m < Width/TILE_WIDTH; ++m) {

//Collaborative loading of Md and Nd tiles into shared memory
9.      Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
10.    Nds[ty][tx] = Nd (m*TILE_WIDTH + ty) * Width + Col];
11.    _synchthreads ( );

12.   for ( int k = 0; k < TILE_WIDTH; ++k)
13.     Pvalue +=Mds [ty][k] * Nds[k] [tx];

14.  Pd [Row] [Col] = Pvalue;
     }
}
```

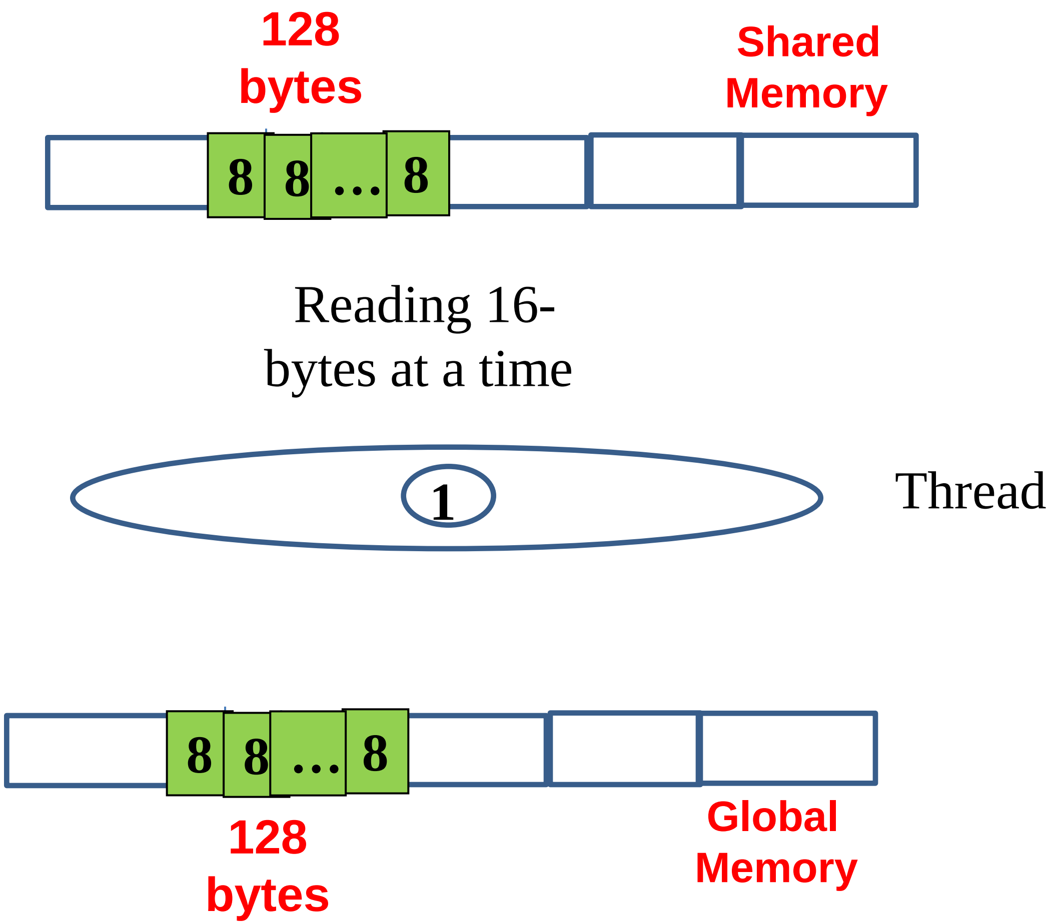
Md Nd

Global Memory Bandwidth

The matrix multiplication kernel using shared memories.

GPU performance : Memory Coalescing

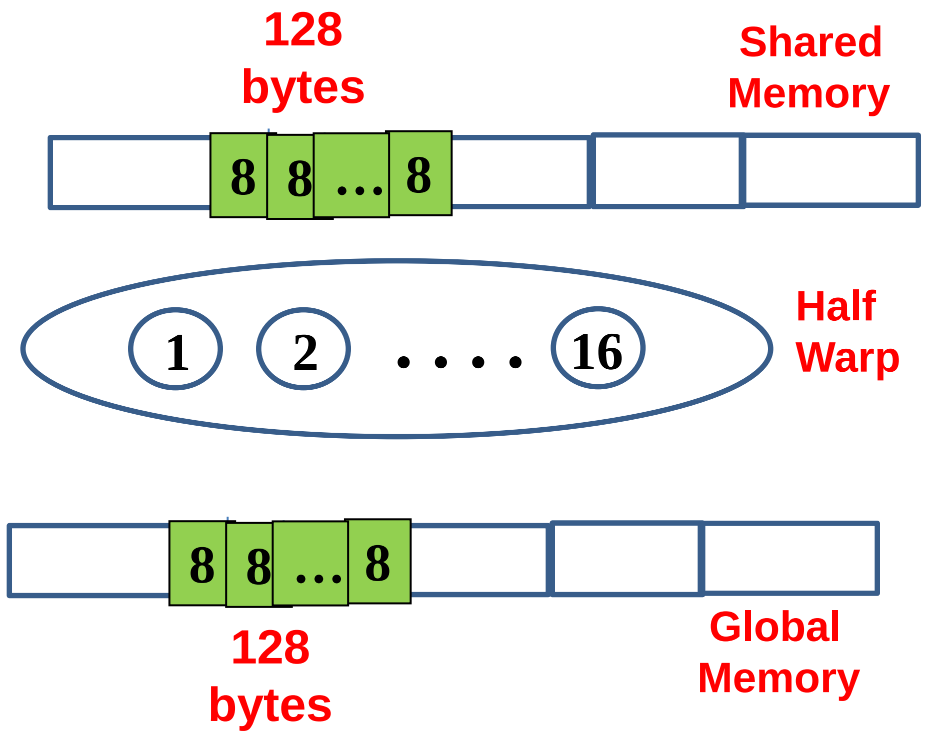
- Request >16-bytes serviced iteratively



GPU performance : Memory Coalescing

Read-Write operation:

- ❖ Collectively by threads in half warp
- ❖ Coalesce memory accesses in single transaction
- ❖ Threads of half-warp collaborate and utilize the memory coalescing

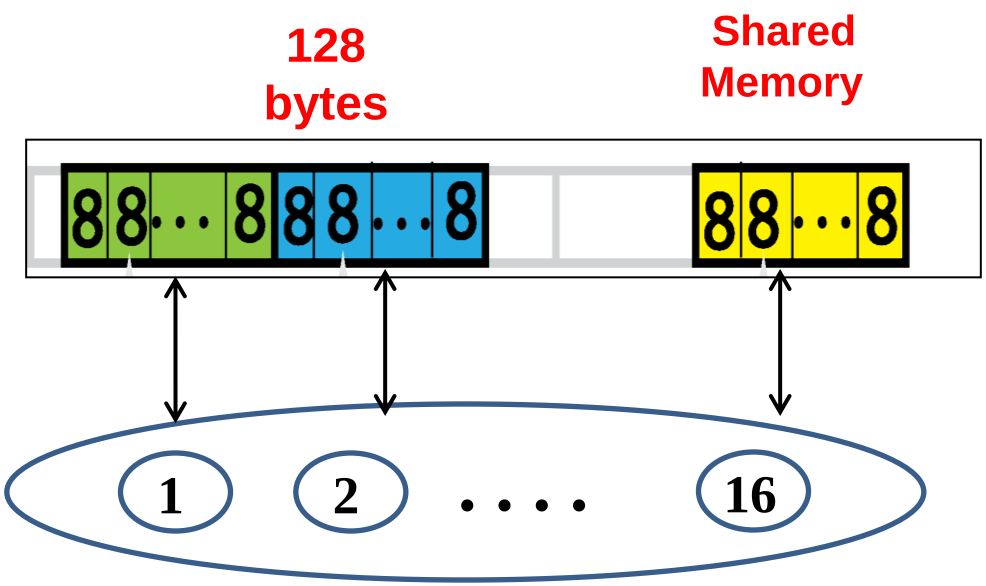


Source & Acknowledgements : NVIDIA, References

GPU performance : Memory Coalescing

Modify operation:

- ❖ Threads work individually
- ❖ on data Iteratively after memory transfer
- ❖ Bank conflicts lead to serialization of memory requests



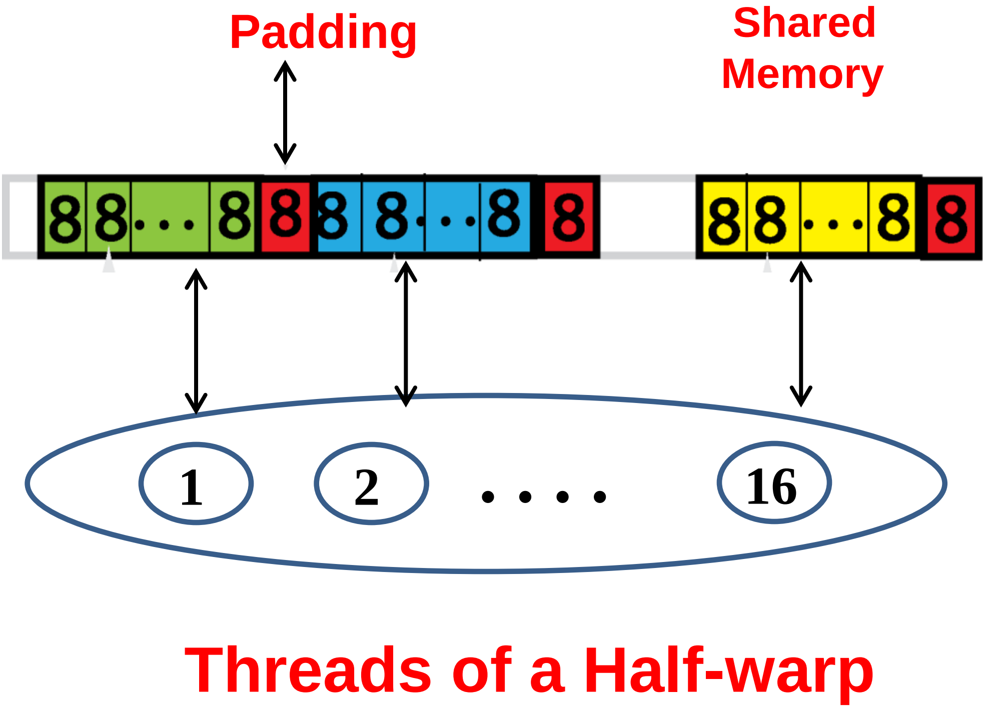
Threads of a Half-warp

Source & Acknowledgements : NVIDIA, References

GPU performance : Memory Coalescing

Modify operation:

❖Pad offset of 8 bytes,
Thereby reduce bank conflicts



Source & Acknowledgements : NVIDIA, References

CUDA Thread Execution - Performance

Global Memory Bandwidth : Dynamic Partitioning of SM resources

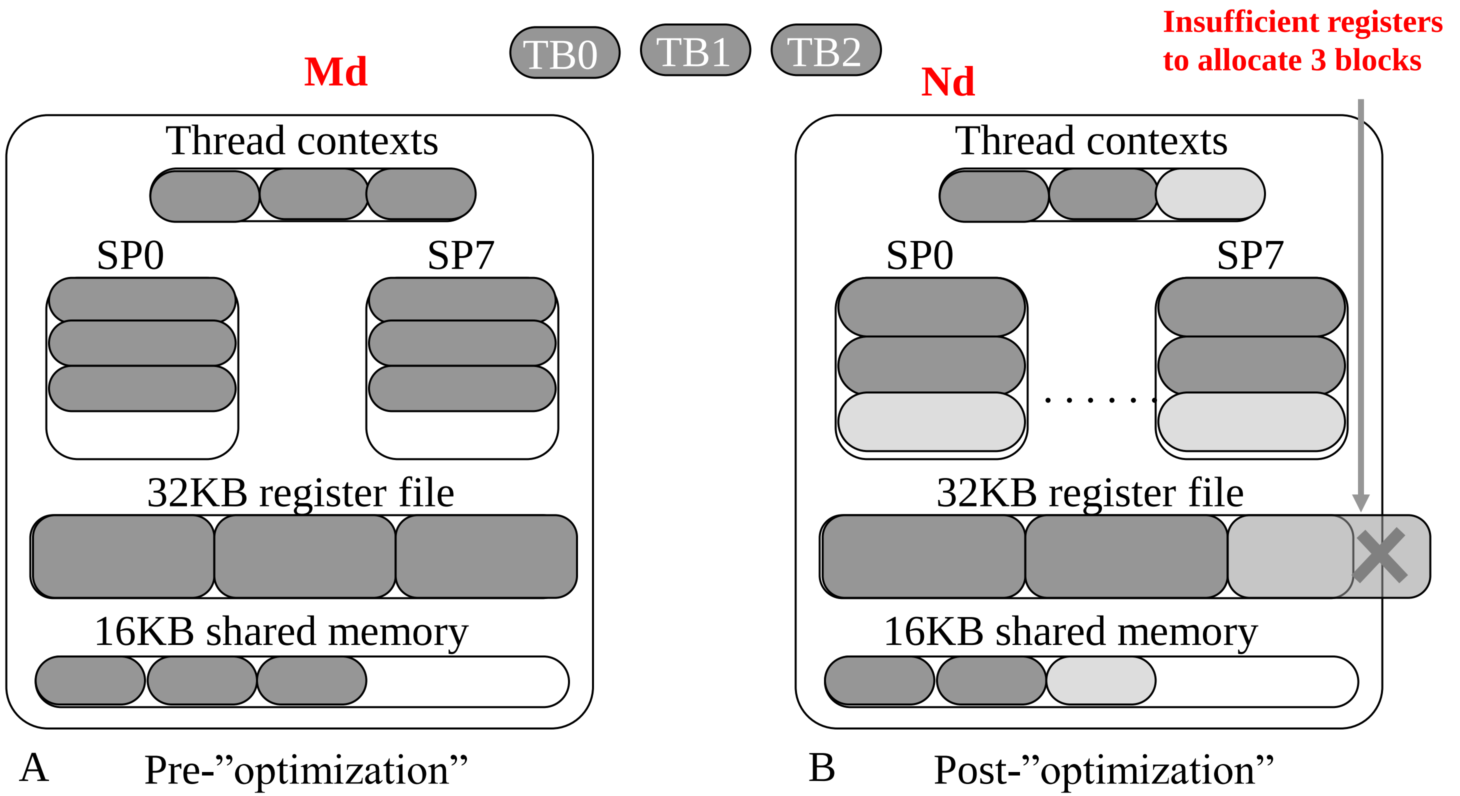


Figure. Interaction of resource limitations.

CUDA Thread Execution - Performance

Global Memory Bandwidth : Prefetching

FP Instruction, Load Instruction, Branch Instruction

```
Loop{

Load current tile to shared
     memory

_synchthreades( )

Computer current tile

_synchthreads( )
}
```

A Without prefetching

```
Load first tile from global memory into
        registers

Loop {
Deposit tile from registers to shared
     memory

_synchthreads( )

Load next tile from global memory into
     registers

Computer current tile

_synchthreads ( )
}
```

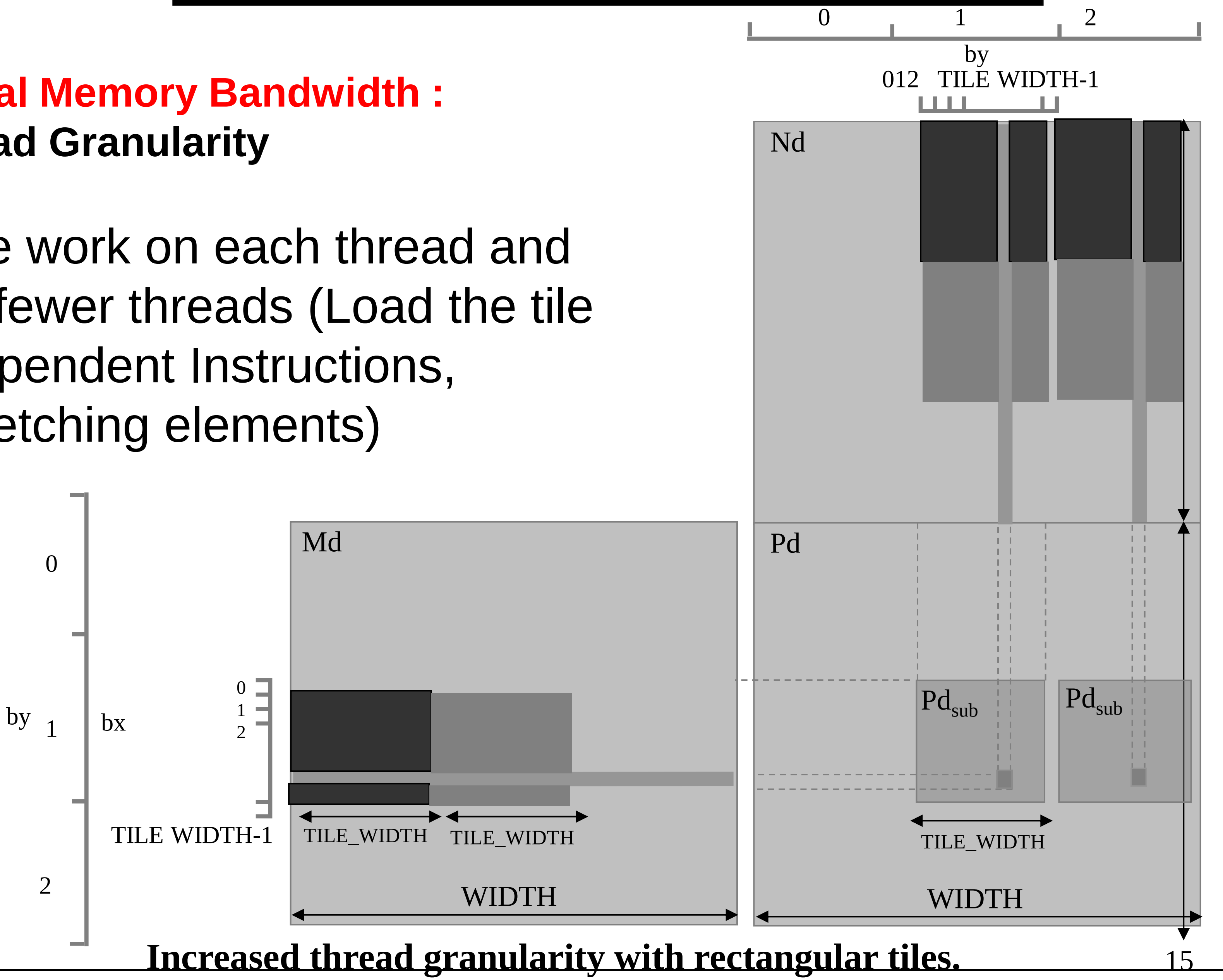
B With prefetching

CUDA Thread Execution - Performance

Global Memory Bandwidth :

Thread Granularity

More work on each thread and use fewer threads (Load the tile Independent Instructions, Prefetching elements)

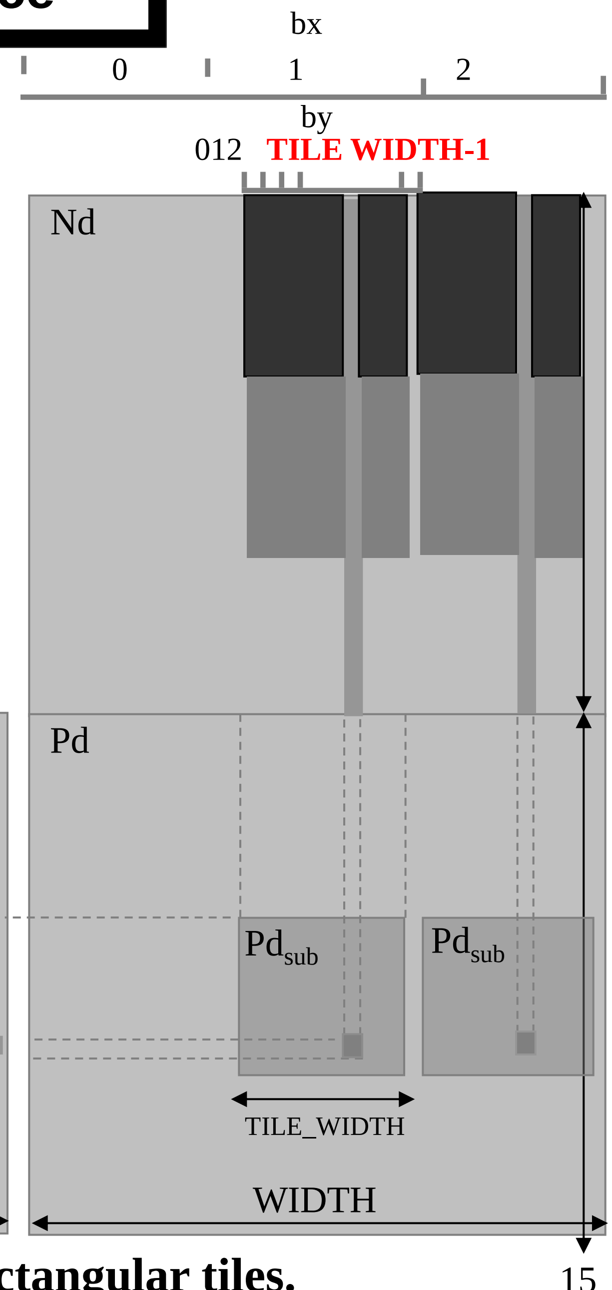


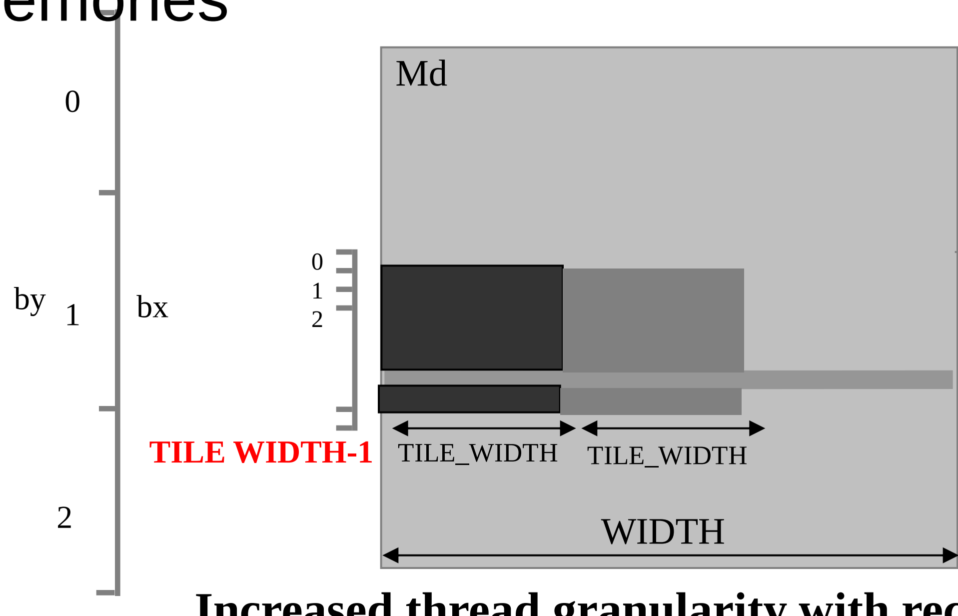
Increased thread granularity with rectangular tiles.

CUDA Thread Execution - Performance

Global Memory Bandwidth :
Thread Granularity

- ❖ Loading of Tiles into registers and depositing these tiles into shared memories
- ❖ No. of Blocks running on shared memories





Increased thread granularity with rectangular tiles.

CUDA Thread Execution - Performance

Instruction mix consideration.

- ❖ Loading of Tiles into registers and depositing these tiles into shared memories
- ❖ No. of Blocks running on shared memories

```
for (int k = 0; k < BLOCK_SIZE; ++k)
    Pvalue += Ms [ty][k] * Ns [k] 9tx0;
```

(a) Loop incurs overhead instruction

```
Pvalue += Ms[ty][0] * Ns[0][tx] += Ms[ty][15]*Ns[15][tx];
```

(b) Loop unrolling improves instruction mix.

- ❖ Executes two floating arithmetic, one loop branch instruction, two address arithmetic instructions, one loop counter increment instruction,

NVIDIA Tool Kit : CUBLAS

- ❖ CUBLAS is an implementation of BLAS (Basic Linear Algebra Subprogram) on top of the CUDA driver. It allows access to the computational resources of NVIDIA GPUs.

  The library is self-contained at the API level, that is, no direct interaction with the CUDA driver is necessary.

- ❖ The basic model by which applications use the CUBLAS library is to:
  - Create matrix and vector objects in GPU memory space
  - Fill them with data
  - Call a sequence of CUBLAS functions

  - Upload the results from GPU memory space back to the host
- ❖ CUBLABS provides helper functions for creating and destroying objects in GPU space, and for writing data to and retrieving data from these objects

Source : NVIDIA, References

CUDA – BLAS Supported features

- BLAS functions implemented (single precision only):
  - Real data: level 1, 2 and 3
  - Complex data: level a and CGEMM

  (Level 1=vector vector O(N), Level 2=matrix vector $O(N^2)$, Level 3=matrix matrix $O(N^3)$)

- For maximum compatibility with existing Fortran environments, CUBLAS uses column-major storage, and 1-based indexing: Since C and C++ use row-major storage, this means applications cannot use the native C array semantics for two-dimensional arrays. Instead, macros or inline functions should be defined to implement matrices on top of one-dimensional arrays.

Source : NVIDIA, References

CUDA - Using CUBLAS

- The interface to the CUBLAS library is the header file **`cublas.h`**

- Function names: cublas(Original name).
  cublasSgemm

- Because the CUBLAS core functions (as opposed to the helped functions) do not return error status directly, CUBLAS provides a separate function to retrieve the last error that was recorded, to aid in debugging

- CUBLAS is implemented using the C-based CUDA tool chain, and thus provides a C-style API. This makes interfacing to applications written in C or C++ trivial.

Source : NVIDIA, References

CUDA - cublasInit, cublasShutdown

❖ **cublasStatus cublasInit()**
initializes the CUBLAS library and must be called before any other CUBLAS API function is invoked. It allocates hardware resources necessary for accessing

❖ **cublasStatus cublasShutdown()**
releases CPI-side resources used by the CUBLAS library. The release of GPU-side resources may be deferred until the application shuts down.

Source : NVIDIA, References

CUDA - cublasGetError, cublasAlloc, cublasFree

❖ **cublasStatus cublasGetError()**
returns the last error that occurred on invocation of any of the CUBLAS core functions. While the CUBLAS helper functions return status directly, the CUBLAS core functions do not, improving compatibility with those existing environments that do not expect BLAS functions to return status. Reading the error status via cublasGetError() rests the internal error state to CUBLAS_STATUS_SUCCESS.

❖ **cublasStatus cublasAlloc (int n, int elemSize, void **devicePtr)**
creates an object in GPU memory space capable of holding an array of n elements, where each clement requires elemSize bytes of storage. Note that this is a device pointer that cannot be dereferenced in host code.
cublasAlloc() is a wrapper around cudaMalloc().
Device pointers returned by cublasAlloc() can therefore be passed to any CUDA device kernels, not just CUBLAS functions.

❖ **cublasStatus cublasFree(const void *device Ptr)**
destroys the object in GPU memory space referenced by device Ptr.

Source : NVIDIA, References

CUDA - cublasSetVector, cublasGetVector

❖ **cublasStatus cublasSetVector(int n, int elemSize, const void *x, int incx, void *y, int incy)**

copies n elements from a vector x in CPU memory space to a vector y in GPU memory space. Elements in both vectors are assumed to have a size of elemSize bytes. Storage spacing between consecutive elements in incx for the source vector x and incy for the destination vector y

❖ **cublasStatus cublasGetVector (int n, int elemSize, const void *x, int incx, void *y, int incy)**

copies n elements from a vector x in GPU memory space to a vector y in CPU memory space. Elements in both vectors are assumed to have a size of elemSize bytes. Storage spacing between consecutive elements is incx for the source vector x and incy for the destination vector y

Source : NVIDIA, References

CUDA - cublasSetMatrix, cublasGetMatrix

- **cublasStatus cublasSetMatrix(int rows, int cols, int elemSize, const void *A, int lda, void *B, int ldb)**

copies a tile of rows x cols elements from a matrix A in CPU memory space to a matrix B in GPU memory space. Each element requires storage of elemSize bytes. Both matrices are assumed to be stored in column-major format, with the leading dimension (that is, the number of rows) of source matrix A provided in lda, and the leading dimension of destination matrix B provided in ldb

- **cublasStatus cublasGetVector (int rows, int cols, int elemSize, const void *A, int lda, void *B, int ldb)**

copies a tile of rows x cols elements from a matrix A in GPU memory space to a matrix B in CPU memory space. Each element requires storage of elemSize bytes. Both matrices are assumed to be stored in column-major format, with leading dimension (that is, the number of rows) of source matrix A provided in lda, and the leading dimension of destination matrix B provided in ldb

CUDA - Calling CUBLAS from FORTRAN

❖ Fortran-to-C calling conventions are not standardized and differ by platform and tool chain.

In particular, differences may exist in the following areas:

- Symbol names (capitalization, name decoration)
- Argument passing (by value or reference)
- Passing of string arguments (length information)
- Passing of pointer arguments (size of the pointer)
- Returning floating-point or compound data types (for example, single-precision or complex data type)

❖ CUBLABS provides provides wrapper functions (in the file fortran.c) that need to be compiled with the user preferred tool chain.Providing source code allows users to make any changes necessary for a particular platform and tool chain.

Source : NVIDIA, References

Part-II(E)

An Overview of CUDA enabled NVIDIA GPUs:

CUDA Memories

Source & Acknowledgements : NVIDIA, References

CUDA Tool Kit 5.0 Preview

- **Nsight Eclipse Edition** : Develop & Debug and Profile GPU Accelerated Applications on Linux - **All in on IDE**
- **RDMA for GPUDirect :** Direct Communication between GPUs and other PCIe Devices
- **GPU Library Object Linking :** Easily Accelerate parallel nested loops starting with Tesla K20 Kepler GPUs
- **Dynamic Parallelism :** library of templated performance primitives such as sort, reduce, etc.
- NVIDIA Performance Primitives (NPP) library for image/video processing
- Layered Textures for working with same size/format textures at larger sizes and higher performance

Source : NVIDIA, References

CUDA Tool Kit 5.0 Preview

❖ **RDMA for GPUDirect : Features**

- **Accelerated communication with network and storage devices :** Avoid unnecessary system memory copies and CPU overhead by copying data directly to/from pinned CUDA host memory

- **Peer-to-Peer Transfers between GPUs :** Use high-speed DMA transfers to copy data from one GPU directly to another GPU in the same system

- **Peer-to-Peer memory access :** Optimize communication between GPUs using NUMA-style access to memory on other GPUs from within CUDA kernels

Source : NVIDIA, References

CUDA Tool Kit 5.0 Preview

❖ RDMA for GPUDirect : Features

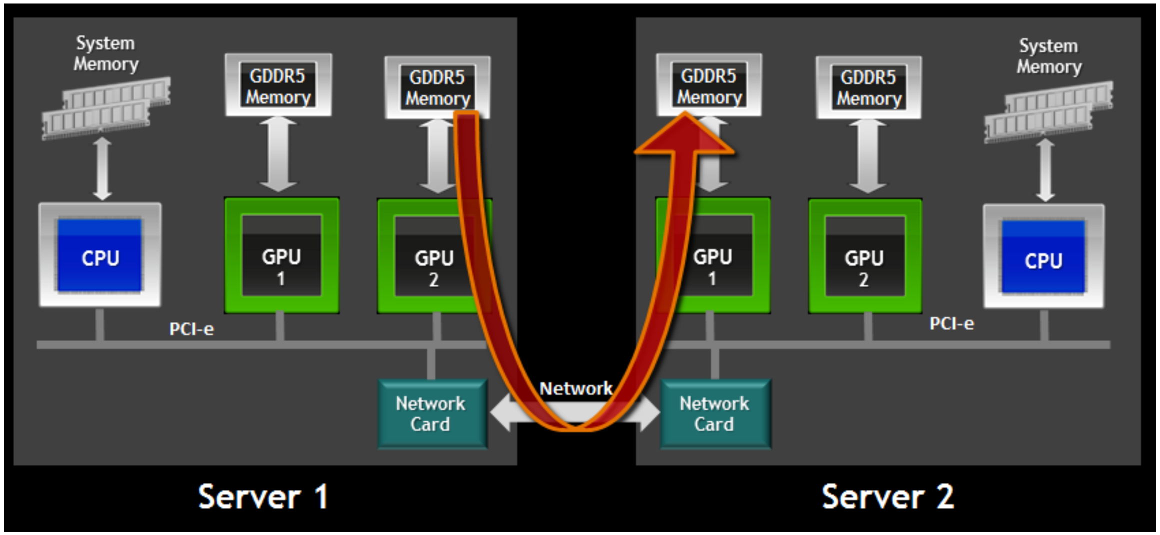
- **Peer-to-Peer memory access :** Optimize communication between GPUs using NUMA-style access to memory on other GPUs from within CUDA kernels
- **RDMA :** Eliminate CPU bandwidth and latency bottlenecks using direct memory access (DMA) between GPUs and other PCIe devices, resulting in significantly improved MPISendRecv efficiency between GPUs and other nodes (new in CUDA 5)
- **GPUDirect for Video :** Optimized pipeline for frame-based devices such as frame grabbers, video switchers, HD-SDI capture, and CameraLink devices.

Source : NVIDIA, References

CUDA Tool Kit 5.0 Preview

❖ RDMA for GPUDirect : Features

GPUDirect™ Support for RDMA, Introduced with CUDA 5



Eliminate CPU bandwidth and latency bottlenecks using direct memory access (DMA) between GPUs and other **PCIe devices**, resulting in significantly improved **MPISendRecv** efficiency between GPUs and other nodes (new in CUDA 5)

Source : http://www.nvidia.in/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

Source : NVIDIA, References

CUDA Tool Kit 5.0 Preview

❖ RDMA for GPUDirect : Features

GPUDirect™ Support for Accelerated Communication with Network and Storage Devices

Without GPUDirect

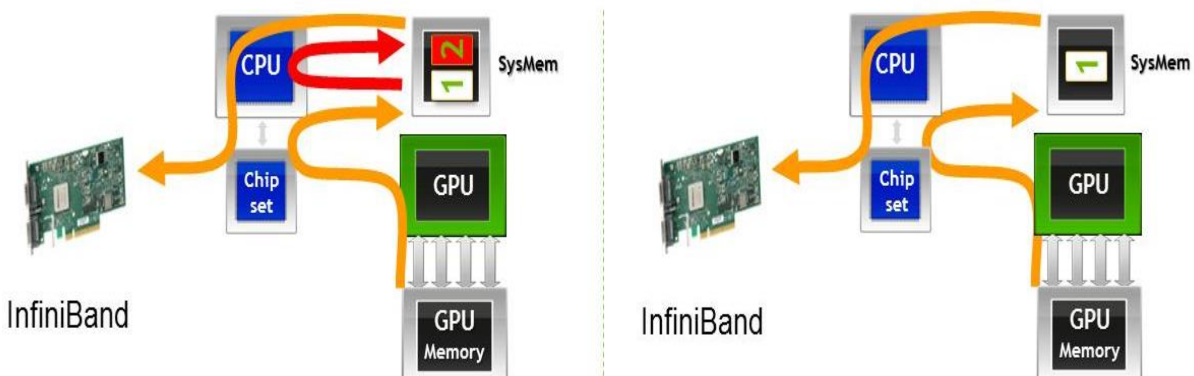
Same data copied three times

1. GPU write to pinned sysmem1
2. CPU copies from system1 to sysmem2
3. InfiniBand driver copies form sysmem2

With GPUDirect

Data only copied twice times

1. Sharing pinned system memory makes
2. System-to-system-copy unnecessary



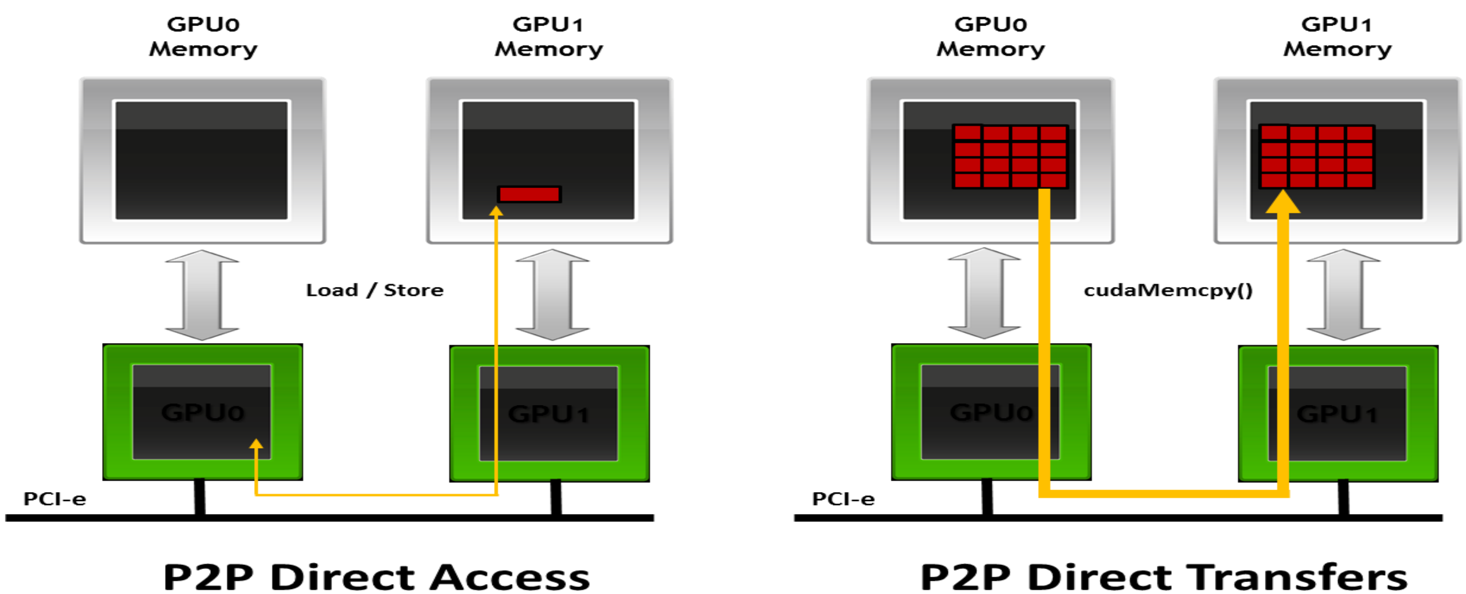
Source : http://www.nvidia.in/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

Source : NVIDIA, References

CUDA Tool Kit 5.0 Preview

❖ RDMA for GPUDirect : Features

NVIDIA GPUDirect Peer-to-Peer (P2P) Communication Between GPUs on the Same PCIe Bus : GPUDirect peer-to-peer transfers and memory access are supported natively by the CUDA Driver. All you need is CUDA Toolkit v4.0 and R270 drivers (or later) and a system with two or more Fermi-architecture GPUs on the same PCIe bus.



Source : http://www.nvidia.in/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

Source : NVIDIA, References

CUDA Tool Kit 4.0/5.0

- ❖ Share GPUs across multiple threads
- ❖ Use all GPUs in the system concurrently from a single host thread
- ❖ No-copy pinning of system memory, a faster alternative to cudaMallocHost()
- ❖ C++ new/delete and support for virtual functions
- ❖ Support for inline PTX assembly
- ❖ Thrust library of templated performance primitives such as sort, reduce, etc.
- ❖ NVIDIA Performance Primitives (NPP) library for image/video processing
- ❖ Layered Textures for working with same size/format textures at larger sizes and higher performance

Source : NVIDIA, References

CUDA Tool Kit 4.0/5.0

❖ **GPUDirect v2.0 : Features :**

- **GPUDirect v2.0 support for Peer-to-Peer Communication :** Accelerated communication with network and storage devices : Avoid unnecessary system memory copies and CPU overhead by copying data directly to/from pinned CUDA host memory
- **Peer-to-Peer Transfers between GPUs :** Use high-speed DMA transfers to copy data from one GPU directly to another GPU in the same system
- **Peer-to-Peer memory access :** Optimize communication between GPUs using NUMA-style access to memory on other GPUs from within CUDA kernels
- **GPUDirect for Video :** Optimized pipeline for frame-based devices such as frame grabbers, video switchers, HD-SDI capture, and CameraLink devices.

Source : NVIDIA, References

CUDA Tool Kit 4.0/5.0

CUDA Multi-GPU Programming : CUDA Programming model provides two basic approaches available to execute CUDA kernels on multiple GPUs (CUDA "devices") concurrently from a single host application:

- ❖ Use one host thread per device, since any given host thread can call `cudaSetDevice()` at most one time.
- ❖ Use the push/pop context functions provided by the CUDA Driver API.
- ❖ Unified Virtual Addressing (UVA) allows the system memory and the one or more device memories in a system to share a single virtual address space.

Source : NVIDIA, References

CUDA Tool KIT 4.0/5.0

CUDA Driver API : Features in which multiple host threads to set a particular context current simultaneously using either `cuCtxSetCurrent()` or `cuCtxPushCurrent()`.

- ❖ Host threads can now share device memory allocations, streams, events, or any other per-context objects (as seen above).
- ❖ Concurrent kernel execution devices of compute capability 2.x is now possible across host threads, rather than just within a single host thread. Note that this requires the use of separate streams; unless streams are specified, the kernels will be executed sequentially on the device in the order they were launched
- ❖ Built on top of UVA, GPUDirect v2.0 provides for direct peer-to-peer communication among the multiple devices in a system and for native MPI transfers directly from device memory.

Source : NVIDIA, References

CUDA Tool Kit 4.0/5.0

Host-CPU – Device GPU CUDA Prog :

- ❖ The algorithm is designed in such a way that each CPU thread (Pthreads, OpenMP, MPI) to control a different GPU.
- ❖ Achieving this is straightforward if a program spawns as many lightweight threads as there are GPUs – one can derive GPU index from thread ID. For example, OpenMP thread ID can be readily used to select GPUs.
- ❖ MPI rank can be used to choose a GPU reliably as long as all MPI processes are launched on a single host node having GPU devices and host configuration of CUDA programming environment.

Source : NVIDIA, References

Fermi Performance : CUDA enabled NVIDIA GPU

Performance Fermi GPU : Device-CPU (NVIDIA)

- One Tesla C2050 (Fermi) with 3 GB memory; Clock Speed 1.15 GHz, CUDA 4.1 Toolkit
- Reported theoretical peak performance of the Fermi (C2050) is 515 Gflop/s in double precision (448 cores; 1.15 GHz; one instruction per cycle) and reported maximum achievable peak performance of DGEMM in Fermi up to 58% of that peak.
- The theoretical peak of the GTX280 is 936 Gflops/s in single precision (240 cores X 1.30 GHz X 3 instructions per cycle) and reported maximum achievable peak performance of DGEMM up to 40% of that peak.

Source & Acknowledgements : NVIDIA, References

CUDA Tool Kit 4.0/5.0 Libraries

- **cuBLAS :** The NVIDIA CUDA Basic Linear Algebra Subroutines (cuBLAS) library is a GPU-accelerated version of the complete standard

- **cuFFT :** The NVIDIA CUDA Fast Fourier Transform library (cuFFT) provides a simple interface for computing FFTs up to 10x faster.

- **cuRAND :** The NVIDIA CUDA Random Number Generation library (cuRAND) delivers high performance GPU-accelerated random number generation (RNG).

- **cuSPARSE :** The NVIDIA CUDA Sparse Matrix library (cuSPARSE) provides a collection of basic linear algebra subroutines used for sparse matrices

Source : NVIDIA, References

CUDA Tool Kit 4.0/5.0 Libraries

- **NPP :** NVIDIA Performance Primitives : The NVIDIA Performance Primitives library (NPP) is a collection of GPU-accelerated image, video, and signal processing functions

- **Thurst :** Thrust is a powerful library of parallel algorithms and data structures. Thrust provides a flexible, high-level interface for GPU programming that greatly enhances developer productivity.

- **NVIDIA Visual Profiler :** The NVIDIA Visual Profiler is a cross-platform performance profiling tool that delivers developers vital feedback for optimizing CUDA C/C++ and OpenCL applications.

Source & Acknowledgements : NVIDIA, References

CUDA Tool Kit 4.0/5.0 Libraries

- **CUDA-GDB debuggers :**CUDA-GDB debuggers : CUDA-GDB supports debugging of both 32 and 64-bit CUDA C/C++ applications.

- **CUDA-MEMCHECK :** CUDA-MEMCHECK detects these errors in your GPU code and allows you to locate them quickly.

- **MAGMA :** MAGMA is a collection of next generation, GPU accelerated ,linear algebra libraries. Designed for heterogeneous GPU-based architectures. It supports interfaces to current LAPACK and BLAS standards.

Source & Acknowledgements : NVIDIA, References

Part-II(F)

An Overview of CUDA enabled NVIDIA GPUs:

Kepler / Results

(*) = Speedup results were gathered using untuned and unoptimized versions of benchmarks & NVIDIA CUDA Prog. Env - This is C-DAC In-house HPC GPU Cluster project work in collaboration with NVIDIA

http://www.nvidia.com; NVIDIA CUDA

CUDA Tool Kit 5.0 Preview

- **Nsight Eclipse Edition** : Develop & Debug and Profile GPU Accelerated Applications on Linux - **All in on IDE**
- **RDMA for GPUDirect :** Direct Communication between GPUs and other PCIe Devices
- **GPU Library Object Linking :** Easily Accelerate parallel nested loops starting with Tesla K20 Kepler GPUs
- **Dynamic Parallelism :** library of templated performance primitives such as sort, reduce, etc.
- NVIDIA Performance Primitives (NPP) library for image/video processing
- Layered Textures for working with same size/format textures at larger sizes and higher performance

Source : NVIDIA, References

CUDA Tool Kit 5.0 Preview

❖ **RDMA for GPUDirect : Features**

- **Accelerated communication with network and storage devices :** Avoid unnecessary system memory copies and CPU overhead by copying data directly to/from pinned CUDA host memory
- **Peer-to-Peer Transfers between GPUs :** Use high-speed DMA transfers to copy data from one GPU directly to another GPU in the same system
- **Peer-to-Peer memory access :** Optimize communication between GPUs using NUMA-style access to memory on other GPUs from within CUDA kernels

Source : NVIDIA, References

CUDA Tool Kit 5.0 Preview

❖ RDMA for GPUDirect : Features

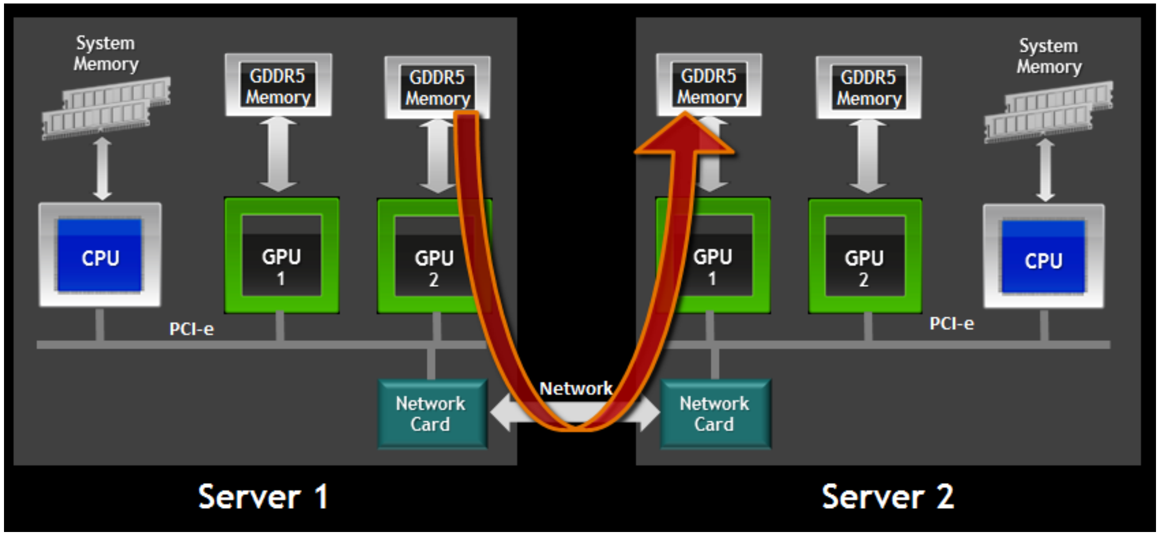
- **Peer-to-Peer memory access :** Optimize communication between GPUs using NUMA-style access to memory on other GPUs from within CUDA kernels
- **RDMA :** Eliminate CPU bandwidth and latency bottlenecks using direct memory access (DMA) between GPUs and other PCIe devices, resulting in significantly improved MPISendRecv efficiency between GPUs and other nodes (new in CUDA 5)
- **GPUDirect for Video :** Optimized pipeline for frame-based devices such as frame grabbers, video switchers, HD-SDI capture, and CameraLink devices.

Source : NVIDIA, References

CUDA Tool Kit 5.0 Preview

❖ **RDMA for GPUDirect : Features**

GPUDirect™ Support for RDMA, Introduced with CUDA 5



Eliminate CPU bandwidth and latency bottlenecks using direct memory access (DMA) between GPUs and other **PCIe devices**, resulting in significantly improved **MPISendRecv** efficiency between GPUs and other nodes (new in CUDA 5)

Source : http://www.nvidia.in/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

Source : NVIDIA, References

CUDA Tool Kit 5.0 Preview

❖ RDMA for GPUDirect : Features

GPUDirect™ Support for Accelerated Communication with Network and Storage Devices

Without GPUDirect

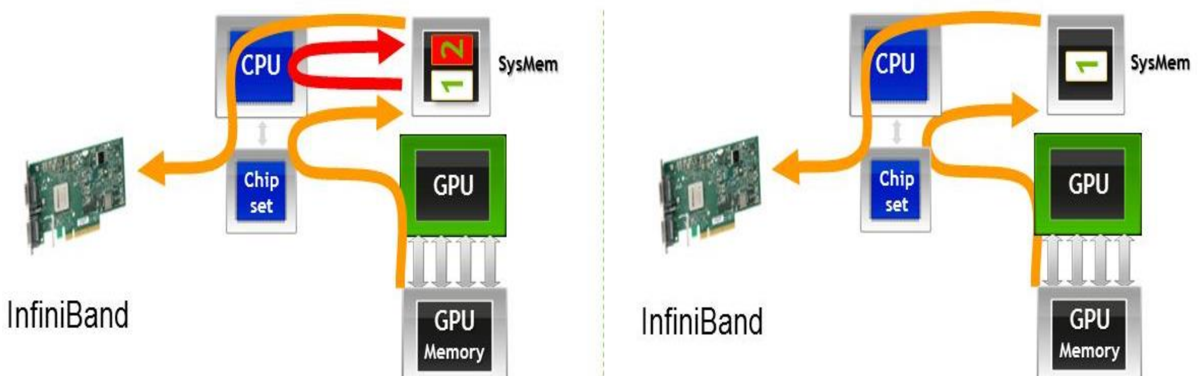
Same data copied three times

1. GPU write to pinned sysmem1
2. CPU copies from system1 to sysmem2
3. InfiniBand driver copies form sysmem2

With GPUDirect

Data only copied twice times

1. Sharing pinned system memory makes
2. System-to-system-copy unnecessary



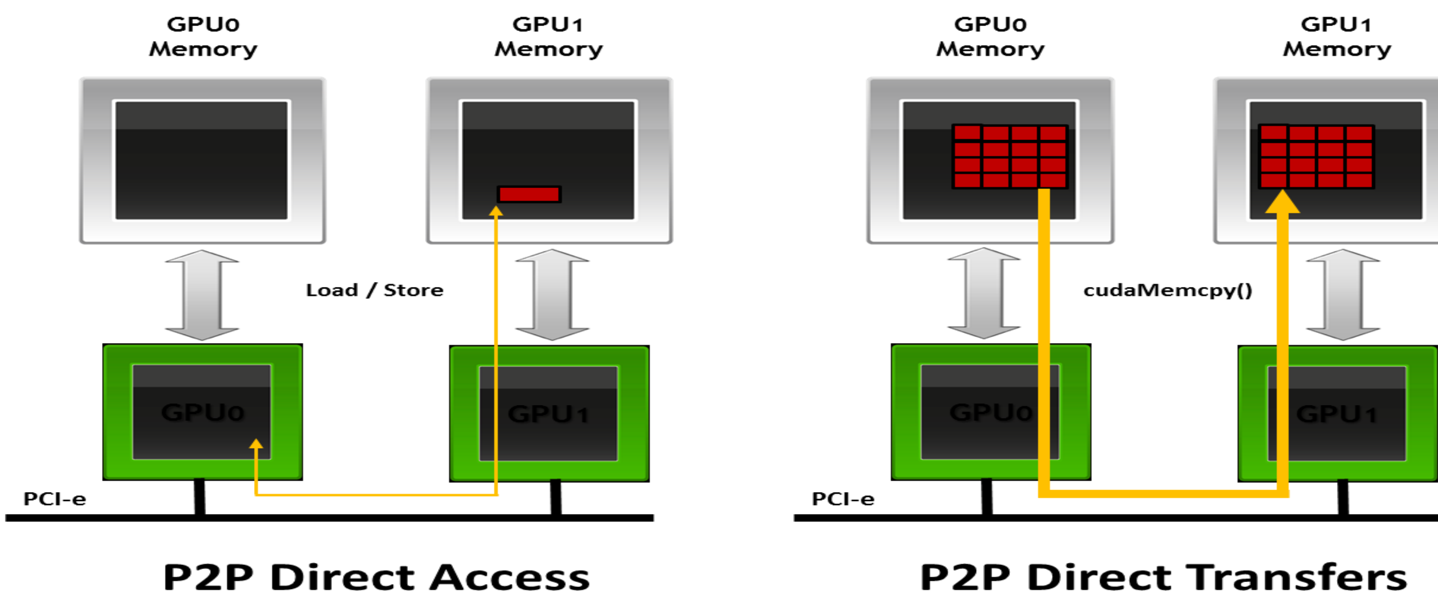
Source : http://www.nvidia.in/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

Source : NVIDIA, References

CUDA Tool Kit 5.0 Preview

❖ RDMA for GPUDirect : Features

NVIDIA GPUDirect Peer-to-Peer (P2P) Communication Between GPUs on the Same PCIe Bus : GPUDirect peer-to-peer transfers and memory access are supported natively by the CUDA Driver. All you need is CUDA Toolkit v4.0 and R270 drivers (or later) and a system with two or more Fermi-architecture GPUs on the same PCIe bus.



Source : http://www.nvidia.in/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

Source : NVIDIA, References

CUDA Tool Kit 4.0/5.0

- ❖ Share GPUs across multiple threads
- ❖ Use all GPUs in the system concurrently from a single host thread
- ❖ No-copy pinning of system memory, a faster alternative to cudaMallocHost()
- ❖ C++ new/delete and support for virtual functions
- ❖ Support for inline PTX assembly
- ❖ Thrust library of templated performance primitives such as sort, reduce, etc.
- ❖ NVIDIA Performance Primitives (NPP) library for image/video processing
- ❖ Layered Textures for working with same size/format textures at larger sizes and higher performance

Source : NVIDIA, References

CUDA Tool Kit 4.0/5.0

❖ **GPUDirect v2.0 : Features :**

- **GPUDirect v2.0 support for Peer-to-Peer Communication :** Accelerated communication with network and storage devices : Avoid unnecessary system memory copies and CPU overhead by copying data directly to/from pinned CUDA host memory

- **Peer-to-Peer Transfers between GPUs :** Use high-speed DMA transfers to copy data from one GPU directly to another GPU in the same system

- **Peer-to-Peer memory access :** Optimize communication between GPUs using NUMA-style access to memory on other GPUs from within CUDA kernels

- **GPUDirect for Video :** Optimized pipeline for frame-based devices such as frame grabbers, video switchers, HD-SDI capture, and CameraLink devices.

Source : NVIDIA, References

CUDA Tool Kit 4.0/5.0

CUDA Multi-GPU Programming : CUDA Programming model provides two basic approaches available to execute CUDA kernels on multiple GPUs (CUDA “devices”) concurrently from a single host application:

- ❖ Use one host thread per device, since any given host thread can call `cudaSetDevice()` at most one time.
- ❖ Use the push/pop context functions provided by the CUDA Driver API.
- ❖ Unified Virtual Addressing (UVA) allows the system memory and the one or more device memories in a system to share a single virtual address space.

Source : NVIDIA, References

CUDA Tool KIT 4.0/5.0

CUDA Driver API : Features in which multiple host threads to set a particular context current simultaneously using either `cuCtxSetCurrent()` or `cuCtxPushCurrent()`.

- ❖ Host threads can now share device memory allocations, streams, events, or any other per-context objects (as seen above).
- ❖ Concurrent kernel execution devices of compute capability 2.x is now possible across host threads, rather than just within a single host thread. Note that this requires the use of separate streams; unless streams are specified, the kernels will be executed sequentially on the device in the order they were launched
- ❖ Built on top of UVA, GPUDirect v2.0 provides for direct peer-to-peer communication among the multiple devices in a system and for native MPI transfers directly from device memory.

Source : NVIDIA, References

CUDA Tool Kit 4.0/5.0

Host-CPU – Device GPU CUDA Prog :

- ❖ The algorithm is designed in such a way that each CPU thread (Pthreads, OpenMP, MPI) to control a different GPU.
- ❖ Achieving this is straightforward if a program spawns as many lightweight threads as there are GPUs – one can derive GPU index from thread ID. For example, OpenMP thread ID can be readily used to select GPUs.
- ❖ MPI rank can be used to choose a GPU reliably as long as all MPI processes are launched on a single host node having GPU devices and host configuration of CUDA programming environment.

Source : NVIDIA, References

Fermi Performance : CUDA enabled NVIDIA GPU

Performance Fermi GPU : Device-CPU (NVIDIA)

- One Tesla C2050 (Fermi) with 3 GB memory; Clock Speed 1.15 GHz, CUDA 4.1 Toolkit
- Reported theoretical peak performance of the Fermi (C2050) is 515 Gflop/s in double precision (448 cores; 1.15 GHz; one instruction per cycle) and reported maximum achievable peak performance of DGEMM in Fermi up to 58% of that peak.
- The theoretical peak of the GTX280 is 936 Gflops/s in single precision (240 cores X 1.30 GHz X 3 instructions per cycle) and reported maximum achievable peak performance of DGEMM up to 40% of that peak.

Source & Acknowledgements : NVIDIA, References

CUDA Tool Kit 4.0/5.0 Libraries

- **cuBLAS :** The NVIDIA CUDA Basic Linear Algebra Subroutines (cuBLAS) library is a GPU-accelerated version of the complete standard
- **cuFFT :** The NVIDIA CUDA Fast Fourier Transform library (cuFFT) provides a simple interface for computing FFTs up to 10x faster.
- **cuRAND :** The NVIDIA CUDA Random Number Generation library (cuRAND) delivers high performance GPU-accelerated random number generation (RNG).
- **cuSPARSE :** The NVIDIA CUDA Sparse Matrix library (cuSPARSE) provides a collection of basic linear algebra subroutines used for sparse matrices

Source : NVIDIA, References

CUDA Tool Kit 4.0/5.0 Libraries

- **NPP :** NVIDIA Performance Primitives : The NVIDIA Performance Primitives library (NPP) is a collection of GPU-accelerated image, video, and signal processing functions

- **Thurst :** Thrust is a powerful library of parallel algorithms and data structures. Thrust provides a flexible, high-level interface for GPU programming that greatly enhances developer productivity.

- **NVIDIA Visual Profiler :** The NVIDIA Visual Profiler is a cross-platform performance profiling tool that delivers developers vital feedback for optimizing CUDA C/C++ and OpenCL applications.

Source & Acknowledgements : NVIDIA, References

CUDA Tool Kit 4.0/5.0 Libraries

- **CUDA-GDB debuggers :**CUDA-GDB debuggers : CUDA-GDB supports debugging of both 32 and 64-bit CUDA C/C++ applications.

- **CUDA-MEMCHECK :** CUDA-MEMCHECK detects these errors in your GPU code and allows you to locate them quickly.

- **MAGMA :** MAGMA is a collection of next generation, GPU accelerated ,linear algebra libraries. Designed for heterogeneous GPU-based architectures. It supports interfaces to current LAPACK and BLAS standards.

Source & Acknowledgements : NVIDIA, References

NVIDIA's Next Generation CUDA : Kepler

❖ Kepler GK10:

- **Dynamic Parallelism** : adds the capability for the GPU to generate new work for itself, synchronize on results, and control the scheduling of that work via dedicated, accelerated hardware paths, all without involving the CPU.

- **Hyper-Q :** Hyper-Q enables multiple CPU cores to launch work on a single GPU simultaneously, thereby dramatically increasing GPU utilization and significantly reducing CPU idle times

Source : http://www.nvidia.in/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

Source & Acknowledgements : NVIDIA, References

NVIDIA's Next Generation CUDA : Kepler

❖ Kepler GK10:

- **Dynamic Parallelism** : adds the capability for the GPU to generate new work for itself, synchronize on results, and control the scheduling of that work via dedicated, accelerated hardware paths, all without involving the CPU.

- **Hyper-Q :** Hyper-Q enables multiple CPU cores to launch work on a single GPU simultaneously, thereby dramatically increasing GPU utilization and significantly reducing CPU idle times

Source : http://www.nvidia.in/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

Source & Acknowledgements : NVIDIA, References

❖ Kepler GK10:

- **Grid Management Unit :** Enabling Dynamic Parallelism requires an advanced, flexible grid management and dispatch control system. The new GK110 Grid Management Unit (GMU) manages and prioritizes grids to be executed on the GPU. The GMU can pause the dispatch of new grids and queue pending and suspended grids until they are ready to execute, providing the flexibility to enable powerful runtimes, such as Dynamic Parallelism. The GMU ensures both CPU- and GPU-generated workloads are properly managed and dispatched.

Source : http://www.nvidia.in/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

Source & Acknowledgements : NVIDIA, References

❖ Kepler GK10:

- **Grid Management Unit :** Enabling Dynamic Parallelism requires an advanced, flexible grid management and dispatch control system. The new GK110 Grid Management Unit (GMU) manages and prioritizes grids to be executed on the GPU. The GMU can pause the dispatch of new grids and queue pending and suspended grids until they are ready to execute, providing the flexibility to enable powerful runtimes, such as Dynamic Parallelism. The GMU ensures both CPU- and GPU-generated workloads are properly managed and dispatched.

Source : http://www.nvidia.in/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

Source & Acknowledgements : NVIDIA, References

NVIDIA's Next Generation CUDA : Kepler

❖ Kepler GK10:

- **GPUDirect :** NVIDIA GPUDirect™ is a capability that enables GPUs within a single computer, or GPUs in different servers located across a network, to directly exchange data without needing to go to CPU/system memory. The RDMA feature in GPUDirect allows third party devices such as SSDs, NICs, and IB adapters to directly access memory on multiple GPUs within the same system, significantly decreasing the latency of MPI send and receive messages to/from GPU memory

Source : http://www.nvidia.in/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

Source & Acknowledgements : NVIDIA, References

GPU -CUDA enabled NVIDIA GPU

❖Tesla C 2075

➢ Peak Double Precision Floating Point Performance	515 Gflops
➢ Peak Single precision floating Performance	1030 Gflops
➢ Memory Bandwidth (ECC off)	148 GBytes/s
➢ Memory Size (GDDr5)	6 GB
➢ CUDA Cores	448 Cores

GPU -CUDA enabled NVIDIA GPU

❖Sustainability of Memory Bandwidth
Main Memory Access Efficiency

- ❖ Each floating point operates on upto 12-16 bytes of source data, the available memory bandwidth cannot sustain even a small fraction of the peak performance is all the source data are accessed from global memory
  - ➢ To address above, CUDA & underlying GPUs offer multiple memory types with different bandwidths & latencies

GPU -CUDA enabled NVIDIA GPU

❖ Sustainability of Memory Bandwidth

Main Memory Access Efficiency

- CUDA & underlying GPUs offer multiple memory types with different bandwidths & latencies
- CUDA memory types have access restrictions to allow programmers to conserve memory bandwidth while increasing the overall performance of applications.

GPU -CUDA enabled NVIDIA GPU

❖ Sustainability of Memory Bandwidth

Main Memory Access Efficiency

- CUDA Programmers are responsible for explicitly allocating space and managing data movement among the different memories to conserve memory bandwidth

- CUDA Programmers shoulders the responsibility of massaging the code to produce the desirable access patterns

- CUDA code should explicitly optimize for GPU's memory hierarchy.

GPU -CUDA enabled NVIDIA GPU

❖ Sustainability of Memory Bandwidth

Main Memory Access Efficiency

- CUDA Provides additional hardware mechanisms at the memory interface can enhance the main memory access efficiency if the access patterns follow **memory coalescing rules.**

General CUDA Program Format

CUDA – Compute Unified Device Architecture

- Step 1 – copy data from main memory to GPU global memory (from host to device)
- Step 2 – threads run code inside kernel function
  - Each thread fetches some data from global memory and stores it in registers
  - Each thread performs computations
  - Each thread stores a result in global memory
- Step 3 – copy results from device back to host

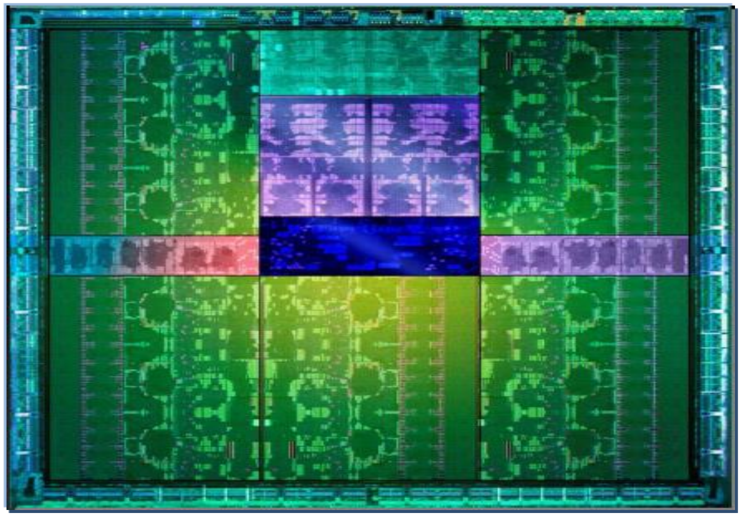
Kepler GK110-the new CUDA Compute Capability 5.0

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Threads Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessors	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Max Threads / Thread Block	1024	1024	1024	1024
Shared Memory Size Configuration (bytes)	16K 48K	16K 48K	16K 32K 48K	16K 32K 48K
Max X Grid Dimension	2^16-1	2^16-1	2^32-1	2^32-1
Hyper-Q	No	No	No	Yes
Dynamic Parallelism	No	No	No	Yes

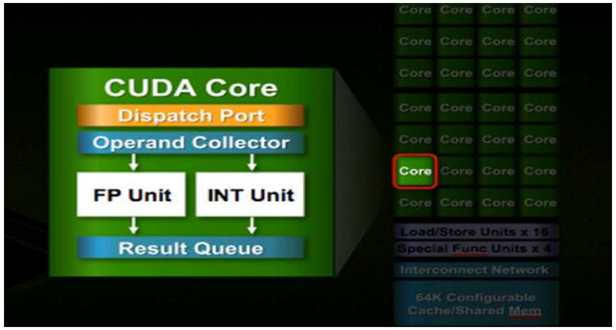
GTX 470/480s have GT100s

C2050s on grid06 and grid07 are compute cap 2.0

GPU Computing – NVIDIA KEPLER GPUs



Source : http://www..nvidia.com



Kepler GK110 supports the new CUDA Compute Capability 5.0

Features	Tesla K20X	Tesla K20 (Kepler GK110)
Peak double Precision Floating Point Performance	1.31 Tflops	1.17 Tflops
Peak Single Precision Floating Performance	3.95 Tflops	3.52 Tflops
Memory Bandwidth (ECC off)	250 GB/s	208.8 B/s
Memory size (GDDR5)	6 GB	5 GB
CUDA Cores	2688	2496

GTX 470/480s have GT100s

C2050s on grid06 and grid07 are compute cap 2.0

NVIDIA GPU Prog. Models

- Current: Cuda 4.1
  - Share GPUs across multiple threads
  - Unified Virtual Addressing
  - Use all GPUs from a single host thread
  - Peer-to-Peer communication
- Coming in Cuda 5
  - Direct communication between GPUs and other PCI devices
  - Easily acceleratable parallel nested loops starting with Tesla K20 Kepler GPU
- Current: OpenCL 1.2
  - Open royalty-free standard for cross-platform parallel computing
  - Latest version released in November 2011
  - Host-thread safety, enabling OpenCL commands to be enqued from multiple host threads
  - Improved OpenGL interoperability by linking OpenCL event objects to OpenGL
- OpenACC
  - Programming standard developed by Cray, NVIDIA, CAPS and PGI
  - Designed to simplify parallel programming of heterogeneous CPU/GPU systems
  - The programming is done through some pragmas and API functions
  - Planned supported compilers – Cray, PGI and CAPS

Kepler Architectural Overview

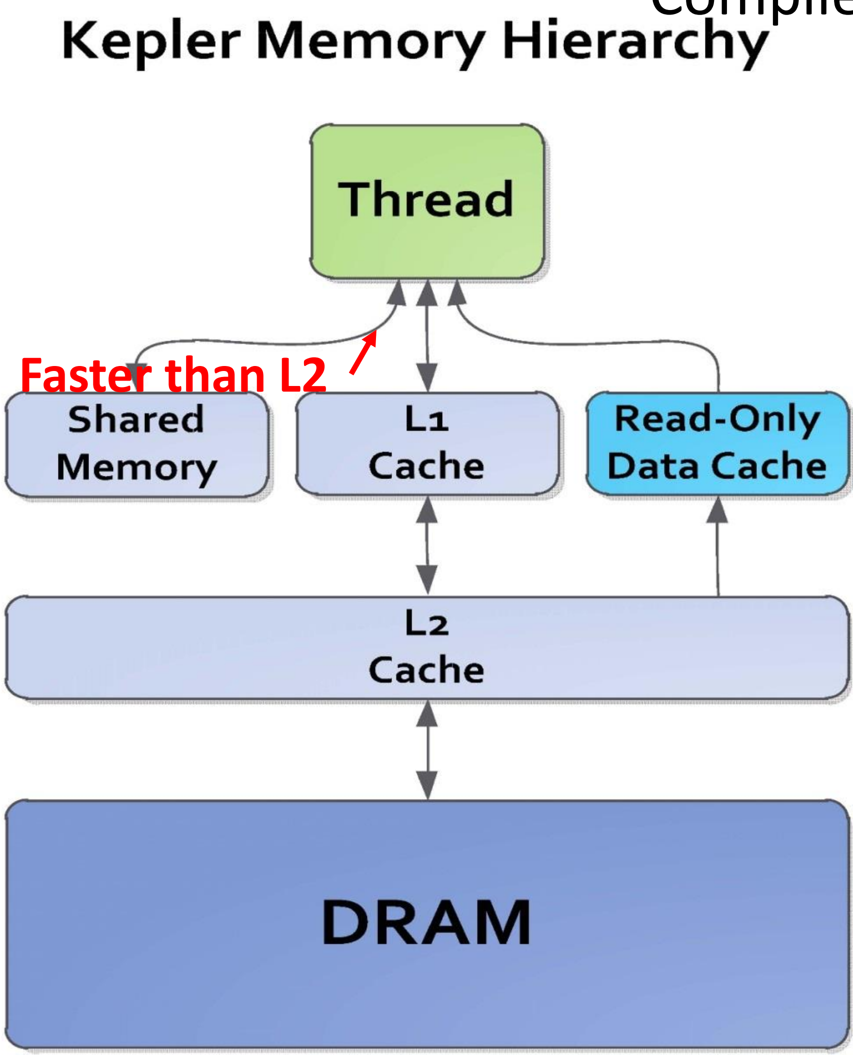
- A full k110 implementation includes 15 SMX units and six 64-bit memory controllers. Different products will use different configurations of K110.

Key features ...

- The new SMX processor architecture
- An enhanced memory subsystem, offering additional caching capabilities, more bandwidth at each level of the hierarchy and a fully redesigned and substantially faster DRAM I/O implementation.

Kepler Memory Subsystem

New: 48 KB Read-only memory cache
Compiler/programmer can use to advantage



Shared memory/L1 cache split:
Each SMX has 64 KB on-chip memory, that can be configured as:

- 48 KB of Shared memory with 16 KB of L1 cache,

or

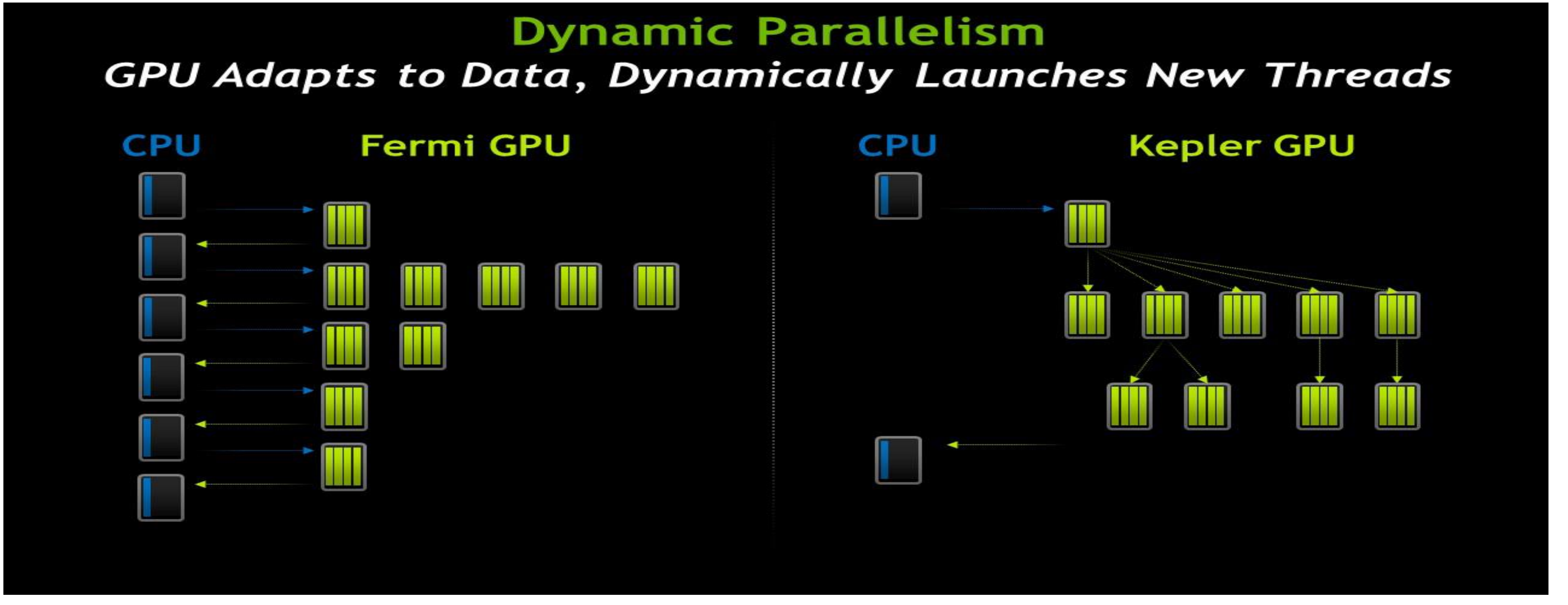
- 16 KB of shared memory with 48 KB of L1 cache

or

- (new) a 32KB / 32KB split between shared memory and L1 cache.

Kepler Dynamic Parallelism

"Dynamic Parallelism allows more parallel code in an application to be launched directly by the GPU onto itself (right side of image) rather than requiring CPU intervention (left side of image)."



NVIDIA – Application Kernels

http://www.nvidia.com

http://www.nvidia.com; NVIDIA CUDA

(*) = Speedup results were gathered using untuned & unoptimized versions of benchmark and NVIDIA Prog. Env on NVIDIA Fermi /Kepler

Present Work : Application Kernels On Hybrid Computing Systems (HPC GPU Cluster)

Results : LINPACK (Top-500) Kepler

Total (CPU+GPU) Peak Performance : 1267 Gflops

CPU Peak Performance (DP) : 96 Gflops (1 Node – 8 Cores)

GPU Peak Performance (DP) : 1170 Gflops (1.17 Tflops)

Nodes/GPUs		LINPACK						Gflops
Nodes	GPUs	T/V	N	NB	P	Q	Time	
1	1	WR10L2L2	34560	768	1	1	100.21	764.4
1	1	WR10L2L2	44968	768	1	1	187.71	785.5

62.13% sustained performance of Top-500 LINPACK is achieved

(*) = Speedup results were gathered using untuned and unoptimized versions of benchmarks & NVIDIA CUDA Prog. Env - This is C-DAC In-house HPC GPU Cluster project work

Present Work : Application Kernels On Hybrid Computing Systems (HPC GPU Cluster)

Results : MAGMA (Open Source Software : NLA) Fermi

Total (CPU+GPU) Peak Performance : 611 Gflops

CPU Peak Performance (DP) : 96 Gflops (1 Node – 8 Cores)

GPU Peak Performance (DP) : 515 Gflops

Node	Library	Routine Used	Matrix Size	Sustained Performance in Gflops
1	MAGMA	DGEMM	10240	302.81
1	CUBLAS	DGEMM	10240	302.75
1	MAGMA	DGETRF	5952	219.31
1		DGETRF	9984	256.29

Intel MKL version 10.2, CUBLAS version 3.2, Intel icc11.1

The routines such as DGETRF (LU factorization of certain class of matrices) show goof performance.

The MAGAMA uses LAPACK, CUDA BLAS, and MAGMA BLAS routines for factorization (LU, QR & Cholesky) of matrices

(*) = Speedup results were gathered using untuned and unoptimized versions of benchmarks & NVIDIA CUDA Prog. Env - This is C-DAC In-house HPC GPU Cluster project work

Present Work : Application Kernels On Hybrid Computing Systems (HPC GPU Cluster)

Results : Jacobi Iterative Method (Fermi)

Total (CPU+GPU) Peak Performance : 611 Gflops

CPU Peak Performance (DP) : 96 Gflops (1 Node – 8 Cores)

GPU Peak Performance (DP) : 515 Gflops

Jacobi Iterative Method : To solve system of dense matrix system of linear equations **[A] {x}= {b}**

Time Taken in Seconds		
Matrix Size	**CUDA API**	**CUBLAS**
1024	1.6439	0.0525
2048	5.4248	0.0972
4096	26.3400	0.2299
8092	87.768	0.7138

(*) = Speedup results were gathered using untuned and unoptimized versions of benchmarks & NVIDIA CUDA Prog. Env - This is C-DAC In-house HPC GPU Cluster project work

C-DAC HPC GPU Cluster : Benchmarks

GPU : Kepler

Results : Total (CPU+GPU) Peak Performance : 1267 Gflops

CPU Peak Performance (DP) : 96 Gflops (1 Node – 8 Cores)

GPU Peak Performance (DP) : 1170 Gflops (1.17 Tflops)

Experiment Results for LINPACK(*) : without any Optimizations

62.13% is sustained performance of LINPACK can be achieved for appropriate matrix sizes i.e., N= 48000 ~ 64000. Further Optimization may improve by10% to 15 %

Visit http://www.nvidia.com

(*=In collaboration with NVIDIA)

(*) = Speedup results were gathered using untuned and unoptimized versions of benchmark and NVIDIA Prog. Env on NVIDIA KEPLER

C-DAC HPC GPU Cluster : Benchmarks

Node = Fermi

Total (CPU+GPU) Peak Performance : 611 Gflops

CPU Peak Performance (DP) : 96 Gflops (1 Node – 8 Cores)

GPU Peak Performance (DP) : 515 Gflops

Experiment Results for DGEMM : Without any Optimizations 60.0% is sustained performance of CUDA (CUBLAS) can be achieved for appropriate matrix sizes i.e., N= 10000 ~ 16000. Further Optimization may improve by10% to 15 %

Visit http://www.nvidia.com

(*=In collaboration with NVIDIA)

(*) = Speedup results were gathered using untuned and unoptimized versions of benchmark and NVIDIA Prog. Env on NVIDIA Kepler

Present Work : Application Kernels On Hybrid Computing Systems (HPC GPU Cluster)

Results : Conjugate Gradient Method
Total (CPU+GPU) Peak Performance : 611 Gflops
CPU Peak Performance (DP) : 96 Gflops (1 Node – 8 Cores)
GPU Peak Performance (DP) : 515 Gflops
Conjugate Gradient Method : To solve system of dense matrix system of linear equations **[A] {x}= {b}**

Time Taken in Seconds		
Matrix Size	**CUDA API**	**CUBLAS**
1024	0.5186	0.0296
2048	1.881	0.0740
4096	8.677	0.2214
8092	33.376	0.7893

(*) = Speedup results were gathered using untuned and unoptimized versions of benchmarks & NVIDIA CUDA Prog. Env - This is C-DAC In-house HPC GPU Cluster project work

Present Work : Application Kernels On Hybrid Computing Systems (HPC GPU Cluster)

Results for DGEMM (CPU+GPU) : In-house (Fermi)

Total (CPU+GPU) Peak Performance : 611 Gflops

CPU Peak Performance (DP) : 96 Gflops (1 Node – 8 Cores)

GPU Peak Performance (DP) : 515 Gflops

Nodes	GPUs	Matrix Size (CPU + GPU)	Sustained Perf in Gflops Total (CPU +GPU)	Utilization (%)
1	1	1024	181.25	29.66
1	1	4096	326.73	53.47
1	1	10240	363.47(*)	59.49
1	1	12288	366.42(*)	59.47

Intel MKL version 10.2, CUBLAS version 3.2, Intel icc11.1

(* = relative error exists). 60% sustained performance of is achieved

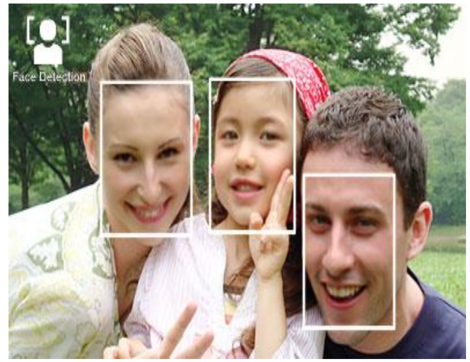
(*) = Speedup results were gathered using untuned and unoptimized versions of benchmarks & NVIDIA CUDA Prog. Env - This is C-DAC In-house HPC GPU Cluster project work

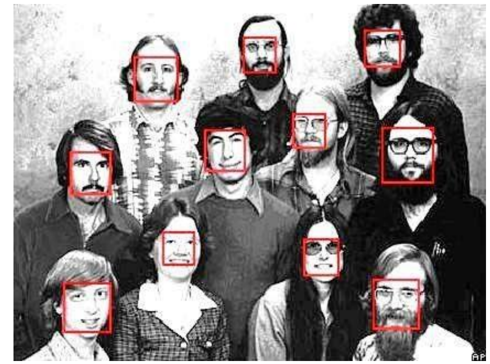
Application : Image Processing – Multi-Core – Many-Core Implementation

MPI – CUDA - GPU Implementation of Face Detection(*)

Using pre trained Haar - classifier and integral image on GPU cluster

Image size	GPU (Fermi) time(sec)	GPU time (sec)
	512 threads/ block	8 threads/ block
132*184	0.000620	0.000285
700*500	0.003376	0.001120
1289*649	0.005940	0.002531





Courtesy : Viola and Jones

Courtesy : C-DAC Intrnal Projects

(*) = Speedup results were gathered using untuned and unoptimized versions of benchmark and NVIDIA Prog. Env on NVIDIA Fermi

- Four kinds of Haar features are used in detection algorithm. Trained cascaded classifiers are obtained, apply these classifiers to detect images
- Parallelize the detection process by mapping each window to a thread for face detection.

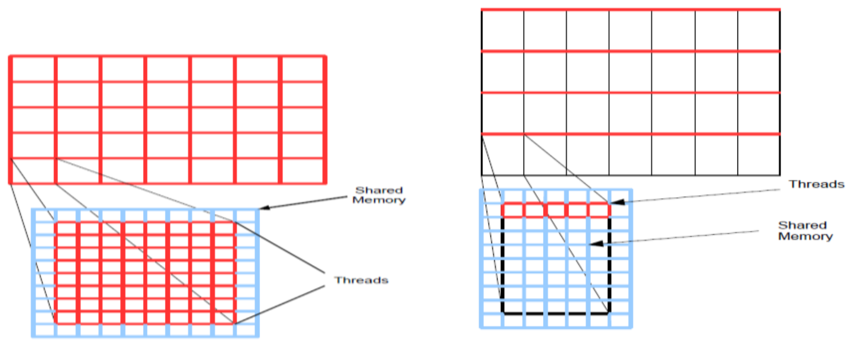
Courtesy : C-DAC Projects & Viola and Jones Alg.

(*) = Speedup results were gathered using untuned and unoptimized versions of benchmarks & NVIDIA CUDA Prog. Env - This is C-DAC In-house HPC GPU Cluster project work

Application : Image Processing – Multi-Core – Many-Core Implementation

MPI – CUDA - GPU Implementation of Edge Detection

- Each thread within the thread block corresponds to a single pixel or Multiple pixels within the image



Courtesy : Viola and Jones

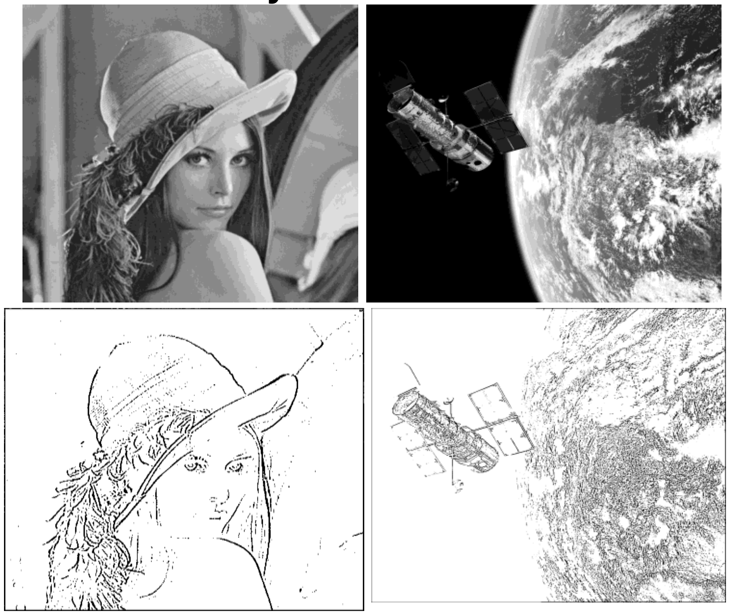
Edge Detection : Canny Edge Detection (*)

Pixels	OpenCV (Time in *ms*)	CUDA - GPU optimized Block Size of 8 x 8 (*Time in ms*)
512 x 512	8.40	0.62
1024 x 1024	28.01	2.30
2048 x 2048	108.52	9.34
4096 x 4096	398.14	38.17

Edge Detection : Laplace Edge Detection (*)

Pixels	OpenCV (Time in *ms*)	MPI (No. of PEs) (Time in *ms*)		CUDA-GPU Block Size of 16 x 16 (*Time in ms*)	
		2	8	UnOptimised	Optimized
512 x 512	2.91	6.91	2.93	0.39	0.21
1024 x 1024	11.01	27.41	13.87	1.53	0.709
2048 x 2048	42.74	112.25	42.05	5.998	2.780
4096 x 4096	173.39	449.97	159.89	23.86	11.27

(*) = Speedup results were gathered using untuned & unoptimized versions of benchmark and NVIDIA Prog. Env on NVIDIA Fermi



Courtesy : C-DAC Projects & Wikipedia

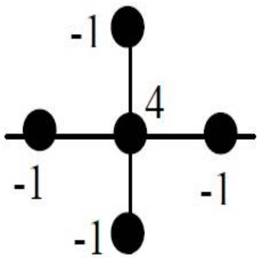
512*512 1024*1024

Application : FDM/FEM Computations (Structured/ Unstructured Grids) - HPC GPU Cluster

Poisson & Parabolic Eq. Solver

$$\frac{\partial U}{\partial t} - \frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} + \frac{\partial^2 U}{\partial z^2} = f(x,y,z);\ \Omega \subseteq \Re^3;\ t \in [t_o, t_f]$$

$$U(x,y,z,t_0) = g \text{ on } \partial\Omega$$

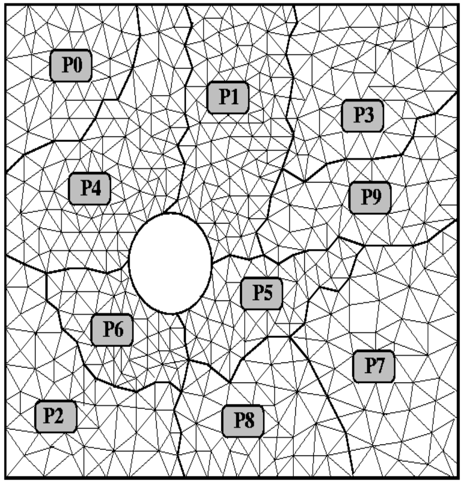


Data Re-arrangement Kernels & Jacobi / CG Methods

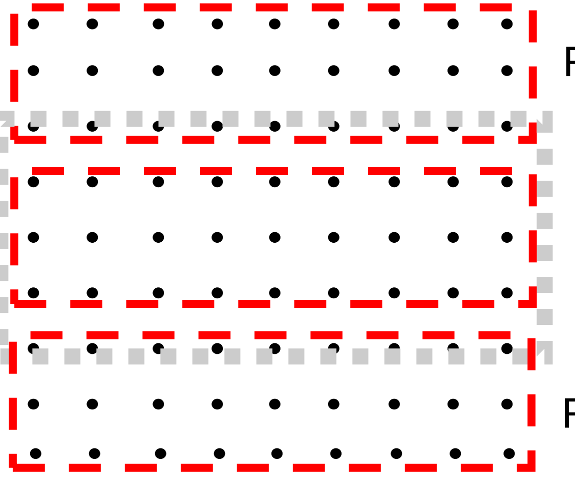
FEM

Graph Partition Software **METIS**

Each Partition **mapped** to each **GPU**

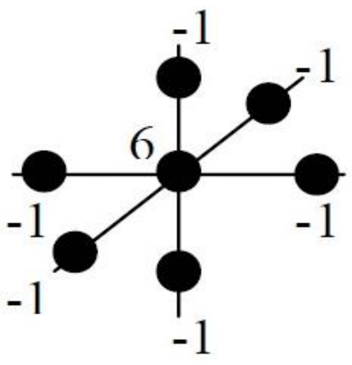


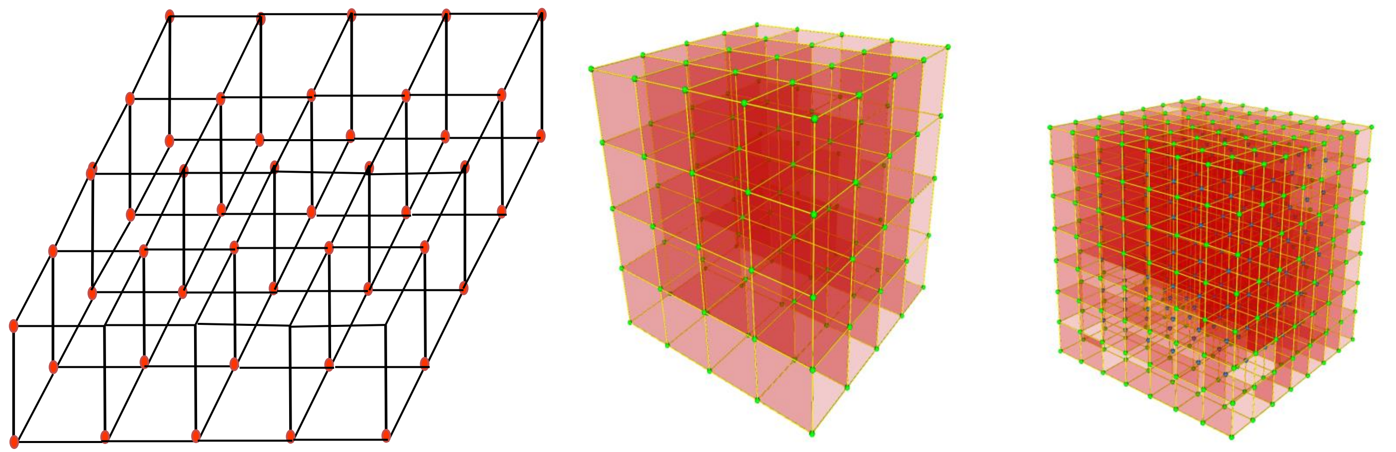
FDM



Courtesy : C-DAC HPC-FTE Student Projects

Application : FDM/FEM Computations (Structured/ Unstructured Grids) - HPC GPU Cluster

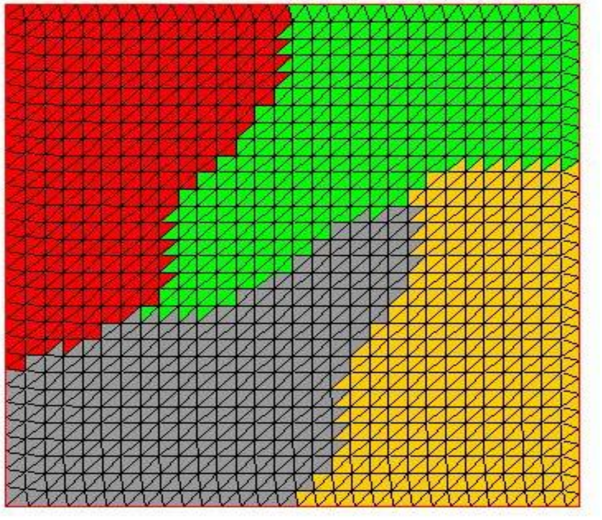




Stencil for Poisson Eq. in 3D

CUDA - Date Access Dominated, basic computation kernels, Generic Stencil Computations

CUDA - Data Re-arrangement Kernels – Coalesced Data access and Basic Read/Write routines Data Reordering routines



Courtesy : Chaman Singh Verma et**.** all; & Jall Open source software

Courtesy : C-DAC HPC-FTE Student Projects, 2011-2012

HPC GPU Cluster : Parallel Finite Difference Computations (Structured Grids)

Heat Transfer : GPU Implementation

❖ Access Pattern within a **32 X 32** block using **32 X 8** threads

- Blocking & Threading
- Use of Shared Memory
- Implicit Handling of Boundary Conditions - part of computations
- Tiling for Stencil Computations

❖ Performance 4x to 6x for un-optimised CUDA code

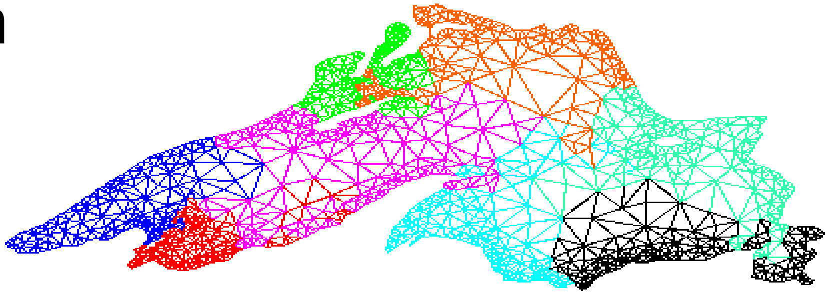
Type of Domain	Nodes/ (Partitions/ MPI Process)	Elapsed time (in seconds) MPI GPU Cluster		
		MPI	CUDA	OpenCL
2D-Structured grid -FDM (64X64)	4096 (1/1)	4.28		
	4096 (2/2)	3.12	0.82	1.28
2D-Structured grid -FDM (128X128)	16384 (1/1)	11.22		
	16384 (4/4)	3.74	0.98	1.42
3D-Structured grid -FDM (64X64X64)	262144 (1/1)	32.28		
	262144 (8/8)	6.64	1.31	2.23

Domain decomposition, with blocks of size - 32x32

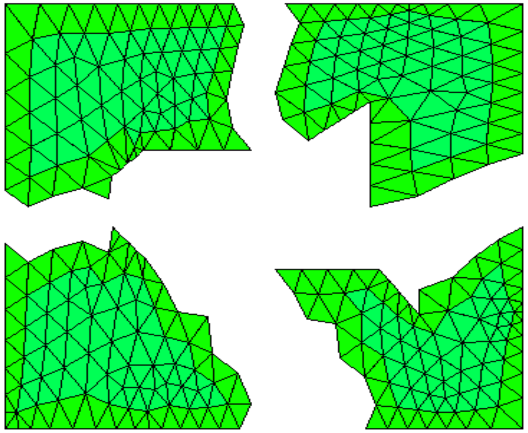
(*) = Speedup results were gathered using untuned and unoptimized versions of benchmarks & NVIDIA CUDA Prog. Env - This is C-DAC In-house HPC GPU Cluster project work

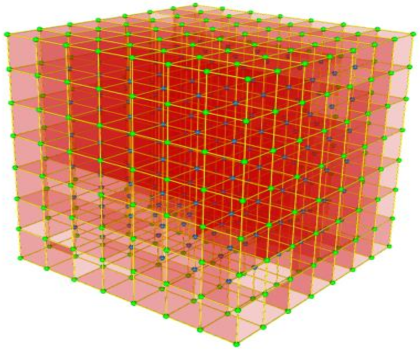
HPC GPU Cluster : Parallel Finite Element Method Comps. (Unstructured Grids)

Heat Transfer : GPU Implementation



- Access Pattern within a **32 X 32** block using **32 X 8** threads
  - Implicit Handling of Boundary Conditions - part of computations
  - Graph Partitioning for Mesh Computations
  - Graph Coloring for solver on a single node





- Performance 4x to 6x for un-optimised CUDA code

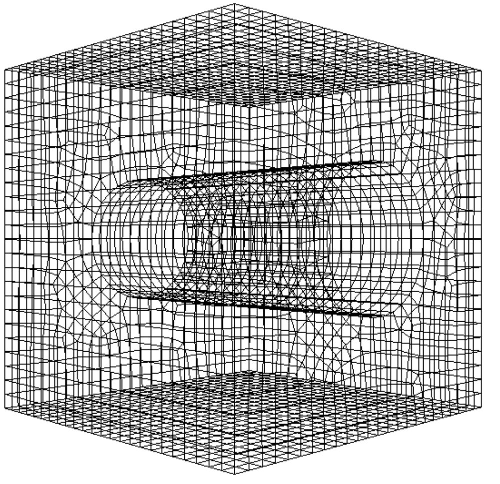
Domain decomposition :Graph Partitioning

Courtesy : metis (George Karypis & Vipin Kumar et**.** **a**ll)
C-DAC HPC-FTE Student Projects , 2011-12
Chaman Singh Verma et**.** all; & Jall Open source software

HPC GPU Cluster : Parallel Finite Element Method Comps. (Unstructured Grids)

Heat Transfer : GPU Implementation

- Access Pattern within a **32 X 32** block using **32 X 8** threads
  - Iterative methods based on Sparse Matrix Computations
  - Tiling – To handle large Mesh computations
  - Graph Partitioning and Graph Coloring techniques
  - Overlapping Comm. & Comps – CUDA Streams
- Performance 4x to 6x for un-optimised CUDA code



Domain decomposition based on Graph Partitioning

Courtesy : Chaman Singh Verma et. all; & Jall Open source software

HPC GPU Cluster : Parallel Finite Element Method Comps. (Unstructured Grids)

Heat Transfer : GPU Implementation

❖ Access Pattern within a **32 X 32** block using **32 X 8** threads

- Implicit Handling of Boundary Conditions - part of computations
- Graph Partitioning for Mesh Computations
- Graph Coloring for solver on a single node

❖ Performance 4x to 6x for un-optimised CUDA code

Type of Domain	Elements/ Nodes/ (Partitions/MPI Process)	Elapsed time (in seconds) MPI GPU Cluster		
		MPI	CUDA	OpenCL
2D-Grid FEM	14450(7396) (1/1)	9.72		
	14450(7396) (4/4)	5.64		
	14450(7396) (8/8)	3.28	0.64	1.12
3D-Grid Grid-FEM	343 (512) (1/1)	1.24		
	3375 (4096) (1/1)	8.63	1.46	3.09
	29791(32768) (1/1)	24.64	3.82	8.04

Domain decomposition based on Graph Partitioning

(*) = Speedup results were gathered using untuned and unoptimized versions of benchmarks & NVIDIA CUDA Prog. Env - This is C-DAC In-house HPC GPU Cluster project work

NVIDIA - NVML APIs : CUDA 5.0

http://www.nvidia.com

(*) = Speedup results were gathered using untuned and unoptimized versions of benchmarks & NVIDIA CUDA Prog. Env - This is C-DAC In-house HPC GPU Cluster project work in collaboration with NVIDIA

http://www.nvidia.com; NVIDIA CUDA

NVML (NVIDIA Management Library)

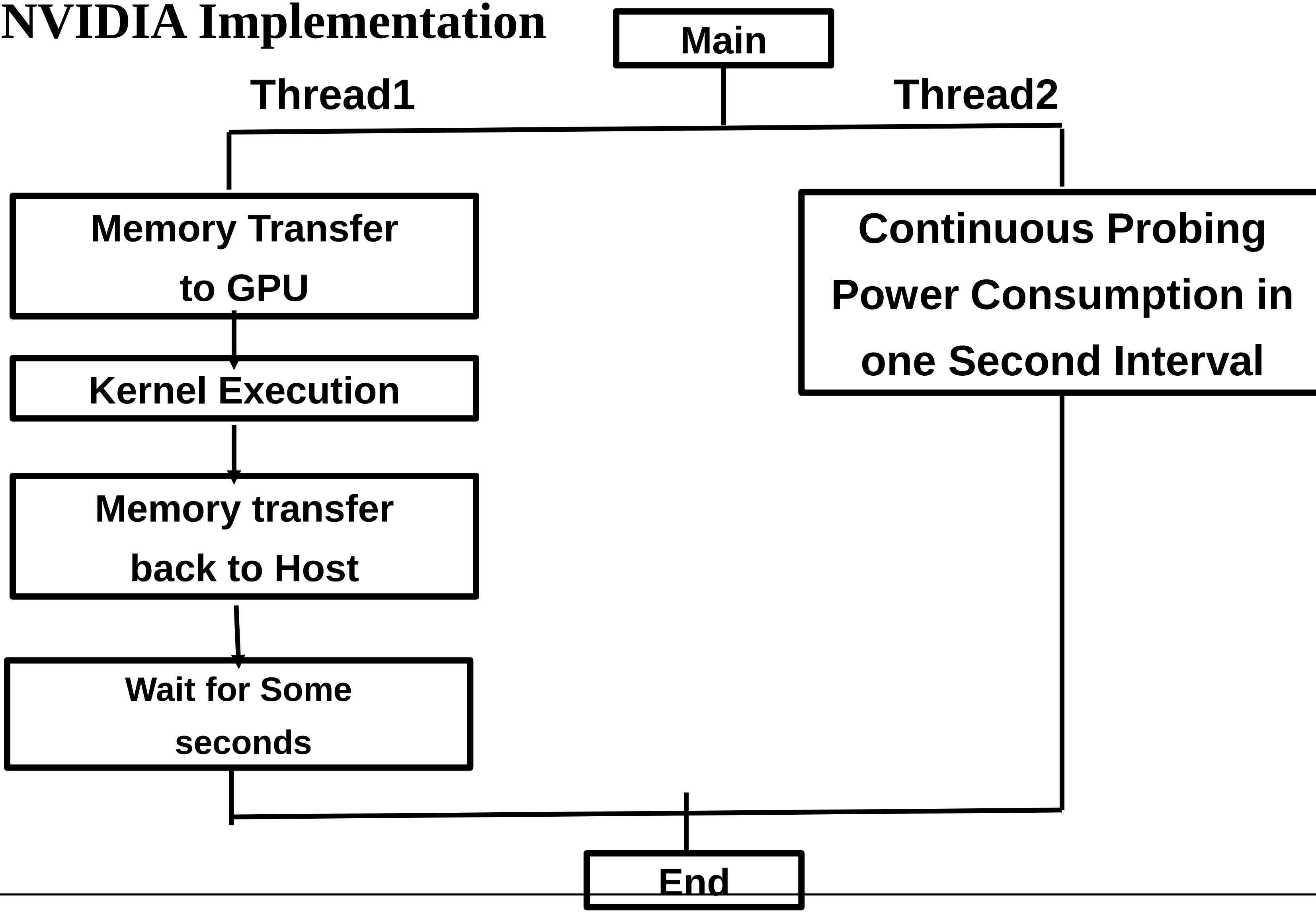
❖ **NVIDIA NVML : Power Measurement**

❖ Rate of sampling power usage is very low while measuring using **`nvidia-smi`** or **`nvml`** library, so unless the kernel is running for a long time we would not notice any change in power.

❖ nvidia provides a high-level utility called **`nvidia-smi`** which can be used to measure power, but its sample rate is too long to obtain useful measurements.

NVML (NVIDIA Management Library)

❖ **NVIDIA Implementation**



NVML Performance & Watts - for Matrix Comps.

Experiment Results CBLAS Lib(*)

- ❖ **Information**
  - ➢ Driver etc…
  - ➢ Device Query
  - ➢ Data Transfer from *host* to *Device*
  - ➢ Memory
  - ➢ Global Memory / Shared Memory Constant Memory
  - ➢ Data Transfer from *Device* to *host*

Time (sec.)	Power in milliWatt
0	30712
1	47064
2	49537
6	**132440**
7	**163942**
8	89673
9	61713
10	52588
11	50209
12	26704
13	19752
29	16797

Matrix Size : 10240 X 10240

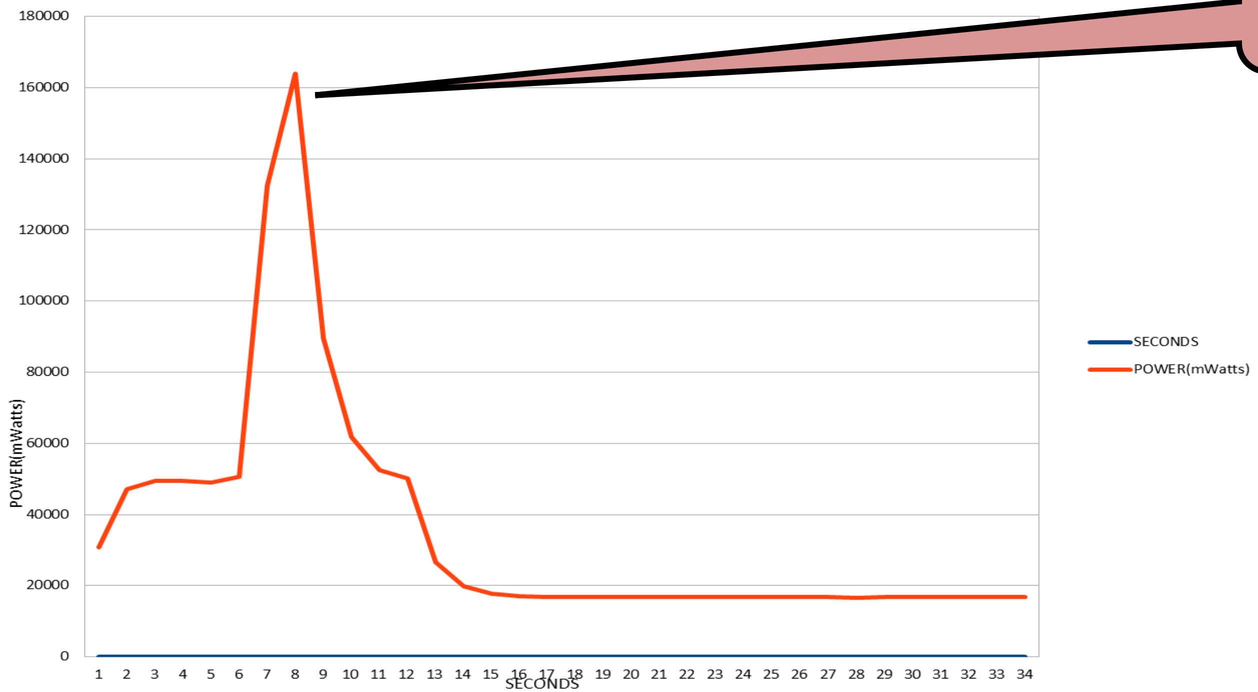
CPU + GPU Time (Sec): 2.575

CBLAS : 834 GFlops

(*) = Speedup results were gathered using untuned and unoptimized versions of benchmark and NVIDIA Prog. Env on NVIDIA KEPLER

NVML Performance & Watts - for Matrix Comps.

Experiment Results CBLAS Lib(*)



Peak mWatts Consumed

16392 Milliwatts

Seconds

Power mWatts

No Optimisations are carried to extract performance

(*) = Speedup results were gathered using untuned and unoptimized versions of benchmark and NVIDIA Prog. Env on NVIDIA KEPLER

NVML Performance & Watts - for Matrix Comps.

Experiment Results CBLAS Lib(*)

Time (sec.)	Power in milliwatt
0	30919
1	46505
4	49729
5	50012

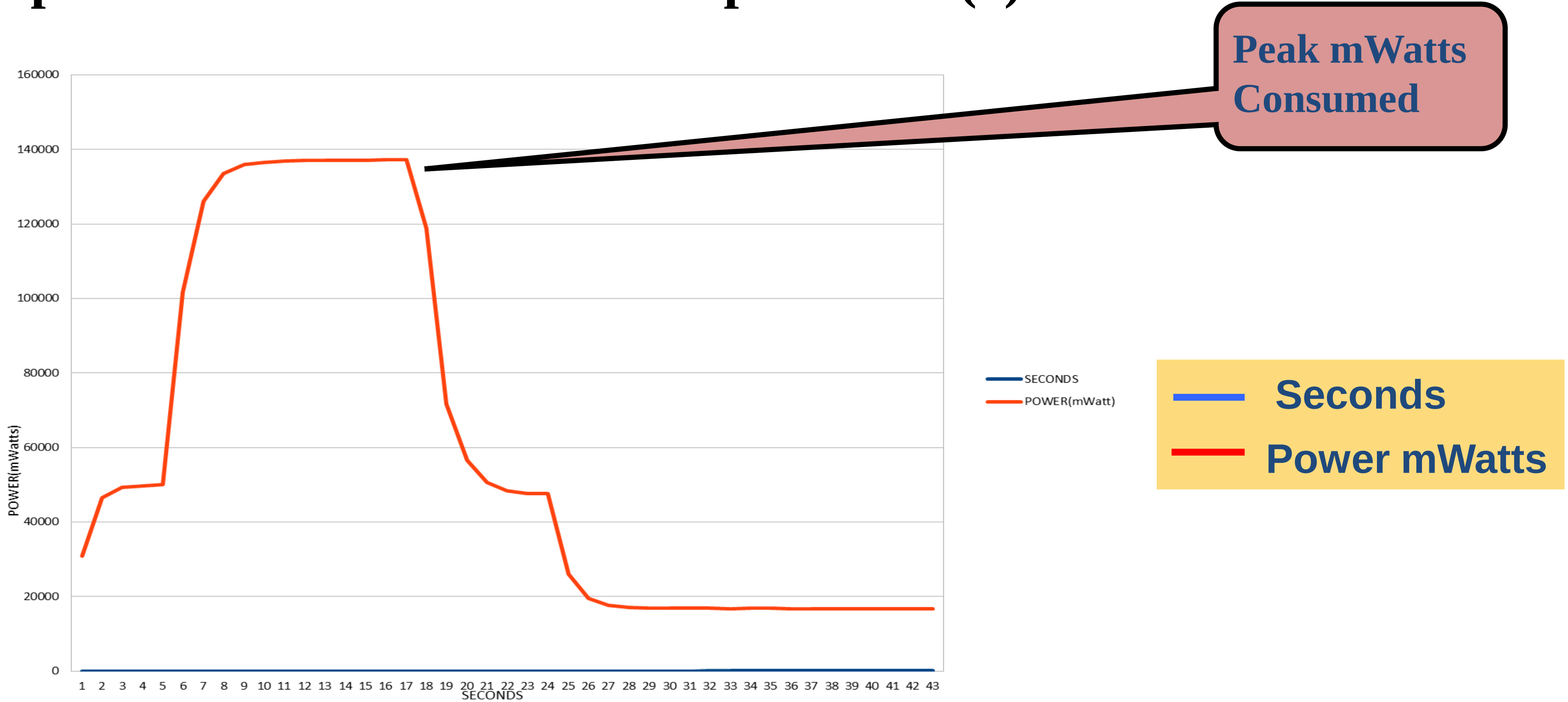
Time (Sec.)	Power in milliwatt
6	101504
7	**133627**
8	**135000**
10	**136574**
12	**137145**
16	**137330**
17	118776
18	71695
19	56504

Time (Sec.)	Power in milliwatt
20	50504
21	48395
23	47540
24	26035
25	19400
27	17656
28	16892
40	16797

(*) = Speedup results were gathered using untuned and unoptimized versions of benchmark and NVIDIA Prog. Env on NVIDIA KEPLER with NVML

NVML Performance & Watts - for Matrix Comps.

Experiment Results User Developed Code (*)



(*) = Speedup results were gathered using untuned and unoptimized versions of benchmark and NVIDIA Prog. Env on NVIDIA KEPLER

NVIDIA carma ARM Processor with CUDA

(*) = Speedup results were gathered using untuned and unoptimized versions of benchmarks (in-house developed) & NVIDIA CUDA Prog. Env - This is C-DAC In-house HPC GPU Cluster project work in collaboration with NVIDIA

http://www.nvidia.com; NVIDIA CUDA

NVIDIA ARM With Carma DevKit

Carma , the board includes the company's Tegra 3 quad-core ARM A9 processor, a Quadro 1000M GPU with 96 cores (good for 270 single-precision GFlops), as well as a PCIe X4 link, one Gigabit Ethernet interface, one SATA connector, three USB 2.0 interfaces as well as a Display port and HDMI. 2GB GPU Memory

- It uses the Tegra 3 chip as the basis and, thus, has four ARM cores and an NVIDIA GPU.
- In addition, the platform has 2 GB of DDR3 RAM (random access memory) as well.
- CUDA toolkit and a Ubuntu Linux-based OS

NVIDIA carma : Performance & Watts - Matrix Comps.

Experiment Results User Developed Code (*)

Matrix-Matrix Multiplication			
CUBLAS (Vendor)		User Code (IJK loop)	
GFLOPS	Time (Sec)	(GFLOPS)	Time (Sec)
125.7783	0.00834	28.9092	0.03627
125.7004	0.00834	28.9070	0.03627
125.7426	0.00834	28.9085	0.03627

SGEMM Matrix Size :

640 X 1280

CUBLAS

Time : 0.00834 sec

GFlops : 125.778

CUDA Mat Mat Mult

Time : 0.03627 sec

GFlops : 28.909

Peak Watts Consumed

39.5 watts

Using External Power Off Meter

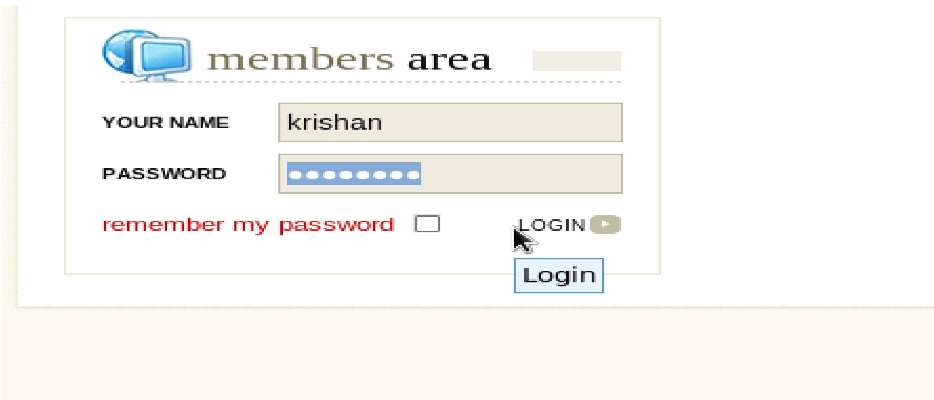
Seconds

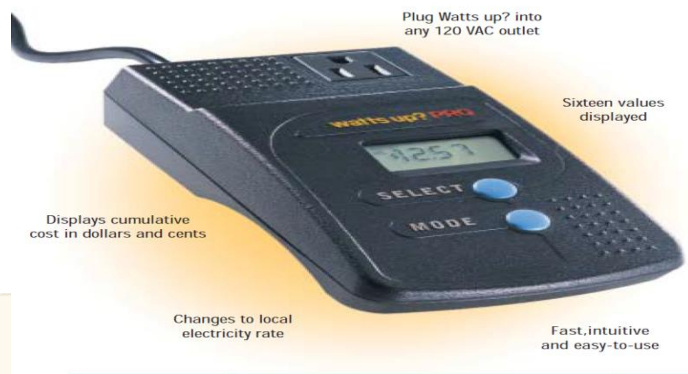
Power Watts

(*) = Speedup results were gathered using untuned and unoptimized versions of benchmark and NVIDIA Prog. Env on NVIDIA KEPLER

NVIDIA – carma - Power Meter : System Details

- Portal developed using TOMCAT to accommodate all servers
  - Login to portal





- Create Individual Session

NVIDIA – carma - Power Meter : System Details

❖ **Display reading of Power meter In tabular form**

Welcome

HOME GRAPHS SESSION LOGOUT

Power Meter Information

No.	DateTime	Meterid	Watts	WattHours	Volts	Amps	VoltAmp	Powercycle	Frequency	Rnc	Sr
1	2013-06-11 11:21:16	1785682602	25	0	2317	101	290	1	498	0	20
2	2013-06-11 11:21:17	1785682602	24	0	2318	100	290	1	498	0	20
3	2013-06-11 11:21:54	1785682602	25	0	2316	101	290	1	498	0	20
4	2013-06-11 11:22:56	1785682602	24	1	2320	101	292	1	498	0	20
5	2013-06-11 11:23:59	1785682602	25	1	2320	103	290	1	498	0	20
6	2013-06-11 11:25:01	1785682602	24	1	2320	102	292	1	498	0	20
7	2013-06-11 11:26:03	1785682602	24	1	2318	102	290	1	498	0	20
8	2013-06-11 11:27:05	1785682602	24	1	2315	101	292	1	498	0	20
9	2013-06-11 11:28:07	1785682602	24	2	2320	102	292	1	498	0	20
10	2013-06-11 11:29:10	1785682602	24	2	2319	101	292	1	498	0	20

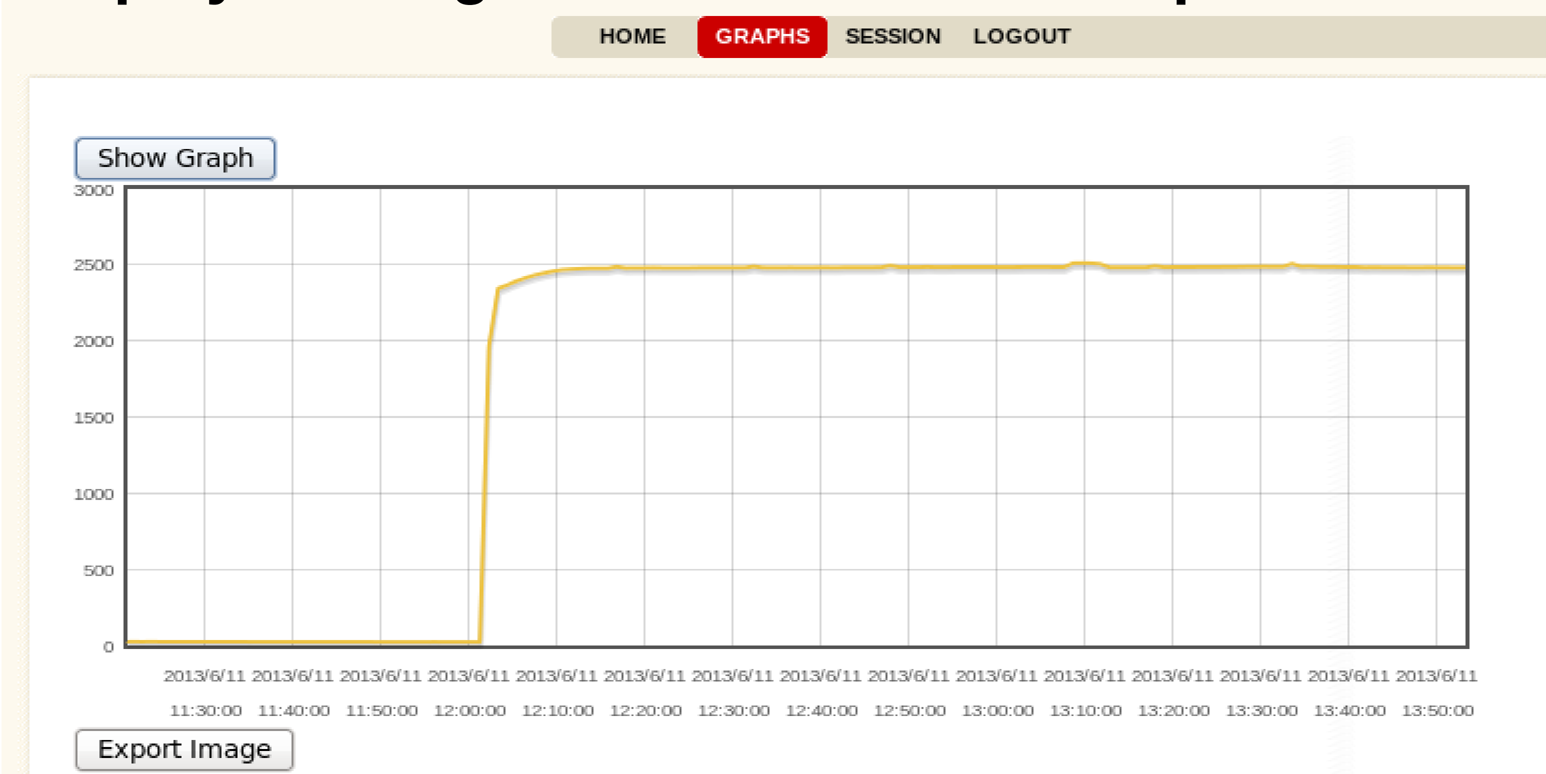
165 records found, displaying 1 to 10.
[First/Prev] 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 [Next/Last]
Export options: CSV | Excel | XML | PDF

(*) = Speedup results were gathered using untuned and unoptimized versions of benchmark and NVIDIA Prog. Env on NVIDIA carma with CUDA

NVIDIA – carma - Power Meter : System Details

Experiment Results User Developed Code (*)

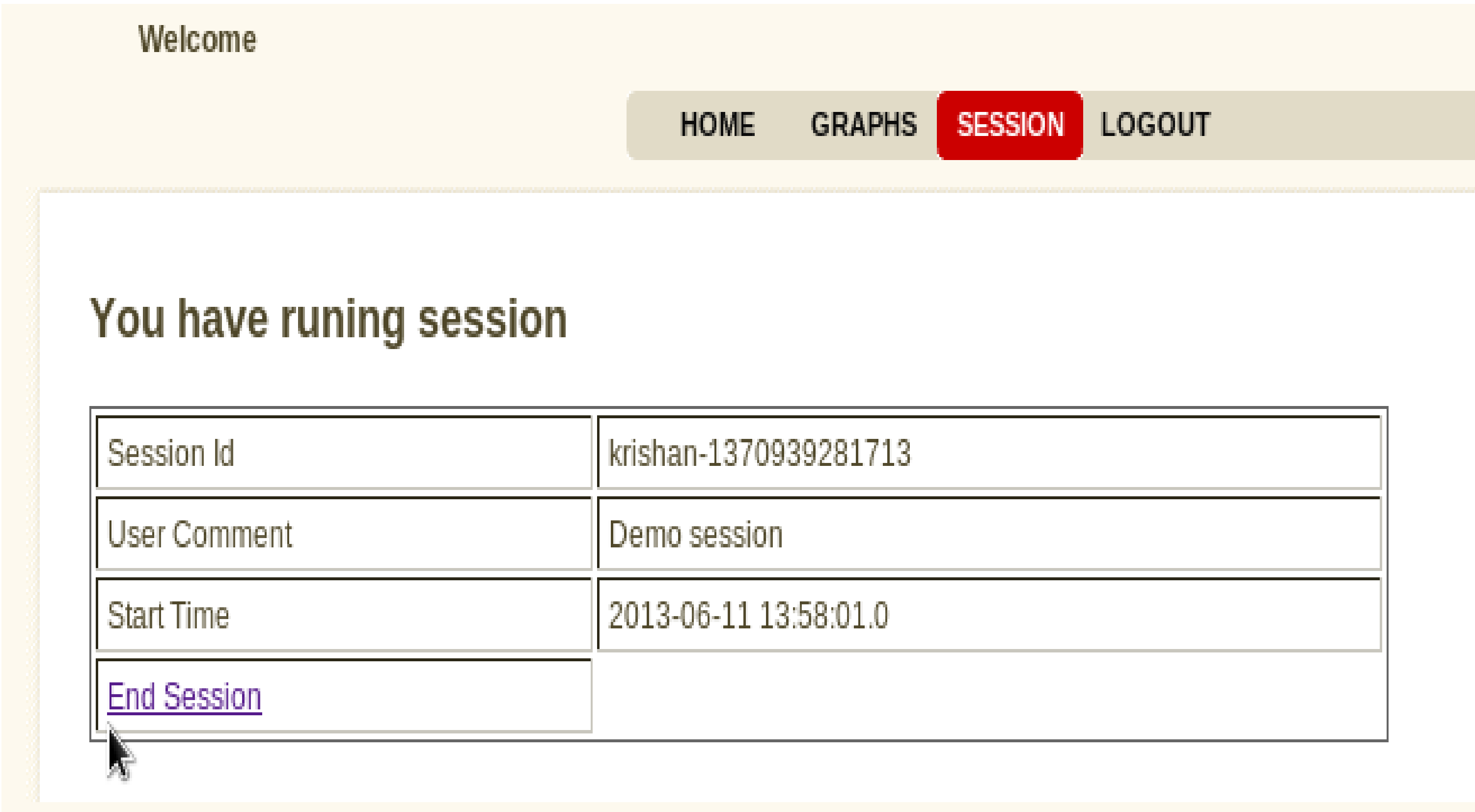
❖ Display reading of Power meter In Graphical format



(*) = Speedup results were gathered using untuned and unoptimized versions of benchmark and NVIDIA Prog. Env on NVIDIA carma with CUDA

NVIDIA – carma - Power Meter : System Details

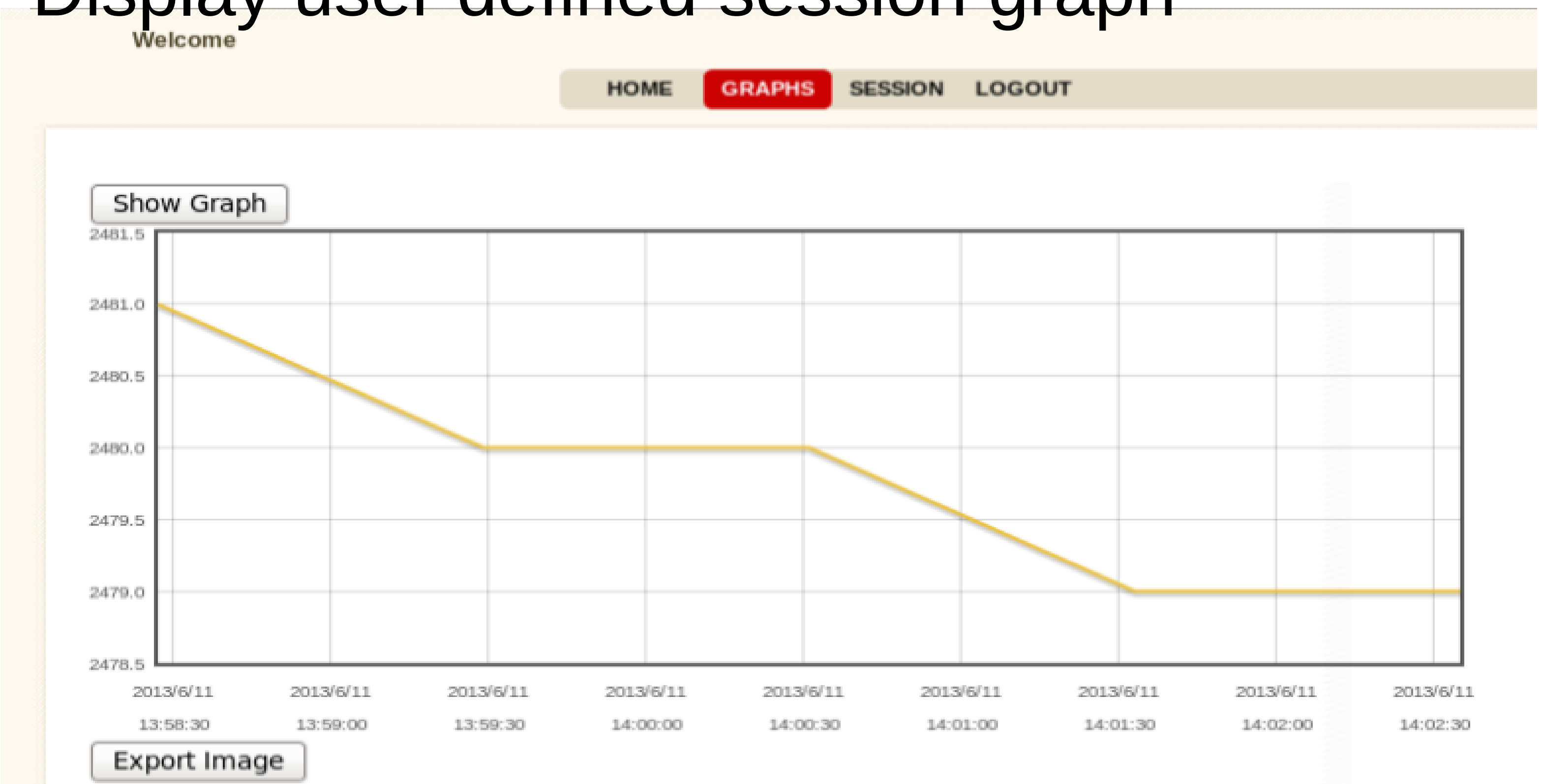
❖ Demo of running particular session in tabular



NVIDIA – carma - Power Meter : System Details

Experiment Results User Developed Code (*)

❖ Display user defined session graph



(*) = Speedup results were gathered using untuned and unoptimized versions of benchmark and NVIDIA Prog. Env on NVIDIA carma with CUDA

NVIDIA – carma - Power Meter : System Details

Systems Details

Node1: Jaguar.stp.cdac.ernet.in (1 GPU C2070)
CPU : Dual socket Quad core Intel Xeon; RAM : 16 GB
OS : centOS release 5.2 with kernel release 2.6.18-92.el5
Compiler : gcc & gnu libtool , NVIDIA CUDA compiler NVCC
nvidia-toolkit: 5.0
MPI : mpich2-1.0.7; **Interconnect :** Gigabit

Node2: Leopard.stp.cdac.ernet.in (2 GPUs C2050)
CPU : Dual socket Quad core Intel Xeon
RAM : 48 GB
OS : centOS release 5.2 with kernel release 2.6.18-92.el5
Compiler : gcc & gnu libtool , NVIDIA CUDA compiler NVCC
nvidia-toolkit: 5.0
MPI : mpich2-1.0.7 **Interconnect** : Gigabit

NVIDIA ARM With KAYLA DevKit(*)

- ❖ Kayla DevKit for computing on the ARM architecture – where supercomputing meets mobile computing.
- ❖ The Kayla DevKit hardware is composed of mini-ITX carrier board and NVIDIA® GeForce® GT640/GDDR5 PCI-e card.
- ❖ The mini-ITX carrier board is powered by NVIDIA Tegra 3 Quad-core ARM processor while GT640/GDDR5 enables Kepler GK208 for the next generation of CUDA and OpenGL application. Pre-installed with CUDA 5 and supporting OpenGL 4.3.
- ❖ Kayla provides ARM application development across the widest range of application types.

❖ **In Progress**

NVIDIA ARM With KAYLA DevKit

Form Factor	**Kayla mITX**
CPU	NVIDIA® Tegra® 3 ARM Cortex A9 Quad-Core with NEON
GPU	NVIDIA® GeForce® GT640/GDDR5 (TO BE PURCHASED SEPARATELY) Buy Now
Memory	2GB DRAM
CPU - GPU Interface	PCI Express x16 / x4
Network	1x Gigabit Ethernet
Storage	1x SATA 2.0 Connector
USB	2x USB 2.0
Software	Linux Ubuntu Derivative OS CUDA 5 Toolkit

Summary

- Good strategies for extracting high performance from individual subsystems on the CUDA enabled NVIDIA GPUs
- NVIDIA - CUDA (GPU is good choice)
- NVIDIA – CUDA Plenty of opportunities for further optimizations
- There are many good strategies for extracting high performance from individual subsystems on CUDA enabled NVIDIA GPU with CUDA Toolkit 5.0
- HPC GPU Cluster – MPI-CUDA with CUDA 5.0 gives advantages for Scalability and Performance for applications
- Power Efficient NVIDIA NVML APIs & Performance Issues

Source & Acknowledgements : NVIDIA, References

Summary

❖ Good strategies for extracting high performance from individual subsystems on the CUDA enabled NVIDIA GPUs

❖ NVIDIA - CUDA (GPU is good choice)

❖ NVIDIA – CUDA Plenty of opportunities for further optimizations

❖ There are many good strategies for extracting high performance from individual subsystems on CUDA enabled NVIDIA GPU with CUDA Toolkit 5.0

❖ HPC GPU Cluster – MPI-CUDA with CUDA 5.0 gives advantages for Scalability and Performance for applications

Source & Acknowledgements : NVIDIA, References

This page is intentionally kept blank

Part-II(F)

An Overview of CUDA enabled NVIDIA GPUs:
Prog. based on OpenACC

Source & Acknowledgements : NVIDIA, References

An Overview of OpenACC

Lecture Outline

Following topics will be discussed

- ❖ Part-I : An introduction to OpenACC
- ❖ Part-II : The OpenACC Pragmas
- ❖ Part-III: OpenACC Basic Examples
- ❖ Part-IV : Summary

Venue : CMSD, UoHYD ; Date : Oct 15-18, 2013

Source : NVIDIA & References given in the presentation

Introduction to OpenACC

- OpenACC: http://www.openacc-standard.org/
- Source : NVIDIA, NVIDIA-PGI & References

3 Ways to Accelerate Applications

Applications		
Libraries	Open ACC Directives	Programming Languages
"Drop-in" Acceleration	Easily Accelerate Applications	Maximum Flexibility

OpenACC Standard

NVIDIA | CRAY THE SUPERCOMPUTER COMPANY | PGI | CAPS

Source : NVIDIA, PGI, CRAY, CAPS, & References given in the presentation

OpenACC : Open Prog. Stanadard for Par. Comp.

"OpenACC will enable programmers to easily develop portable applications that maximize the performance and power efficiency benefits of the hybrid CPU/GPU architecture of Titan."

--Buddy Bland, Titan Project Director, Oak Ridge National Lab

"OpenACC is a technically impressive initiative brought together by members of the OpenMP Working Group on Accelerators, as well as many others. We look forward to releasing a version of this proposal in the next release of OpenMP."

--Michael Wong, CEO OpenMP Directives Board

Source : NVIDIA & References given in the presentation

OpenACC : The standard for GPU Devices

Easy: Directives are the easy path to accelerate compute intensive applications

Open: OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors

Powerful: GPU Directives allow complete access to the massive parallel power of a GPU

Source : NVIDIA & References given in the presentation

OpenACC : High-level, with low-level access

- Compiler directives to specify parallel regions in C, C++, Fortran
  - OpenACC compilers offload parallel regions from host to accelerator
  - Portable across OSes, host CPUs, accelerators, and compilers
- Create high-level heterogeneous programs
  - Without explicit accelerator initialization,
  - Without explicit data or program transfers between host and accelerator
- Programming model allows programmers to start simple
  - Enhance with additional guidance for compiler on loop mappings, data location, and other performance details
- Compatible with other GPU languages and libraries
  - Interoperate between CUDA C/Fortran and GPU libraries
  - e.g. CUFFT, CUBLAS, CUSPARSE, etc.

Source : NVIDIA & References given in the presentation

OpenACC : High-level, with low-level access

❖ Full OpenACC 1.0 Specification available online http://www.openacc-standard.org

❖ Quick reference card also available

❖ Beta implementations available now from PGI, Cray, and CAPS

❖ Information is given in References


The OpenACC™ API
QUICK REFERENCE GUIDE

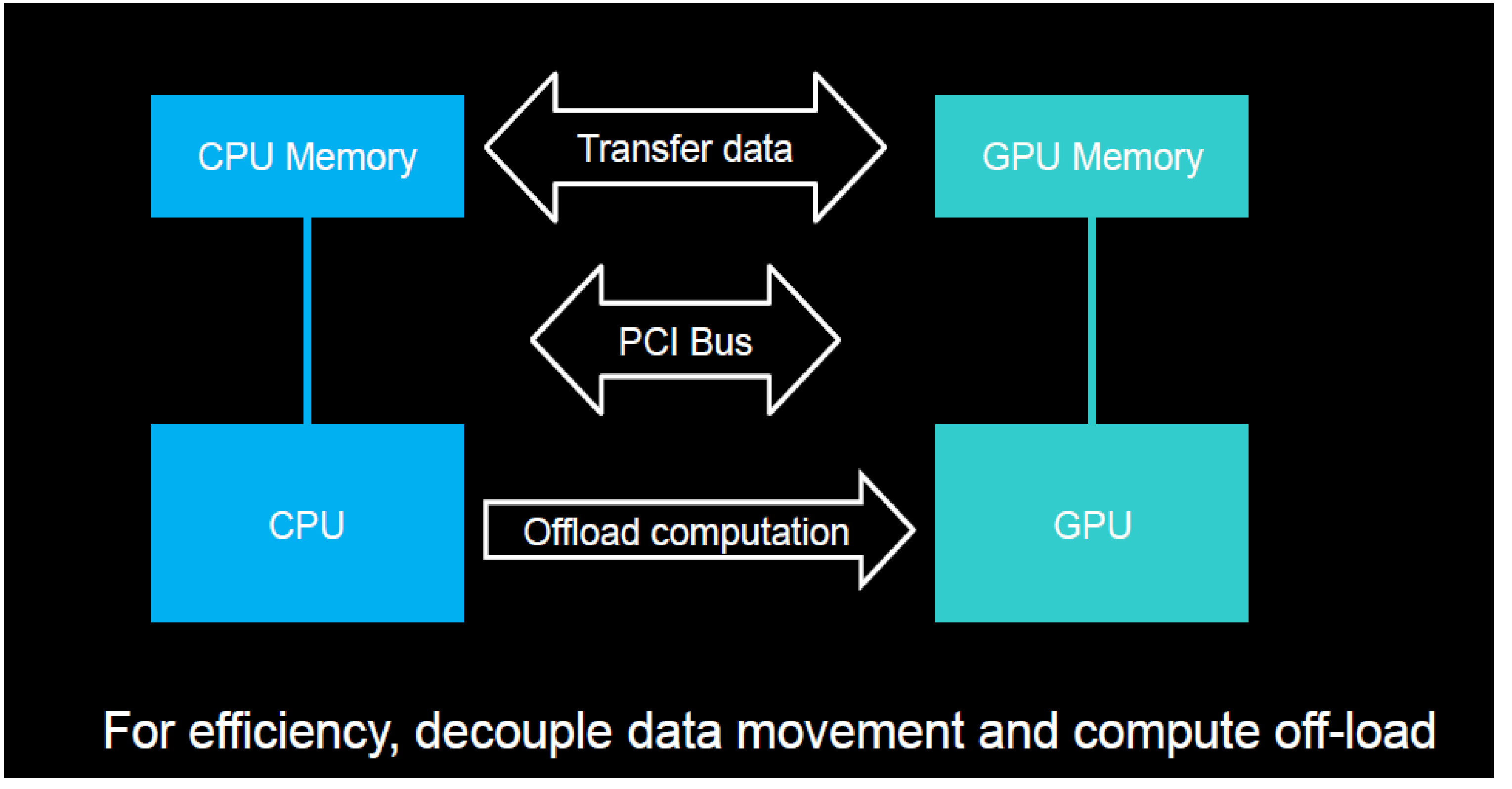
The OpenACC Application Program Interface describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators.

Most OpenACC directives apply to the immediately following structured block or loop; a structured block is a single statement or a compound statement (C or C++) or a sequence of statements (Fortran) with a single entry point at the top and a single exit at the bottom.

Version 1.0, November 2011

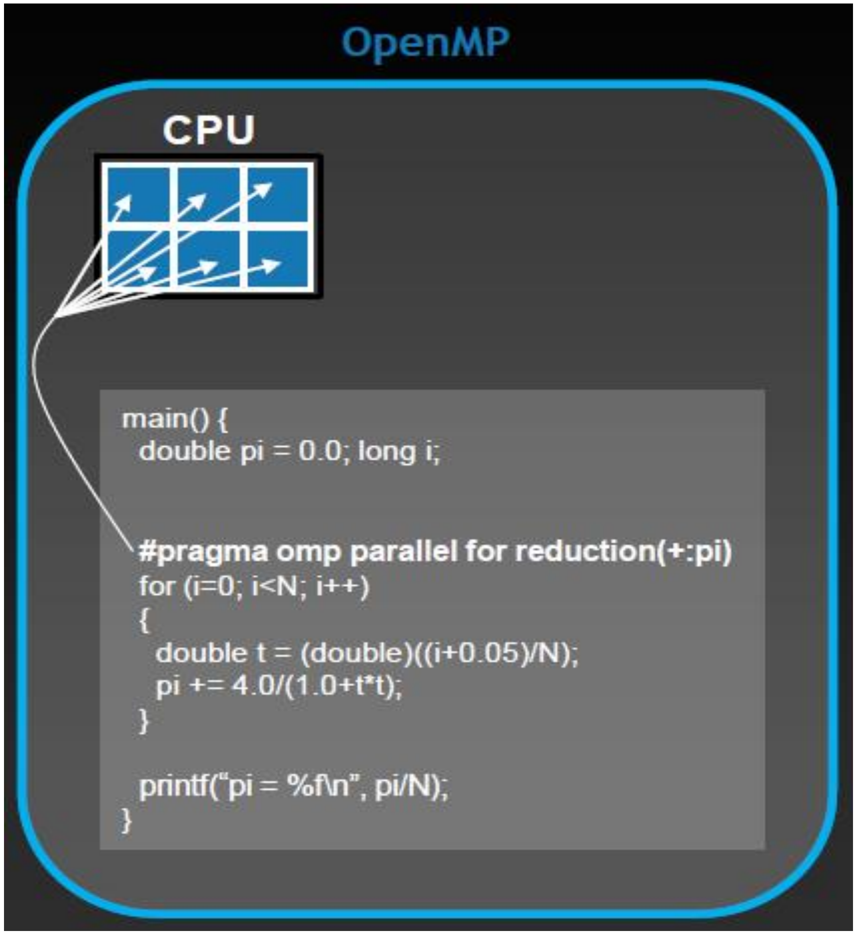
Source : NVIDIA & References given in the presentation

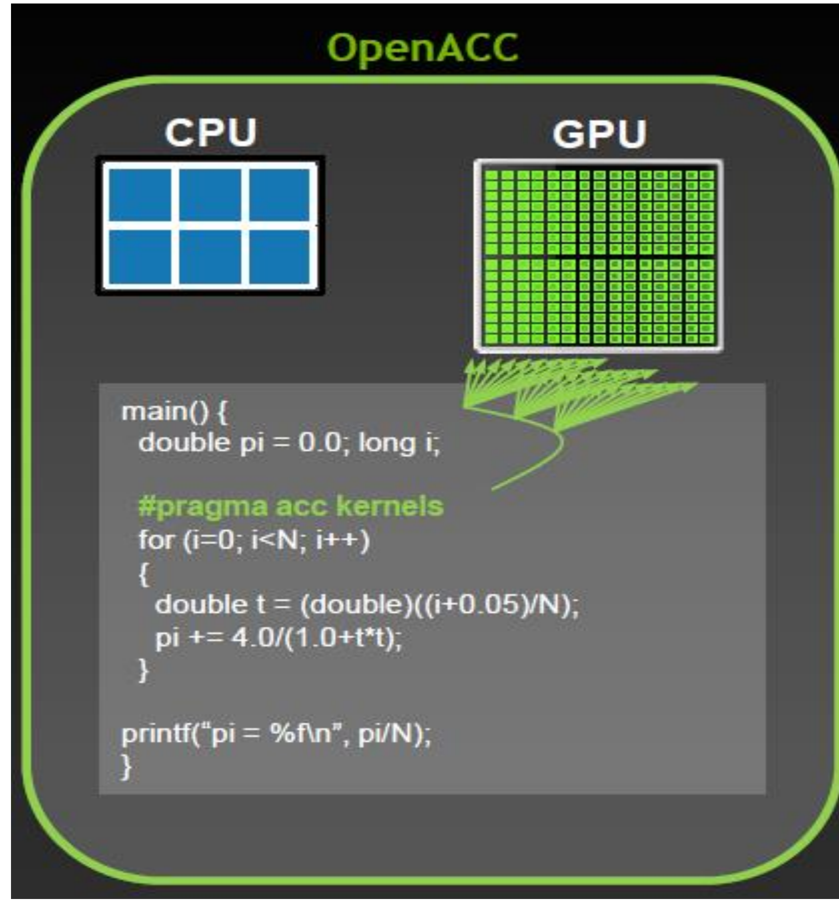
OpenACC Basic Concepts



Source : NVIDIA & References given in the presentation

Familiar to OpenACC Programmers





Source : NVIDIA & References given in the presentation

OpenACC Compile & Run

Compile and run

C:

pgcc –acc -ta=nvidia -Minfo=accel –o saxpy_acc saxpy.c

Fortran:

pgf90 –acc -ta=nvidia -Minfo=accel –o saxpy_acc saxpy.f90

Compiler output:

```
pgcc -acc -Minfo=accel -ta=nvidia -o saxpy_acc saxpy.c
saxpy:
8, Generating copyin(x[:n-1])
     Generating copy(y[:n-1])
     Generating compute capability 1.0 binary
     Generating compute capability 2.0 binary
9, Loop is parallelizable
     Accelerator kernel generated
      9, #pragma acc loop worker, vector(256) /* blockIdx.x threadIdx.x */
        CC 1.0 : 4 registers; 52 shared, 4 constant, 0 local memory bytes; 100% occupancy
        CC 2.0 : 8 registers; 4 shared, 64 constant, 0 local memory bytes; 100% occupancy
```

Source : NVIDIA & References given in the presentation

What is OpenACC?

- Accelerator programming API standard to program accelerators
  - Portable across operating systems and various types of host CPUs and GPU accelerators.
  - Allows parallel programmers to provide simple hints, known as "**directives**," to the compiler, identifying which areas of code to accelerate, without requiring programmers to modify or adapt the underlying code itself.
  - Aimed at incremental development of accelerator code
- Effort driven by vendors with the input from users/ applications

Source : NVIDIA & References given in the presentation

OpenACC Vendor Support

- The current vendors support OpenACC are: Cray: High-Level GPU directives
  - **PGI: PGI accelerator directives**
  - **CAPS Enterprise: HMPP**
  - **NVIDIA: CUDA, OpenCL**
  - **Others: As this defacto standard gains traction**
- Strong interaction with the OpenMP accelerator subcomittee with input from other institutions

Source : NVIDIA & References given in the presentation

Impact of OpenACC

- Phase 1: First Standardization of High-Level GPU directives. [Short-term, Mid-term]
  - Heavily influenced by NVIDIA hardware.

- Phase 2: Experiences from OpenACC will drive the effort of OpenMP for Accelerators
  - More general solution
  - Might take years to develop
  - Better interoperability with OpenMP

Source : NVIDIA & References given in the presentation

Overview of the OpenACC directives

- ❖ Directives facilitate code development for accelerators

- ❖ Provide the functionality to:

  - ➢ Initiate accelerator startup/shutdown
  - ➢ Manage data or program transfers between host (CPU) and accelerator
  - ➢ Scope data between accelerator and host (CPU)
  - ➢ Manage the work between the accelerator and host.
  - ➢ Map computations (loops) onto accelerators
  - ➢ Fine-tune code for performance

Source : NVIDIA & References given in the presentation

Execution Model

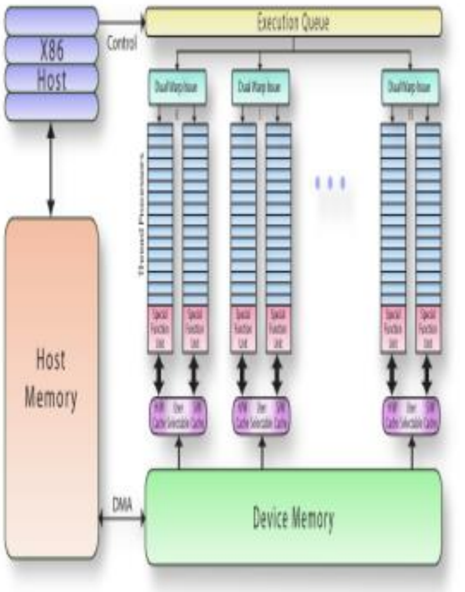
- Bulk of computations executed in CPU, compute intensive regions offloaded to accelerators
- Accelerators execute parallel regions:
  - Use work-sharing and kernel directives
  - Specification of coarse and fine grain parallelization
- The **host** is responsible for
  - Allocation of memory in accelerator
  - Initiating data transfer
  - Sending the code to the accelerator
  - Waiting for completion
  - Transfer the results back to host
  - De-allocating memory
  - Queue sequences of operations executed by the device

Source : NVIDIA & References given in the presentation

Execution Model

❖ **Parallelism:**

- ➢ Support coarse-grain parallelism
  - Fully parallel across execution units
  - Limited synchronizations across
  - coarse-grain parallelism
- ➢ Support for fine-grain parallelism
  - Often implemented as SIMD
  - Vector operations
- ➢ Programmer need to understand the differences between them.
  - Efficiently map parallelism to accelerator
  - Understand synchronizations available
- ➢ Compiler may detect data hazards
  - Does not guarantee correctness of the code



Source : NVIDIA & References given in the presentation

Memory Model

- Host + Accelerator memory may have completely separate memories
  - **Host may not be able to read/write device memory that is not mapped to a shared virtual addressed.**
- All data transfers must be initiated by host
  - **Typically using direct memory accesses (DMAs)**
- Data movement is implicit and managed by compiler
- Device may implement weak consistency memory model
  - **Among different execution units**
  - **Within execution unit: memory coherency guaranteed by barrier**

Source : NVIDIA & References given in the presentation

Memory Model (2)

- ❖ Programmer must be aware of:
  - ➢ Memory bandwidth affects compute intensity
  - ➢ Limited device memory
  - ➢ Assumptions about cache:
    - **Accelerators may have software or hardware managed cache**
    - **May be limited to read only data**
- ❖ Caches are managed by the compiler with hints by the programmer
- ❖ Compiler may **auto-scope** variables based on static information or enforce runtime checks.

Source : NVIDIA & References given in the presentation

Categories of OpenACC APIs

- Accelerator Parallel Region / Kernels Directives
- Loop Directives
- Data Declaration Directives
- Data Regions Directives
- Cache directives
- Wait / update directives
- Runtime Library Routines
- Environment variables

Source : NVIDIA & References given in the presentation

Directives Format

- **C/C++:**

#pragma acc *directive-name [clause [,clause]…] new-line*

- **Fortran:**

!$acc *directive-name [clause [, clause]…]*

c$acc *directive-name [clause [, clause]…]*

$acc *directive-name [clause [, clause]…]

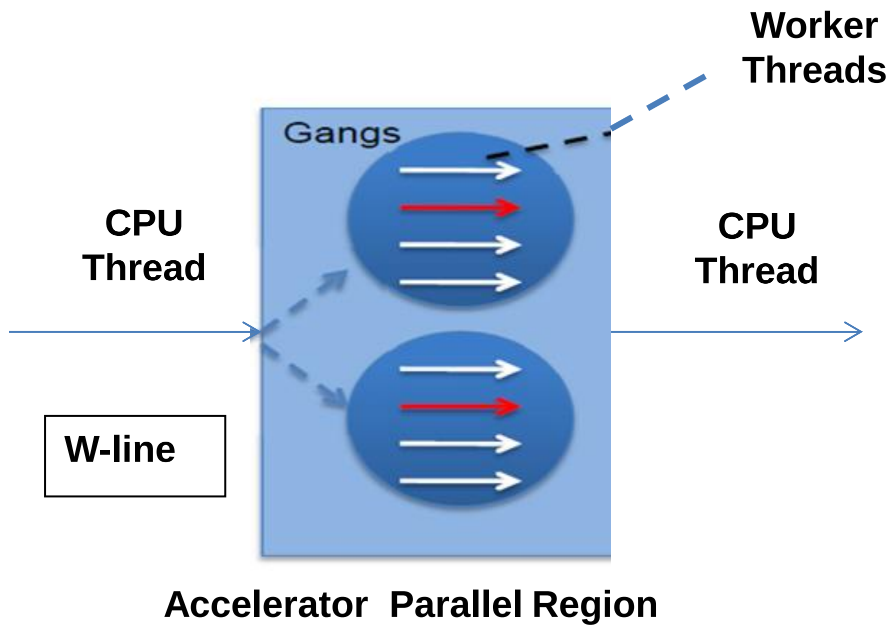
Source : NVIDIA & References given in the presentation

OpenACC Parallel Directive

- Starts parallel execution on accelerator
- Specified by:
  - **#pragma acc parallel [clause [,clause]…] new-line structured block**
- When encountered:
  - Gangs of workers threads are created to execute on accelerator
  - One worker in each gang begins executing the code following the structured block
  - Number of gangs/workers remains constant in parallel region

Source : NVIDIA & References given in the presentation

OpenACC Parallel Directive



Source : NVIDIA & References given in the presentation

OpenACC Parallel Directive (2)

- The clauses for the ***!$acc parallel*** directive are:
  - if(condition)
  - async [(scalar-integer-expression)]
  - num_gangs (scalar-integer-expression)
  - num_workers (scalar-integer-expression)
  - vector_length (scalar-integer-expression)
  - reduction (operator:list)
  - copy (list)
  - copyout (list)
  - create (list)
  - private (list)
  - firstprivate (list)

Source : NVIDIA & References given in the presentation

OpenACC Parallel Directive (3)

- The clauses for the **!$acc parallel** directive are:
  - present (list)
  - present_or_copy (list)
  - present_or_copyin (list)
  - present_or_copyout (list)
  - present_or_create (list)
  - deviceprt (list)
- If **async** is not present, there is an implicit barrier at the end of accelerator parallel region.
- **present_or_copy** default for aggregate types (arrays)
- **private or copy** default for scalar variables

Source : NVIDIA & References given in the presentation

OpenACC Kernel Directive

- Defines a region of a program that is to be compiled into a sequence of kernels for execution on the accelerator
- Each loop nest will be a different kernel
- Kernels launched in order in device
- Specified by:
  - ***#pragma acc kernels [clause [,clause]…] new-line***
    structured block

Source : NVIDIA & References given in the presentation

OpenACC Kernel Directive (2)

- Kernels directive may not contain nested parallel or kernel directive
- Configuration of gangs and worker thread may be different for each kernel
- The clauses for the ***!$acc kernels*** directive are: if(condition)
  - async [(scalar-integer-expression)]
  - copy (list)
  - copyin (list)
  - copyout (list)
  - create (list)
  - private (list)
  - firstprivate (list)

Source : NVIDIA & References given in the presentation

OpenACC Kernel Directive (3)

- The clauses for the ***!$acc kernels*** directive are: present (list)
  - present_or_copy (list)
  - present_or_copyin (list)
  - present_or_copyout (list)
  - present_or_create (list)
  - deviceprt (list)
- If **async** is present, kernels or parallel region will execute asynchronous on accelerator
- **present_or_copy** default for aggregate types (arrays)
- **private or copy** default for scalar variables

Source : NVIDIA & References given in the presentation

OpenACC Parallel/Kernel Clauses

- **if clause**
  - Optional clause to decide if code should be executed on accelerator or host
- **async clause**
  - Specifies that a parallel accelerator or kernels regions should be executed asynchronously
  - The host will evaluate the integer expression of the async clause to test or wait for completion with the wait directive
- **num_gangs clause**
  - Specifies the number of gangs that will be executed in the accelerator parallel region
- **num_workers clause**
  - Specifies the number of workers within each gang for a accelerator parallel region

Source : NVIDIA & References given in the presentation

OpenACC Parallel/Kernel Clauses

- **vector_length clause**
  - Specifies the vector length to use for the vector or SIMD operations within each worker of a gang
- **private clause**
  - A copy of each item on the list will be created for each gang
- **firstprivate clause**
  - A copy of each item on the list will be created for each gang and initialized with the value of the item in the host
- **reduction clause**
  - Specifies a reduction operation to be perform across gangs using a private copy for each gang.
  - Support for: +, *, max, min, &, |, &&, ||
  - Other operators available in Fortran: .neqv., .eqv.

Source : NVIDIA & References given in the presentation

OpenACC Data Directive

- The data construct defines scalars, arrays and subarrays to be allocated in the accelerator memory for the duration of the region.

- Can be used to control if data should be copied-in or out from the host

- Specified by:

  - *#pragma acc data [clause [,clause]…] new-line*
    structured block

Source : NVIDIA & References given in the presentation

OpenACC Data Directive

- The clauses for the ***!$acc data*** directive are:
  - if(condition)
  - copy (list)
  - copyin (list)
  - copyout (list)
  - create (list)
  - present (list)
  - present_or_copy (list)
  - present_or_copyin (list)
  - present_or_copyout (list)
  - present_or_create (list)
  - deviceptr (list)

Source : NVIDIA & References given in the presentation

OpenACC Data Directive

❖ copy clause

➢ Specifies items that need to be copied-in from the host to accelerator, and then copy-out at the end of the region

➢ Allocates accelerator memory for the copy items.

❖ copy-in clause

➢ Specifies items that need to be copied-in to the accelerator memory

➢ Allocates accelerator memory for the copy-in items

❖ copy-out clause

➢ Specifies items that need to be copied-out to the accelerator memory

➢ Allocates accelerator memory for the copy-out items

Source : NVIDIA , PGI & References given in the presentation

OpenACC Data Directive (2)

- ❖ **create clause**
  - ➢ Specifies items that need to allocated (created) in the accelerator memory
  - ➢ The values of such items are not needed by the host

- ❖ **copy-in clause**
  - ➢ Specifies items that need to be copied-in to the accelerator memory
  - ➢ Allocates accelerator memory for the copy-in items

- ❖ **present clause**
  - ➢ Specifies items are already present in the accelerator memory
  - ➢ The items were already allocated on other data, parallel or kernel regions. (i.e. inter-procedural calls)

Source : NVIDIA & References given in the presentation

OpenACC Data Directive (3)

- **present_or_copy clause**
  - Tests if a data item is already present in the accelerator. If not, it will allocate the item in the accelerator and copy-in and out its value from/to the host
- **present_or_copyin clause**
  - Test if a data item is already present in the accelerator. If not, it will allocate the item in the accelerator and copy-in its value from the host
- **present_or_copyout clause**
  - Test if a data item is already present in the accelerator. If not, it will allocate the item in the accelerator and copy-out its value to the host
- **present_or_create clause**
  - Test if a data item is already present in the accelerator. If not, it will allocate the item in the accelerator (no initialization)

Source : NVIDIA & References given in the presentation

OpenACC Loop Directive

- Used to describe what type of parallelism to use to execute the loop in the accelerator.

- Can be used to declare loop-private variables, arrays and reduction operations.

- Specified by:
  - **#pragma acc loop [clause [,clause]…] new-line for loop**

Source : NVIDIA & References given in the presentation

OpenACC Loop Directive (2)

- The clauses for the **!$acc loop** directive are:
  - collapse (n)
  - gang [(scalar-integer-expression)]
  - worker [(scalar-integer-expression)]
  - vector [(scalar-integer-expression)]
  - seq
  - independent
  - private (list)
  - reduction (operator : list)

- **collapse directive**
  - Specifies how many tightly nested loops are associated with the **loop** construct

Source : NVIDIA & References given in the presentation

OpenACC Loop Clauses

- **gang clause**
  - Within a parallel region: it specifies that the loop iteration need to be distributed among **gangs**.
  - Within a kernel region: that the loop iteration need to be distributed among **gangs**. It can also be used to specify how many gangs will execute the iteration of a loop
- **worker clause**
  - Within a parallel region: it specifies that the loop iteration need to be distributed **among workers of a gang**.
  - Within a kernel region: that the loop iteration need to be distributed **among workers of a gang**. It can also be used to specify how many workers of a **gang** will execute the iteration of a loop
- **seq clause**
  - Specifies that a loop needs to be executed sequentially by the accelerator

Source : NVIDIA & References given in the presentation

OpenACC Loop Clauses

- **vector clause**
  - Within a parallel region: specifies that the loop iterations need to be in vector or SIMD mode. It will use the vector length specified by the parallel region
  - Within a kernel region: specifies that the loop iterations need to be in vector or SIMD mode. If an argument is specified, the iterations will be processed in vector strips of that length.
- **independent clause**
  - Specifies that there are no data dependences in the loop
- **private clause**
  - Specifies that a copy of each item on the list will be created for each iterations of the loop.
- **reduction clause**
  - Specifies that a reduction need to be perform associated to a gang, worker or vector

Source : NVIDIA & References given in the presentation

OpenACC Cache Directive

- Specifies array elements or subarrays that should be fetched into the highest level of the cache for the body of the loop.

- Specified by:
  - **#pragma acc cache(list) new-line**

Source : NVIDIA & References given in the presentation

OpenACC Combined Directive

- Some directives can be combined into a single one

- Combined directives are specified by:

  - ***#pragma acc parallel loop [clause [,clause]…] new-line***
    for loop
  - ***#pragma acc kernels loop [clause [,clause]…] new-line***
    for loop

Source : NVIDIA & References given in the presentation

OpenACC Declare Directive

- Used in the variable declaration section of program to specify that a variable should be allocated, copy-in/out in an implicit data region of a function, subroutine or program .

- If specified within a Fortran Module, the implicit data region is valid for the whole program.

- Specified by:
  - ***#pragma acc declare [clause [,clause]…] new-line***

Source : NVIDIA & References given in the presentation

OpenACC Declare Directive (2)

- The clauses for the **!$acc data** directive are: copy (list)
  - copyin (list)
  - copyout (list)
  - create (list)
  - present (list)
  - present_or_copy (list)
  - present_or_copyin (list)
  - present_or_copyout (list)
  - present_or_create (list)
  - deviceptr (list)
  - device_resident (list)

Source : NVIDIA & References given in the presentation

OpenACC Update Directive

- Used within a data region to update / synchronize the values of the arrays on both the host or accelerator

- Specified by:
  - #pragma acc update [clause [,clause]…] new-line

- The clauses for the **!$acc update** directive are:
  - host (list)
  - device (list)
  - if (condition)
  - async [(scalar-integer-expression)]

Source : NVIDIA & References given in the presentation

OpenACC Wait Directive

- It causes the program to wait for completion of an asynchronous activity such as an accelerator parallel, kernel region or update directive
- Specified by:
  - #pragma acc wait [(scalar-integer-expression)] new-line
- It will test and evaluate the integer expression for completion
- If no argument is specified, the host process will wait until all asynchronous activities have completed
- Can be specified per CPU/Thread basis.

Source : NVIDIA & References given in the presentation

OpenACC runtime calls

- int acc_get_num_devices(acc_device_t)
- void acc_set_device_type(acc_device_t)
- acc_device_t acc_get_device_type()
- acc_set_device_num(int, acc_device_t)
- int acc_get_device_num(acc_device_t)
- int acc_async_test(int)
- int acc_async_test_all()
- void acc_async_wait(int)
- void acc_async_wait_all()
- void acc_init(acc_device_t)
- void acc_shutdown (acc_device_t)
- int acc_on_device(acc_device_t)
- void* acc_malloc(size_t)
- void acc_free(void*)

setenv ACC_DEVICE_TYPE
NVIDIA setenv
ACC_DEVUCE_NUM 1
Environment Variables

Source : NVIDIA & References given in the presentation

OpenACC runtime calls

- Some vendors will provide implementations of OpenACC at the end of this year.
- The OpenACC Cray implementation is available
- Use OpenACC as the standard GPU programming directives
- applications users are starting to use
- Visit References for runtime calls

Source : NVIDIA & References given in the presentation

This page is intentionally kept blank

Part-III

Introduction to Heterogeneous Computing

Why OpenCL ?

Heterogeneous Computing with OpenCL

Lecture Outline

Following topics will be discussed

- Part-I : An introduction to Heterogeneous comp. - OpenCL
- Part-II : The OpencL Specification - Kernels
- Part-III : OpenCL Device Architectures
- Part-IV : OpenCL Basic Examples
- Part-V : Understanding OpenCL's Concurrency and Execution Model

Source : NVIDIA, Khronos AMD, References

Source : References given in the presentation

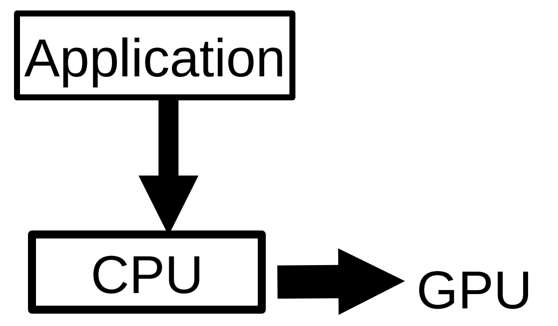
Software in Many-core world

GPU Computing : Think in Parallel - Some Design Goals

- ❖ Performance =

  parallel hardware + scalable parallel program

- ❖ GPU Computing drives new applications
  - Reducing “Time to Discovery”
  - 100 x Speedup changes science & research methods

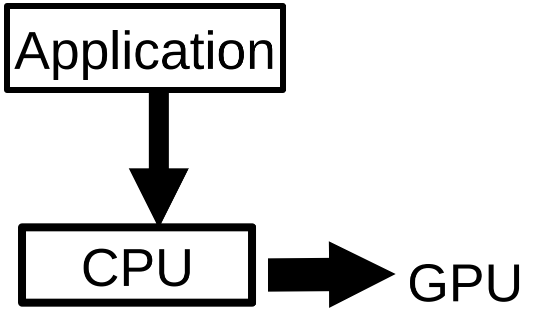


- ❖ New applications drive the future of GPUs
  - Drives new GPU capabilities
  - Drives hunger

Source : NVIDIA, Khronos, AMD, References

GPU Computing: Take Advantage of Shared Memory

- ❖ Hundreds of times faster than global memory
- ❖ Threads can cooperate via shared memory
- ❖ Use one/ a few threads to load/computer data shared by all threads
- ❖ Use it to avoid non-coalesced access
  - Stage loads and stores in shared memory to re-order non-coalesceable addressing
  - Matrix transpose example later



Source : NVIDIA, Khronos AMD, References

GPU Computing : Think in Parallel

GPU Computing: Optimise Algorithms for the GPU

- ❖ Maximize independent parallelism
- ❖ Maximize arithmetic intensity (math/bandwidth)
- ❖ Sometimes it’s better to recompute than to cache
  - GPU spends its translators on ALUs, not memory
- ❖ Do more computation on the GPU to avoid costly data transfers
  - Even low parallelism computations can sometimes be faster than transferring back and forth to host

Source : NVIDIA, Khronos AMD, References

Software in Many-core world

❖ High Level Abstraction that hide complexity of hardware

❖ A heterogeneous programming language exposes heterogeneity

- Trend towards increasing abstraction
- **One language does'nt have to address the needs of every community of programmers**
- High level frame works - High level languages and map to a low-level hardware abstraction layer for portability

❖ OpenCL is hardware-abstraction

Source : NVIDIA, Khronos AMD, References

Part-III(A)

Introduction to OpenCL

Standardization

OpenCL tries to Standardize Parallel Programming

To standardize general purpose parallel programming for any application

Suitable for Heterogeneous systems – different Mircoprocessor Architectures (Ex : PCs - X86; PCs with discrete or integrated GPUs, Cell Phones, Embedded Systems

Khronos OpenCL working group making aggressive progress (www.khronos.org)

Source : NVIDIA, Khronos AMD, References

OpenCL tries to Standardize Parallel Programming

What Does OpenCL Mean ? : Challenging Objectives :

❖ Standardize framework and language for multiple heterogeneous processors

➢ Developed in collaboration with industry leaders

❖ Software Developers

➢ OpenCL enabled you to write parallel programs that will run portably on many devices

➢ Royalty free – with no cost to use the API

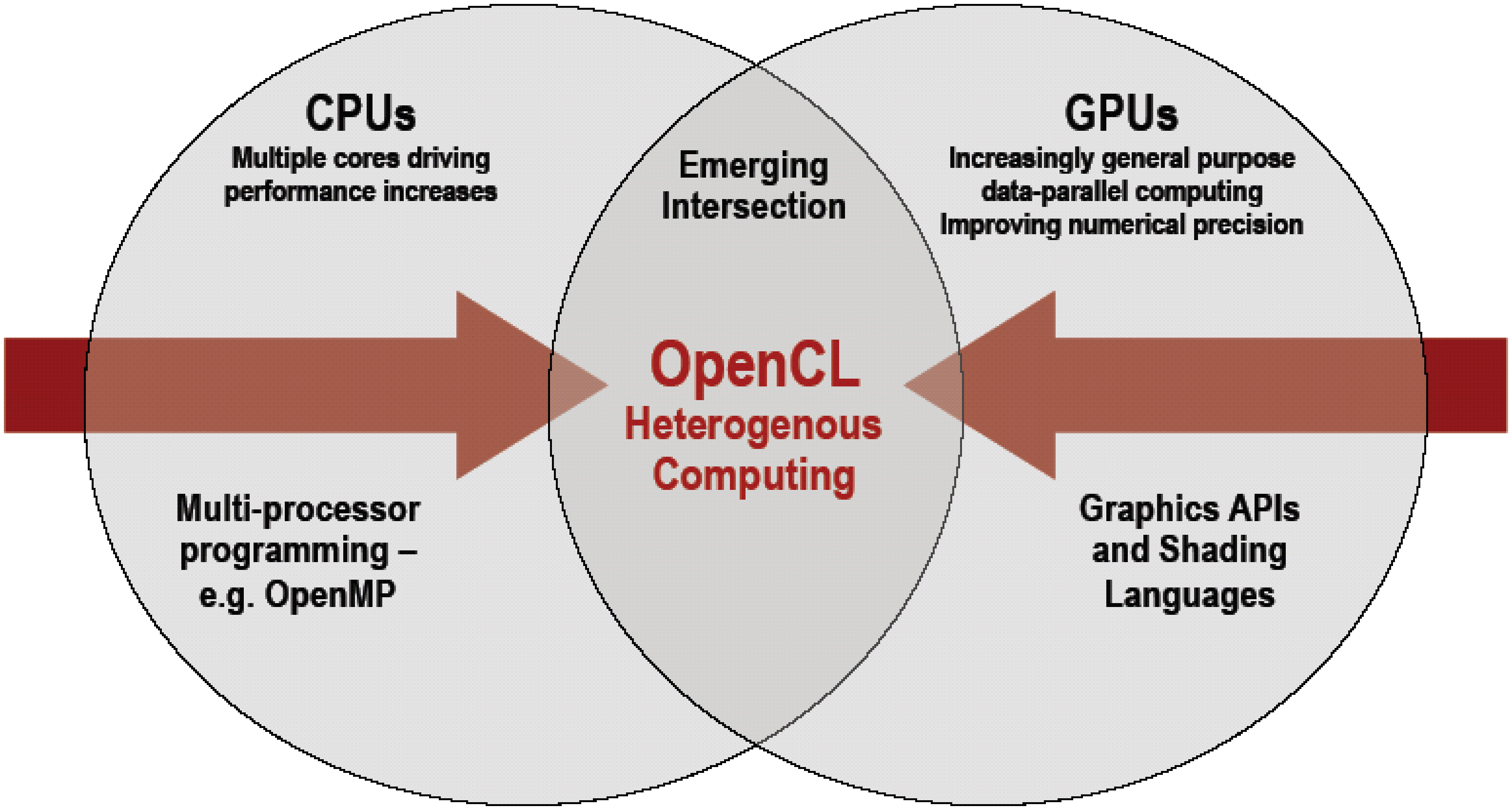
❖ **End-User Benefits**

➢ A wide range of innovative applications will be enabled and accelerated by unleashing the parallel computing capabilities of their systems and devices

Source : NVIDIA, Khronos AMD, References

OpenCL tries to Standardize Parallel Programming

Processor Parallelism : Processor Parallelism



OpenCL – Open Computing Language

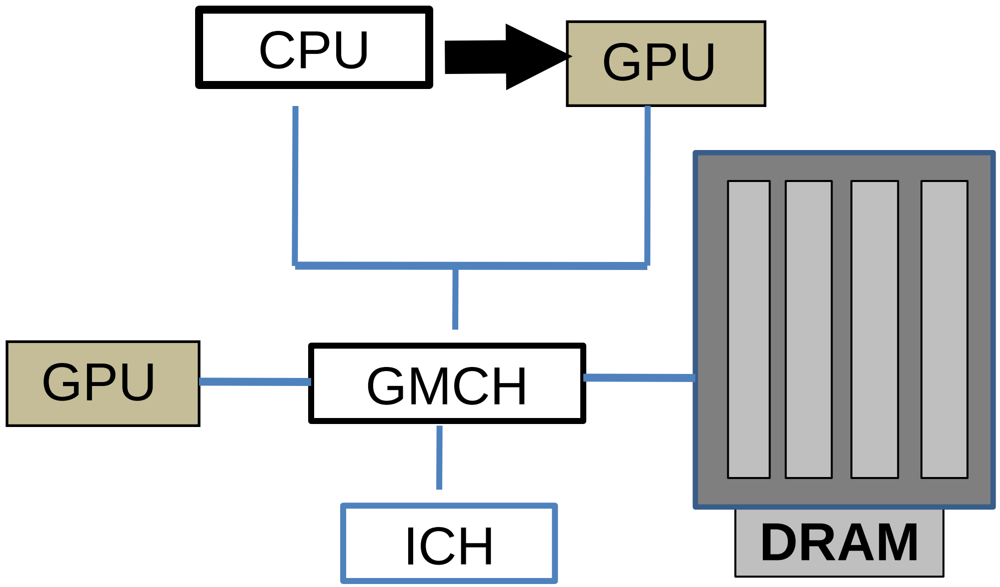
Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors

Source : NVIDIA, Khronos AMD, References

OpenCL tries to Standardize Parallel Programming

Why OpenCL

Need Hybrid Programming on Heterogeneous Comp. Platforms



The future belongs to heterogeneous many-core platforms

Source : Khronos, OpenCL Prog, Guide by Aaftab Munshi etc. &References

OpenCL tries to Standardize Parallel Programming

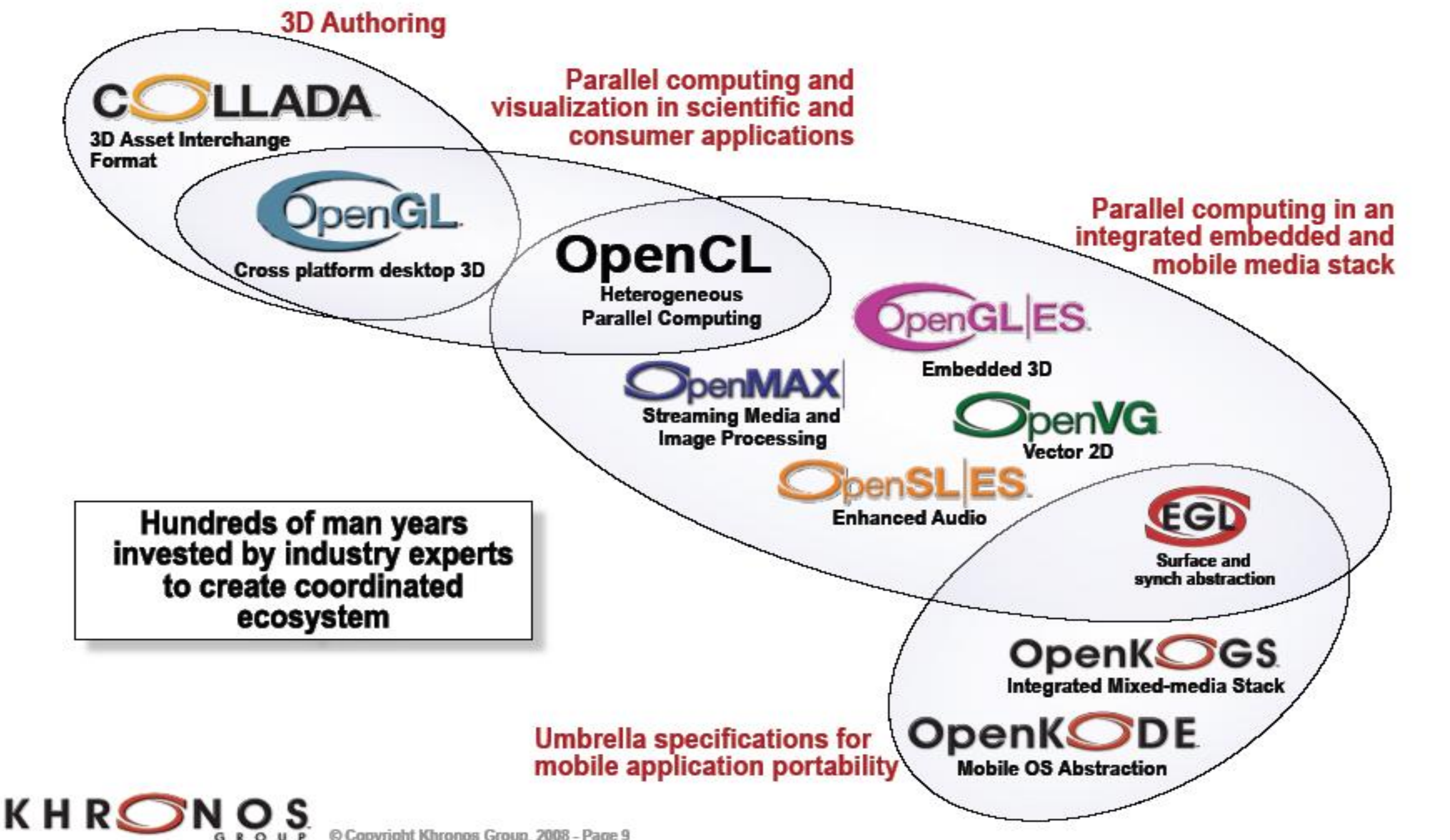
OpenCL Specification working group :

3DLabs, Activation Blizzard, AMD, Apples, ARM, Barco, Broadcom, Codeplay, Electronic Arts, Ericsson, Freescale, Hi, IBM, Intel, Imagine technologies, Motorla, Movid, Nokia, Nvidia, QNX, RapidMind Samsung, Seaweed,Takuni, Texas Instruments, University (Sweden), Microsoft

- **Here are some of the other companies in the OpenCL working group**

Source : NVIDIA, Khronos AMD, References

OpenCL and the Khronos EcoSystem



Source : NVIDIA, Khronos AMD, References

OpenCL tries to Standardize Parallel Programming

Why OpenCL

Hybrid Programming on Heterogeneous Comp. Platforms

Co-existence of Accelerators
Intel Xeon (Phi) RC-FPGA, & GPGPUs

Heterogeneous Comp. Platforms – Power & Energy Efficiency

Capacitance = 2.2 C
Voltage = 0.6V
Frequency = 0.5f
Power = 0.396 CV^2f

How our software should adapt to these platforms ?

Source : NVIDIA, Khronos AMD, References

OpenCL tries to Standardize Parallel Programming

Background & Challenging Objectives :

- OpenGL: Open Graphics Library
  - Widely supported application programming interface (API) for graphics ONLY
- OpenCL: "CL" Stands for Computing Language
  - providing an API library
  - Modifies C and C++ parallel programming
  - New Initiatives for other programming languages(Fortran)

Aim: to standardize general purpose parallel programming for any application

Source : NVIDIA, Khronos   AMD, References

The OpenCL Standard

OpenCL Working Group : Challenging Objectives

- ❖ Diverse Industry Participation
  - Processor vendors, System OEMS, Middleware vendors, Application Developers
- ❖ Many Industry-leading experts involved in OpenCL's design
  - A healthy diversity of industry perspectives
- ❖ Apple initially proposed the working group
  - And served as specification editor

Source : NVIDIA, Khronos AMD, References

The OpenCL Standard

❖ Challenging Objectives :

- ➢ Arrive at a common set of programming standards that are acceptable to a range of competing needs and requirements

- ➢ ***The Khronos*** consortium – manages the OpenCL standard

  - Developed an applications programming interface (API) that is general enough to run on significantly different architectures while being adaptable enough that each hardware platforms can still obtain high performance.

  - Using the core language and correctly following the specification, any program designed for one-vendor can execute on another's hardware.

Source : NVIDIA, Khronos AMD, References

The OpenCL Standard

Challenging Objectives :

- Diverse Industry Participation
  - Processor vendors, System OEMS, Middleware vendors, Application Developers
- Many Industry-leading experts involved in OpenCL's design
  - A healthy diversity of industry perspectives
- Apple initially proposed the working group
  - And served as specification editor

Source : NVIDIA, Khronos AMD, References

The OpenCL Standard

❖ **Challenging Objectives** :

- OpenCL C is a restricted version of the C99 language with extension appropriate for executing data-parallel code on a variety of heterogeneous devices.

- Aimed for full support for the IEEE 754 formats

- Programming language, well suited to the capabilities of current heterogeneous platforms

Source : NVIDIA, Khronos AMD, References

The OpenCL Standard

❖ **Challenging Objectives :**

➢ The model set forth by OpenCL creates portable, vendor- and device-independent programs that are capable of being accelerated on many different platforms.

- The OpenCL API is C wit h a C++ Wrapper API that is defined in terms of the C-API.
- There are third-party bindings for many languages, including Java, Python, and .NET
- The code that executes on an OpenCL device, which in general is not the same device as the host-CPU, is written in the OpenCL C language.

Source : NVIDIA, Khronos AMD, References

OpenCL : Standardize Parallel Programming

- **Threading in Model for data level parallelism OpenCL**
  - Closely resembles the models in AMD-ATI Stream, CUDA & RapidMind
  - OpenCL threading is largely implicit
  - OpenCL allows programmers to manage threads more explicitly if programmers wish
- **Task-level parallelism**
  - Concurrently execute multiple kernels on multiple kernels on multiple CPUs, GPUs or systems with mixed architecture

Source : NVIDIA, Khronos AMD, References

OpenCL Design Requirements

- **Use all computational resources in system**
  - Program GPUs, CPUs and other processors as peers
  - Support both data- and task- parallel compute models
- **Efficient c-based parallel programming model**
  - Abstract the specified of underlying hardware
- **Abstraction is low-level, high-performance but device-portable**
  - Approachable –but primarily targeted at expert developers
  - Ecosystem foundation – no middleware or “convenience” functions
- **Implementation on a range of embedded, desktop, and server systems**
  - HPC desktop, and handheld profiles in on specification
- **Drive future hardware requirements**
  - Floating point precision requirements
  - Application to both consumer and HPC applications

Source : NVIDIA, Khronos AMD, References

OpenCL Design Requirements

- **Efficient c-based parallel programming model**
  - Abstract the specified of underlying hardware
- **Abstraction is low-level, high-performance but device-portable**
  - Approachable –but primarily targeted at expert developers
  - Ecosystem foundation – no middleware or “convenience” functions

Source : Khronous, References

OpenCL Design Requirements

- ❖ **Implementation on a range of embedded, desktop, and server systems**
  - ➢ HPC desktop, and handheld profiles in on specification
- ❖ **Drive future hardware requirements**
  - ➢ Floating point precision requirements
  - ➢ Application to both consumer and HPC applications

Source : NVIDIA, Khronos AMD, References

Design Goals of OpenCL

- Use all computational resources in system
  - GPUs and CPUs as peers
  - Data- and task- parallel compute model
- Efficient parallel programming model
  - Based on C
  - Abstract the specifics of underlying hardware
- Specify accuracy of floating-point computations
  - IEEE 754 compliant rounding behaviour
  - Define maximum allowable error of math functions

Source : NVIDIA, Khronos AMD, References

OpenCL Task Parallel Execution Model

- ❖ Data-parallel execution model must be implemented by all OpenCL compute devices
- ❖ Some computer devices such as CPUs can also execute task parallel compute kernels
  - ➢ Executes as a single work-item
  - ➢ A compute kernel written in OpenCL
  - ➢ A native C / C++ function

Source : NVIDIA, Khronos AMD, References

Part-III(B)

OpenCL – Models

Conceptual Foundations of OpenCL

An Application for a heterogeneous platform must carry out the following steps.

- ❖ Discover the completion that make-up the heterogeneous system
- ❖ Probe the characteristics of these components, so that the software can adapt to specific features of different hardware elements
- ❖ Create the blocks of instructions (Kernels) that will run on the platform

Source : NVIDIA, Khronos AMD, References

Conceptual Foundations of OpenCL

An Application for a heterogeneous platform must carry out the following steps.

- ❖ Set up and manipulate memory objects involved in the computation.
- ❖ Execute the kernels in the right order and on the right components of the system
- ❖ Collect the final results
  - Above steps are accomplished through a series of APIs inside OpenCL plus a programming environment for the kernels

Source : NVIDIA, Khronos AMD, References

Conceptual Foundations of OpenCL

An Application for a heterogeneous platform must carry out the following steps.

- ❖ Discover the completion that make-up the heterogeneous system
- ❖ Probe the characteristics of these components, so that the software can adapt to specific features of different hardware elements
- ❖ Create the blocks of instructions (Kernels) that will run on the platform

Source : NVIDIA, Khronos AMD, References

Conceptual Foundations of OpenCL

An Application for a heterogeneous platform must carry out the following steps.

- ❖ Set up and manipulate memory objects involved in the computation.
- ❖ Execute the kernels in the right order and on the right components of the system
- ❖ Collect the final results
  - Above steps are accomplished through a series of APIs inside OpenCL plus a programming environment for the kernels

Source : NVIDIA, Khronos AMD, References

The OpenCL Specification – Models

- The OpenCL specification is defined in four parts, called models, that can be summarized as follows.

  - Platform Model
  - Execution Model
  - Memory Model
  - Programming Model

Source : NVIDIA, Khronos AMD, References

The OpenCL Specification – Models

❖ OpenCL Software Stack

- **Platform Layer**
  - ➢ Query and select computer devices in the system
  - ➢ Initialize a compute device(s)
  - ➢ Create compute contexts and work-queues
- **Runtime**
  - ➢ Resource management
  - ➢ Execute compute kernels
- **Compiler**
  - ➢ A subset of ISO C99 with appropriate language additions
  - ➢ Compile and build compute program executable
  - ➢ Online or offline

Source : NVIDIA, Khronos AMD, References

The OpenCL Specification – Models

❖ The OpenCL specification is defined in four parts, called models, that can be summarized as follows.

➢ Platform Model

• High Level description of the heterogeneous system

➢ Execution Model

• An abstract representation of how stream of instructions execute on the heterogeneous system

Source : NVIDIA, Khronos AMD, References

The OpenCL Specification – Models

- The OpenCL specification is defined in four parts, called models, that can be summarized as follows.
  - Memory Models
    - The Collection of memory regions within OpenCL and how they interact during at OpenCL computation
  - Programming Model
    - The high-level abstractions a programmer uses when designing algorithms to implement an application

Source : NVIDIA, Khronos AMD, References

The OpenCL Specification

- The OpenCL specification is defined in four parts, called models, that can be summarized as follows.

  - Platform Model
  - Execution Model
  - Memory Model
  - Programming Model

Source : NVIDIA, Khronos AMD, References

Part-III(C)

OpenCL Specification

Platform Model

(In brief)

The OpenCL Specification

❖ Platform model :

- Specifies that there is one processor coordinating the execution (***the host***) and one or more processors capable of executing OpenCL C Code (***the devices***).
- It defines an abstract hardware model that is used by programmers when writing OpenCL functions (Called ***Kernels***) that execute on the devices.
- The platform model defines the relation between the host an device.
  - i.e., OpenCL implementation executing on a host x86 GPU, which is using a GPU device as an accelerator

Source : NVIDIA, Khronos AMD, References

The OpenCL Specification

❖ **Platform model :**

- Platforms can be thought of a vendor – specific implementations of the OpenCL API.
- The platform model also presents an abstract device architecture that programmers target writing OpenCL C code.
- Vendors map this abstraction architecture to the physical hardware.

Source : NVIDIA, Khronos AMD, References

OpenCL PLATFROM AND DEVICES

Host-Device Interaction

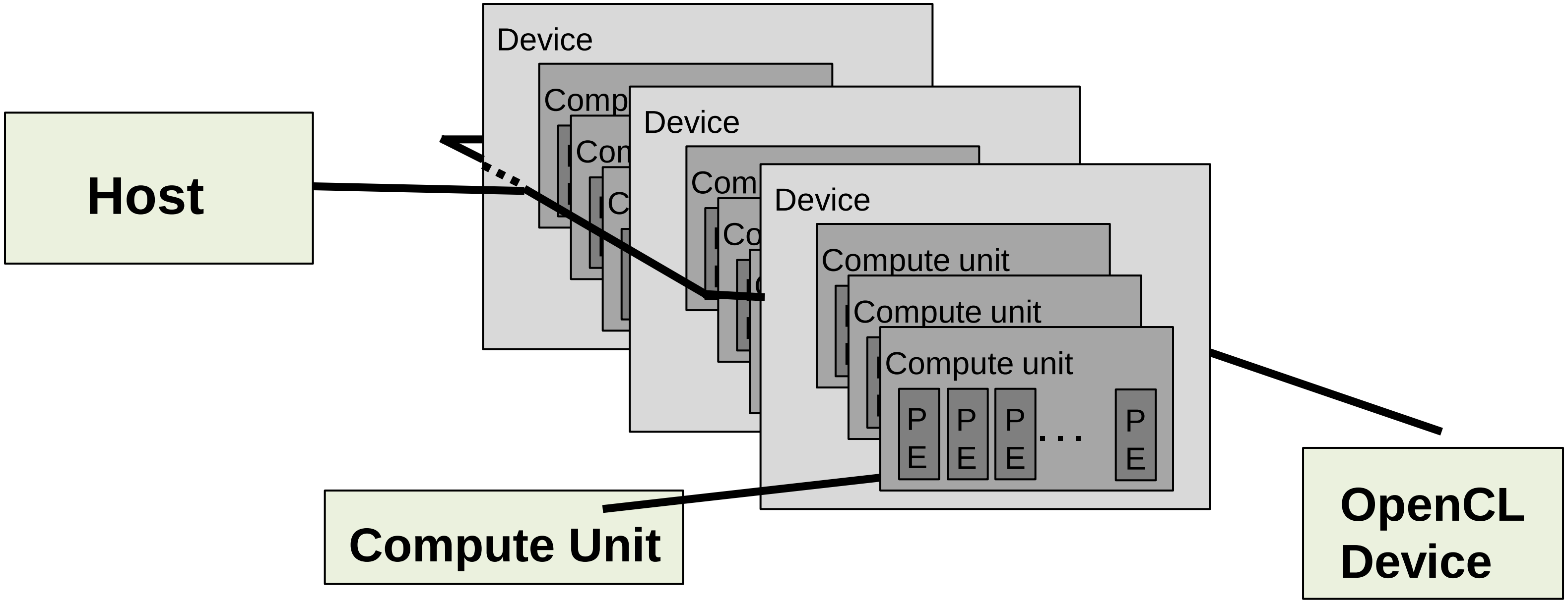
❖ Platform Model

- Provides an abstract hardware model for devices
- Present an abstract device architecture that programmers target when writing OpenCL C code.
- Vendor-specific implementation of the OpenCL API.

❖ Platform Model

- Defines a device as an array of compute units
    - Compute units are further divided into processing elements
    - OpenCL device schedule execution of instructions.

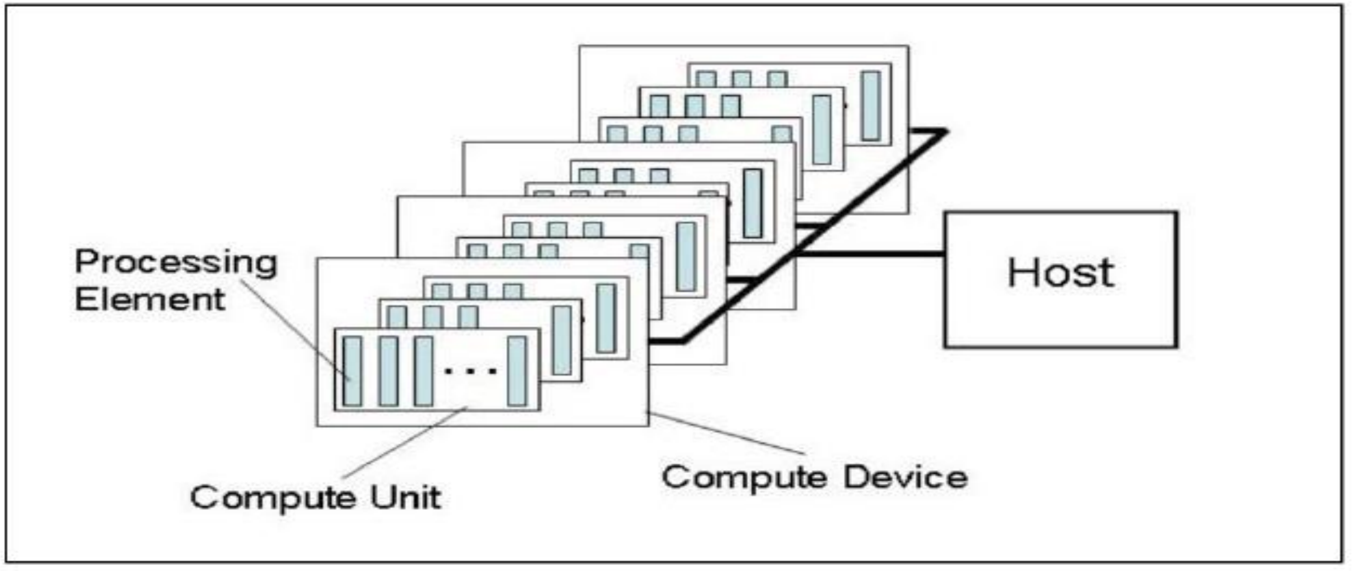
OpenCL Platform Model



The platform model defines an abstract architecture for devices.

- The host is connected to one or more devices
- Device is where the stream of instructions (or kernels) execute (an OpenCL device is often referred to as a **compute device**
- A device can be a CPU, GPU, DSP, or any other processor provided by Hardware and supported by the OpenCL Vendor

OpenCL Platform Model



- ❖ One Host + one or more compute Devices
  - ➢ Each compute Device is connected to one or more Compute Units.
    - Each compute Unit is further divided into one or more Processing Elements

Source : NVIDIA, Khronos AMD, References

OpenCL PLATFROM Model

How to discover available platforms for a given system ?

```
cl_int
ClGetPlatformIds(cl_unit num_entries,
                 cl_platform_Id *platforms,
                 cl_unit *num_platforms)
```

❖ Platform Model

- Defines a device as an array of compute units
  - Compute units are further divided into processing elements
  - OpenCL device schedule execution of instructions.

Source : NVIDIA, Khronos   AMD, References

OpenCL PLATFORM Model

How to discover available platforms for a given system.

❖ Application calls **`ClGetPlatformIds() twice`**

- The **first** call passes an **unsigned int** pointer as the **`num_platforms`** argument and NULL is passes as the **platform** argument.
  - The programmer can then allocate space to hold the platform information.
- The **second** call, a **`cl_platform_id`** pointer is passed to the implementation with enough space allocated for **`num_entries`** platforms.

Source : NVIDIA, Khronos AMD, References

OpenCL PLATFROM AND DEVICES

After platforms have been discovered, How to determine which implementation (vendor) the platform was defined by ?

The **`ClGetPlatformInfo()`** call determines implementation

The **`clGetDeviceIDs()`** call works very similar to **`ClGetPlatformId()`**

How to use `device_type` argument ?

GPUs : **`cl_DEVICE_TYPE_GPU`**

CPUs : **`cl_DEVICE_TYPE_CPU`**

All devices : **`cl_DEVICE_TYPE_ALL`** & other options

`Cl_GetDeviceinfo()` is called to retrieve information such as name, type, and vendor from each device.

Source : NVIDIA, Khronos AMD, References

OpenCL PLATFROM Model

After platforms have been discovered, How to determine which implementation (vendor) the platform was defined by ?

The **`clGetDeviceIDs()`**

```
cl_int
clGetDeviceIDs(cl_platform_id platform,
                  cl_DEVICE_TYPE_GPU device_type,
                  cl_unit num_entries,
                  cl_device_id *devices,
                  cl_uint *num_devices)
```

OpenCL PLATFORM Model

How to get printed information about the OpenCL, supported platforms and devices in a system ?

`CLinfo` prorgam in the AMD APP SDK

Uses **`clGetplatforminfo()`** and **`clGetDeviceInfo()`**

Hardware details such as memory size and bas widths are available using the commands

$ **`./CLinfo`** program gives complete information

OpenCL PLATFROM AND DEVICES

$ **./CLinfo**

Number of platforms :	1
Platform Profiles :	FULL_PROFILE
Platform Version :	OpenCL 1.1 AMD SDK –v2.4
Platform Name :	AMD Accelerated Parallel Processing
Platform Vendor :	Advanced Micro Devices, Inc.
Number of Devices :	2
Device Type :	CL_DEVICE_TYPE_GPU
Name :	Cypress
Max Compute Units :	20
Address bits	32

OpenCL PLATFROM AND DEVICES

$ **./CLinfo**

Max Memory Allocation:	268435456
Global Memory size :	1073741824
Constant buffer size :	65536
Local Memory type :	Scratchpad
Local Memory size :	32768
Device endianess :	little
Device Type :	CL_DEVICE_TYPE_CPU
Max Compute units :	16
Name :	AMD Phenom™ 11 X4 945 Processor

Source : NVIDIA, Khronos AMD, References

Part-III(D)

OpenCL Specification

Execution Model

(In brief)

The OpenCL Specification

❖ **Execution model :**

➢ Defines

- How the OpenCL environment is configured on the host
- How kernels are executed on device

➢ This includes

- Setting up an OpenCL context on the host,
- Providing mechanism for host-device interaction, &
- defining a concurrency model used for kernel execution on device
- The host sets up a kernel for the GPU to run and instantiates it with some special degree of parallelism.

Source : NVIDIA, Khronos AMD, References

The OpenCL Execution Model

❖ Execution Model

- ➢ Application consists of **two** distinct parts
- ➢ **The host program**
  - Runs on the host
  - OpenCL does not define the details of how the host progrma works, only how it interacts with objects defined in OpenCL
- ➢ **A Collection of Kernels**
  - The Kernel execute on the OpenCL device

Source : NVIDIA, Khronos AMD, References

The OpenCL Execution Model

❖ **Execution Model - Kernels**

➢ **A Collection of Kernels**

- Execute on the OpenCL device
- Do the real work of an OpenCL application
- Simple functions transform **input** memory objects into **output** memory objects

Execution Model - Kernels

➢ **OpenCL defines two types of Kernels**

- **OpenCL** Kernels & **Native** Kernels

Source : Khronous, & References

The OpenCL Execution Model

❖ **Execution Model : Defines how the kernels execute**

➢ **Several Steps Exist.**

- **FIRST :** How an individual kernel runs on an OpenCL device ?
- **Second:** How the host defines the **context** for kenrel execution
- **THIRD:** How the kernels are enqueued for execution

Source : NVIDIA, Khronos AMD, References

The OpenCL Execution Model

❖ **Execution Model - Kernels**

➢ **OpenCL Kernels**

- Written in OpenCL C programming language and compiled with the OpenCL Compiler
- All OpenCL implementations must support OpenCL Kernels

➢ **Native Kernels**

- Functions created outside of **OpenCL** and accessed within **OpenCL** through a function pointer. (An Optional functionality within in **OpenCL** exist)

Source : NVIDIA, Khronos AMD, References

The OpenCL Execution Model

- The OpenCL Execution Environment defines the following how the kernel execute
  - Contexts
  - Command Queues
  - Events
  - Memory Objects (Buffers -large array /images
    - Buffers (allocate buffer & return memory object)
    - Image (2D & 3D)
  - Flush & Finish

Source : NVIDIA, Khronos AMD, References

Part-III(E)

OpenCL Specification :Execution Model

How a Kernel Execute on an OpeCL Device

(In brief)

OpenCL Execution Model

❖ **OpenCL Program :**

- Kernels
  - Basic unit of executable – similar to a C function'
  - Data-parallel or task parallel
- Host Program
  - Collection of computer kernels and internal functions
  - Analogous to a dynamic library

Source : Khronous, & References

OpenCL Execution Model

- **Compute kernel**
  - Basic unit of executable code – similar to a C function
  - Data-parallel or task-parallel
- **Compute Program**
  - Collection of computer kernels and internal functions
  - Analogous to a dynamic library
- **Applications queue compute kernel execution instances**
  - Queued in-order
  - Executed in-order or out-of-order
  - Events are used to implement appropriate synchronization of execution instances

The OpenCL Execution Model

❖ **How a Kernel Execute on an OpeCL Device ?**

- **1.** A kernel defined on the Host

- **2. Issues a command :** The host program issues a command that submits the kernel for execution on an OpenCL device.

- **3. Creation of Integer index space :** The OpenCL runtime system creates an integer index space

- **4. Work-item :** An instance of the Kernel executes for each point in this index space and each such instance of an executing a kernel a **work-item**

- **Work-item** is identified by its coordinates in the index space & these coordinates are the global ID for the work-item.

Kernel Execution on an OpenCL Device

- OpenCL Approach :
  - The unit of concurrent execution in OpenCL is a ***work-item***
  - Map a single iteration of the loop to a ***work-item***
  - Tell the OpenCL runtime to generate as many ***work-items*** as elements in the input and output arrays
  - Allow the runtime to map those ***work-items*** to the underlying hardware i.e. CPU or GPU Cores in whatever way it views appropriate.

Source : NVIDIA, Khronos AMD, References

Kernel Execution on an OpenCL Device

- OpenCL implements hierarchy concurrent model
- OpenCL describes execution in fine-grained **work-items** and can dispatch vast number of **work-items** on architecture with hardware support for **fine-grained** threading
- When a kernel is executed, the programmer specifies the number of **work-items**
  - **Work-items** have unique global IDs from the index space
- **Work-items** are organized into **work-groups**. **Work-groups** have a unique work-group ID
- **Work-items** have a unique **local ID** within a **work-group**

Kernel Execution on an OpenCL Device

❖ Define N-Dimensional computation domain

- **Work-items** should be created as an n-dimensional range (NDRange)
- Each independent element of execution in N-D domain is called a **work-item**
- The N-D domain defines the total number of **work-items** that execute in parallel – global work size.
- The host program involves a kernel over an index space called an ***NDRange***
  - NDRange = "N-dimensional Range" & it can be a 1, 2 or 3-dimensional Range

Source : NVIDIA, Khronos AMD, References

Kernel Execution on an OpenCL Device

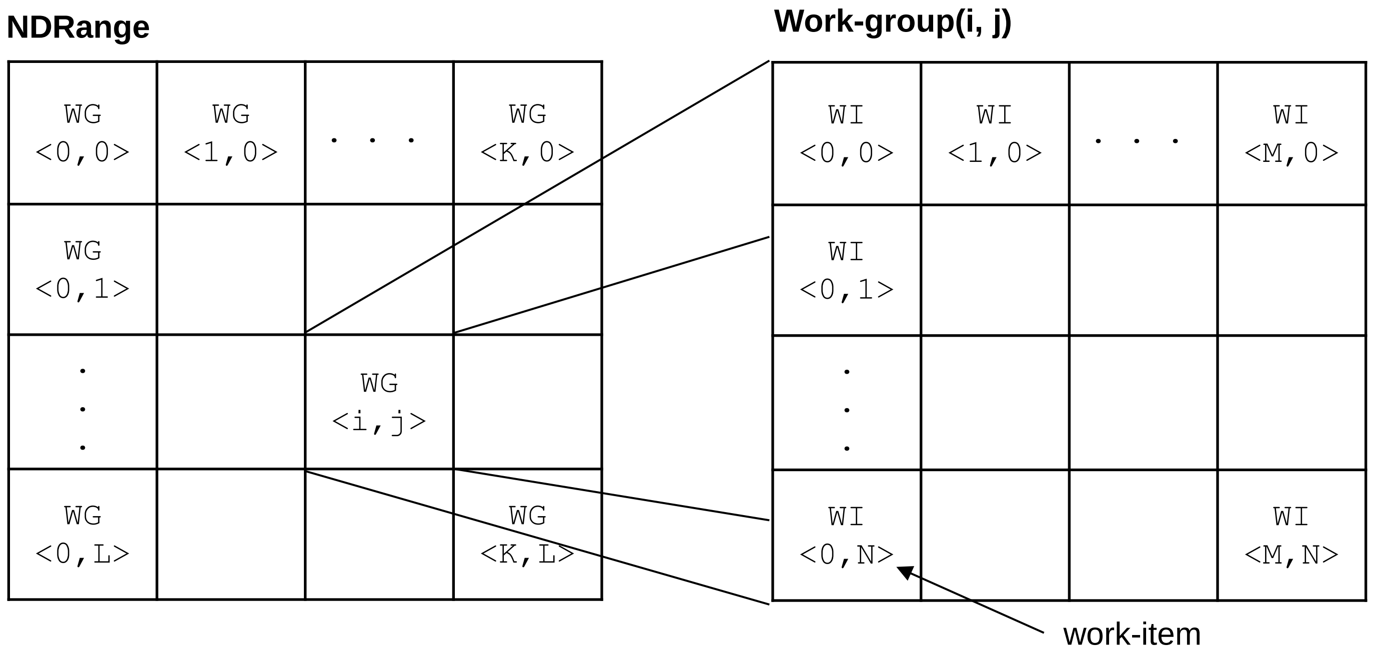
- Work-items can be grouped together – **work-group**
  - **Work-items** in **work-group** can communicate with each other
  - Can synchronize executing among **work-items** in group to coordinate memory access
- Execute multiple work-groups in parallel –
  - Provide more **coarse grained decomposition** of index space
- Mapping of global **work-size** to **work-groups**
  - Implicit or explicit

Source : NVIDIA, Khronos AMD, References

Kernel Execution on an OpenCL Device

Work-groups & Work-items

Scalability : Divide work-items of an **NDRange** into smaller, equally sized workgroups.



Work-items are created as an NDRange and grouped in workgroups.

An index space with **N** dimensions require work-groups to be specified using the same **N** dimensions : thus, a **three** dimensional index space requires **three**-dimensional work-groups.

Kernel Execution on an OpenCL Device

More about workgroups & work-items

- An **NDRange** is a *one-*, *two*-, or *three-* dimensional index space of **work-items** that will often map to the dimensions of either the input or the output data.

- The dimensions of the **NDRange** are specified as an **N**-element array of type **size_t** where **N** represents the number of dimensions used to described the **work-items** being created.

Kernel Execution on an OpenCL Device

- Kernels are the part of an OpenCL program that actually execute on a device. The OpenCL API
  - Enables an application to create a context for management of the execution of OpenCL commands, including those
    - describing the movement of data between and OpenCL memory structures and
    - the execution of kernel code that process this data to perform some meaningful task.
- The goal is often to represent parallelism programmability at the **finest granularity.**
- The generalization of the OpenCL interface and the lowest level kernel language **allows** efficient mapping to a wide range of hardware

Source : NVIDIA, Khronos AMD, References

Kernels and the OpenCL Execution Model

Work-groups & work-items

- ❖ Note that OpenCL requires that the index space sizes are evenly divisible by the work-group sizes in each dimension.
- ❖ For hardware efficiency, the work-group size is usually fixed to a favorable size
  - To satisfy the divisibility requirement, round-up the index space size in each dimension is required.
  - Specify the extra work-items in each dimension in such way that these extra items return immediately without outputting any data
  - Developer can pass “**NULL**” (implementation takes care-off)

Part-III(F)

OpenCL Specification :Execution Model

Context

(In brief)

OpenCL Execution Model : Contexts

- ❖ Kernels are defined on the **host** and host the establishes the **context** for the kernels.
- ❖ Host defines the "**NDRange**"
- ❖ Host defines the "**queues** " that control the details of how and when the kernels execute

(Important functions are defined in APIs within OpenCL's definition.)

Task : Host Defines the Context for the OpenCL Application

- ➢ The **context** defines the environment within which the kernels are defined and execute

OpenCL Execution Model : Contexts

❖ How to co-ordinate the mechanisms for host-device interaction ?

❖ How to manages the memory objects that are available on the device ?

❖ How to keep track of the programs and kernels that are created for each device ?

❖ Support of APIs

➢ Before a **host** can request that a kernel be executed on a device, **a context** must be configured on the **host.**

- Enables it to pass commands and data to the **device**

OpenCL Execution Model : Contexts

- **The API function to create a context is clCreateContext()**
- The **context** is an abstract container that exists on the **host.**
- A **context**
  - Coordinates the mechanisms for host-device interaction,
  - Manages the memory objects that are available to the devices
  - Keeps track of the programs and kernels that are created for each device.
- The properties argument is used to restrict the scope of the **context**
  - **Context** may provide a specific platform ,enable graphics interoperability, or enable other parameters in the future.

OpenCL Execution Model : Contexts

❖ A **context**

- The number and IDs of the devices that the programmer wants to associate with the context must be supplied.

Remark : In OpenCL, the process of discovering platforms and devices and setting up a context is tedious. However, after the code to perform these steps is written once, it can be reused or almost any project.

OpenCL Execution Model : Contexts

❖ **How context includes OpenCL Devices and a program object from which the kernels are pulled for execution ?**

❖ A **context** is defined in terms of the following resources **:**

- **Devices :** the collection of OpenCL devices to be used by the host
- **Kernels :** the OpenCL functions that run on the OpenCL device.
- **Program Objects :** the program source code and executable that implement the kernels
- **Memory Objects : :** a set of objects in memory that are visible to OpenCL devices and contain values that can be operated on by instances of a kernel.

OpenCL Execution Model : Contexts

- The context is created and manipulated by **host** using the functions from the OpenCL API
  - **On Heterogeneous platform,** the host may choose the GPU, other cores on the CPU, or combination of these.
  - Once the choice made, the choice defines the OpenCL devices within the current **context**
  - **Program Objects :** One of more program objects that contain the code for the kernels.
    - These can be thought as a " Dynamic library from which the functions used by a kernel are **pulled**.

OpenCL Execution Model : Contexts

❖ **More about Program Objects :**

➢ The program object is built at **runtime** within the host program

- Which target platform will be standard to OpeCL Specification ?

- How de we specify this information in host program ?

➢ Built the program object from the **source** at runtime.

- Compile the program source code to create the code for kernel. (The host program defines devices within the context)

OpenCL Execution Model : Contexts

- **More about Program Objects :**

More about Source Code :

- Regular String either statistically defined in the host program
- Loaded from a file at runtime
- Dynamically generated inside the host program

- **Context includes OpenCL Devices and a program object from which the kernels are pulled for execution**

OpenCL Execution Model : Contexts

- **More about Program Objects :**

More about Source Code :

- Regular String either statistically defined in the host program
- Loaded from a file at runtime
- Dynamically generated inside the host program

- **Context includes OpenCL Devices and a program object from which the kernels are pulled for execution**

OpenCL Execution Model : Contexts

```
clCreateContext(
   const cl_context_properties *properties,
   cl_unit num_devices,
   const cl_Device_id *devices,
   void (CL_CALLBACK *pfn_notify) (
                 const char *errinfo,
                const void *private_info
                size_t cb,
                 void  *user_data)
    void *user_data,
   cl_int *errcode_ret}
```

OpenCL Execution Model : Context

- **"Context"; How the OPenCL Kernels works with memory ?**
- What is needed for **Command queue** ?
- Detailed memory model needs to be understand and How the openCL memory works at higher level ?
  - About Heterogeneous Systems :
    - Multiple Address Spaces to manage
  - OpenCL introduced the concept of Memory Object
    - Explicitly defined on the host
    - Explicitly moved between the host and the OpenCL devices

OpenCL Execution Model : Contexts

- The OpenCL specification also provides an API call tat alleviates the need to build a list of devices.
  - **`clCreateContextFromType()`** allows a programmer to create a context that automatially includes all devices of the specified type (e.g., CPUS, GPUs, and all devices)
  - After a creating a context, the function **`clGetContextinfo()`** can be used to query information such as the number of devices present and device structures.
- In OpenCL, the process of discovering platforms and devices and setting up a **`context`** is tedious. However, after the code to perform these steps is written once, it can be reused or almost any project.

OpenCL Execution Model : Context

A brief summary of OpenCL Context

❖ Context is the

- OpenCL Devices
- Program Objects
- Kernels
- Memory Object

that a kernel uses when it executes

Command-Queues :

How the host program issues commands to the OpenCL devices ?

OpenCL Execution Model : Context

A brief summary of OpenCL Context

- ❖ Context is the heart of any OpenCL application
- ❖ Context provide a container for
  - ➢associating devices,
  - ➢Memory Objects (e.g., buffers and images),
  - ➢command-queue (providing interface between the context and an individual object)
- ❖ Context drives the communication with, and between, specific drives and OpenCL defines it memory model in terms of these

OpenCL Execution Model : Context

A brief summary of OpenCL Context

- ❖ Example : A memory object is allocated with a context but can be updated by a particular device, and OpenCL/memory guarantees that all devices, within the same context, will see these updates as well defined synchronizing points

- ❖ Context – update as the program progresses, allocating or deleting memory objects and so on.

  - ➢associating devices,

  - ➢Memory Objects (e.g., buffers and images),

  - ➢command-queue (providing interface between the context and an individual object)

OpenCL Execution Model : Context

In general, an application's OpenCL Usage look similar to this Context

1. Query which platforms are present
2. Query the set of devices supported by each platform
   a. Choose the select devices, using **clGetDeviceInfo(),** on specific capabilities
3. Create contexts from a selection of devices (each context must be created with devices from a single platform), then with a context you can

OpenCL Execution Model : Context

In general, an application's OpenCL Usage look similar to this Context

3. Create contexts from a selection of devices (each context must be created with devices from a single platform), then with a context you can

   a. Create one or more command-queues

   b. Create programs to run on one or more associated devices

   c. Create a kernel from those programs

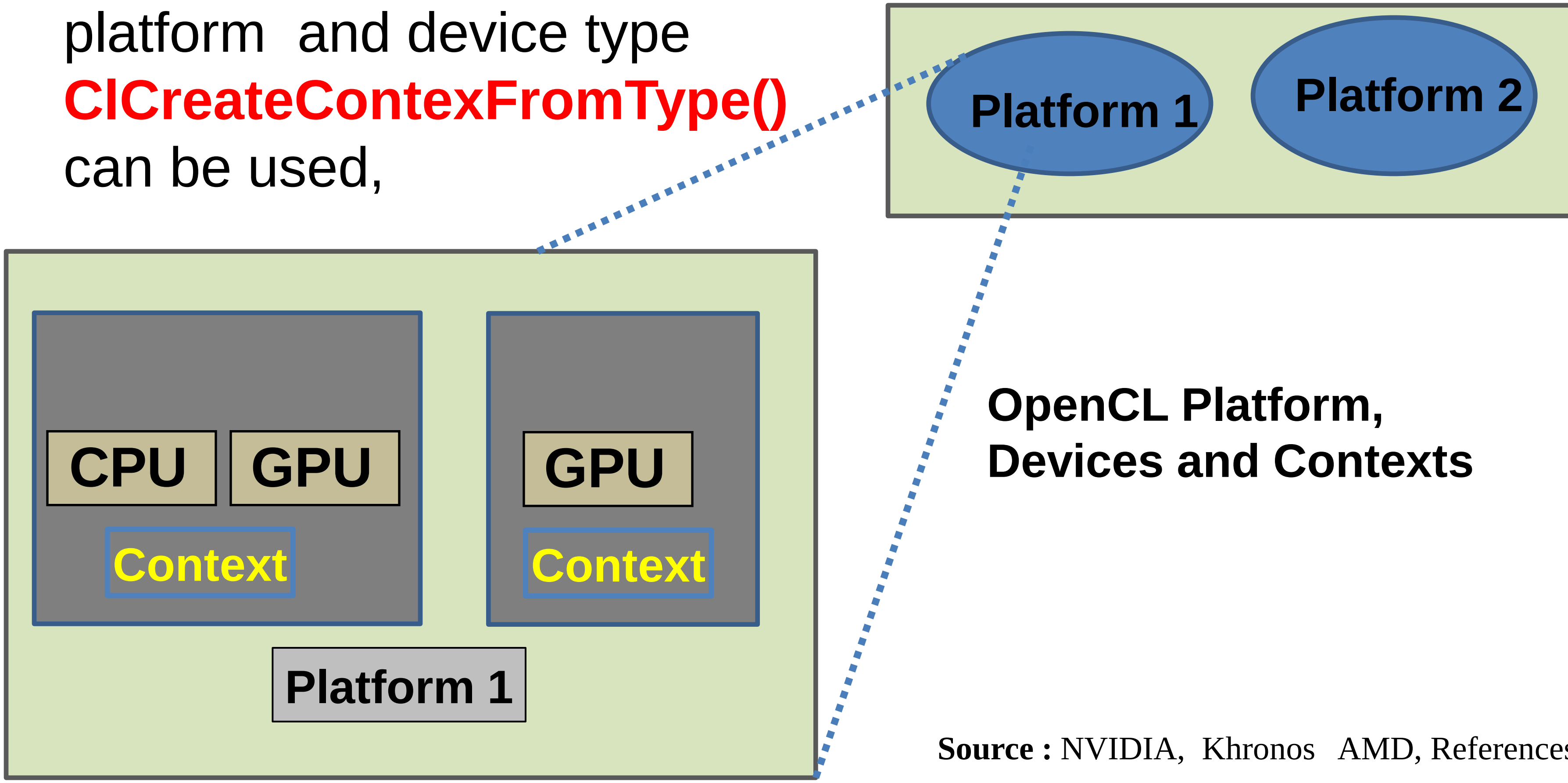
   d. Allocate memory buffer and images either on the host or on the device

   e. Write or copy data to and from a particular device

   f. Submit kernels (setting the appropriate arguments to a command-queue for execution

OpenCL Execution Model : Context

❖ Given a platform and a list of associated devices, an OpenCL context is created with the command **ClCreateContext(),** and with a platform and device type **ClCreateContexFromType()** can be used,



OpenCL Platform, Devices and Contexts

Source : NVIDIA, Khronos AMD, References

Part-III(G)

OpenCL Specification :Execution Model Command-Queues

(In brief)

Source : NVIDIA, Khronos AMD, References

What is command-queues ?

- The interaction between the host and the **OpenCL** devices occurs through commands posted by a host to the **command-queue.**

- These commands wait in the command-queue until they execute on the OpenCL device

- Check for successful completion of "definition of the context" A command-queue is created by the host and attached to a **single** OpenCL device after the context has been defined.

OpenCL Execution Model : Command-Queues

About command-queues

- ❖ The host places commands into the command-queue, and commands are then scheduled for execution on the associated device. OpenCL supports three types of commands :

- ❖ **Kernel Execution commands :** executes a kernel on the processing elements of an OpenCL device

- ❖ **Memory commands :** transfer data between the host and different memory objects move data between memory objects, or map and unmap memory objects from the host address space.

- ❖ **Synchronization commands :** put constraints on the order in which commands execute.

OpenCL Execution Model : Command-Queues

❖ **About** **command-queues**

➢ Mechanism that the **host** uses to request action by the **devices**.

➢ Communication with a device occurs by submitting commands to a **command-queue**.

➢ Each **command-queue** is associated with only one device

- **Step 1 : Host** decides which **device** to work with
- **Step 2 :** A **context** is created
- **Step 3 :** One **command-queue** needs to be created per device

➢ Whenever the host needs an action to be performance by a device, it will submit commands to the proper command queue.

OpenCL Execution Model : Command-Queues

About command-queues

The API **`clCreateCommandQueue()`** is used to create a command queue and associate it with a device.

```
Cl_Command_queue
clCreateCommandQueue(
    cl_context context,
    cl_device_id device,
    cl_command_queue_properties properties
    cl_int*  errcode_ret)
```

OpenCL uses default **`in-order command queue`**

If **`out-of-order`** queues are used, it is up to the user to specify dependencies that enforce a correct execution order.

OpenCL Execution Model : Command-Queue

❖ **About command-queue**

➢ Any API that specifies **host-device** interaction will always begin with **clEnqueue** and require a command queue as a parameter.

➢ For ex :

- the **ClEnqueueReadBuffer()**command requests that the device send data to the host and
- **clEngueueNDRangeKernel()** requests that a kernel is executed on the device.

OpenCL Execution Model : Command-Queues

❖ **Remarks : context & command-queue**

- **First Step - Context :** The programmer defines the context and the command-queues, defines memory and the program objects

- The programmer builds any data structures needed on the host to support the application

- **Next Step - Command queue :**
  - Memory objects are moved from host onto the devices
  - Kernel arguments are attached to memory objects and then submitted the command-queue for execution

OpenCL Execution Model : Command-Queues

❖ **Remarks : context & command-queue**

➢ **Next Step - Command queue :**

- When the kernel has completed its work, memory objects produced in the computation may be copied back on the host.

❖ **Other Information : command-queue**

❖ What is the order in which the commands execute ?

❖ How the commands execution relates to the execution of the host program. ?

OpenCL Execution Model : Command-Queue

Other Information : command-queue

❖The commands always execute asynchronously to the host program

❖The host program submits commands to the command-queue and then continue without waiting for a commands to finish

- ❖ If necessary, for the host to wait on a command, this can be explicitly established with a synchronization

❖Commands within a single queue execute relative to each other in one of the two modes :

- ❖ In-order execution & Out-or-order execution

OpenCL Execution Model : Command-Queues

Other Information : command-queue

❖ Errors : Multiple executions occurring in-side an application may lead to potential disaster i.e. abnormal exist with error messages

- Data may be accidently used before it has been written or kernels may be execute in an order that leads to wrong answers.

❖ The programmer needs some way to manager any constraints on the commands.

❖ Synchronization commands can be used to tell set of kernels to wait until an earlier set finishes.

OpenCL Execution Model : Command-Queue

Other Information : command-queue

- To support custom synchronization protocols, commands submitted the **command-queue** generate event objects.
- A command can be told to wait until certain conditions on the event object exists.
- It is possible to associate multiple queues with a single queues with a single context for any of the OpenCL devices within that context,
  - These two queues rub concurrently and independently with no explicit mechanism within OpenCL to synchronize between them.

Source : NVIDIA, Khronos   AMD, References

OpenCL PLATFORM AND DEVICES: Events

❖ What is an event ?

- Any operation that **`enqueues`** a command into a command queue – that is any API call the begins with **`clEnqueue`** – produces an ***event***. Events have two main roles to OpenCL

  1. Representing dependencies
  2. Providing a mechanism for profiling

- API Calls the begin with **`clEnqueue`** also take a "**wait list**" of events as a parameter.

- By generating an event for one API call and passing it as an argument to a successive call, OpenCL allows us to represent dependencies.

- A **`ClEnqueue`** call will block until all events in its wait list have completed.

OpenCL : Specification : Heterogeneous Prog.

The Execution Environment

- Contexts
- Command Queues
- Events
- Memory Objects (Buffers -large array /images
  - Buffers (allocate buffer & return memory object)
  - Image (2D & 3D)
- Flush & Finish

Part-III(H)

OpenCL Specification :

Memory Model

The OpenCL Specification

❖ **Memory model :**

- Defines the abstract memory hierarchy that kernels use, regardless of the actual underlying memory architecture
- The memory model closely resembles current GPU memory hierarchies. Other accelerators has no limited adoptability.
- To support code portability, OpenCL's approach is to define an abstract memory model that programmers can target when writing code and vendors can map to their actual memory hardware
- The memory spaces (*global memory, constant memory, local memory, private memory*) defined by OpenCL are used and are relevant within OpenCL programs.
- The memory spaces of OpenCL closley model those of modern GPUs

Source : NVIDIA, Khronos AMD, References

OpenCL : Specification : Heterogeneous Prog.

- **OpenCL Memory Model defines five distinct memory-regions**
  - Host memory
  - Global memory
  - Constant Memory
  - Local Memory
  - Private Memory
- **OpenCL Writing kernels**
  - Kernels begin with the keyword **_kernel** and must have a return type of **void**.

Source : NVIDIA, Khronos AMD, References

OpenCL : Specification : Execution Model

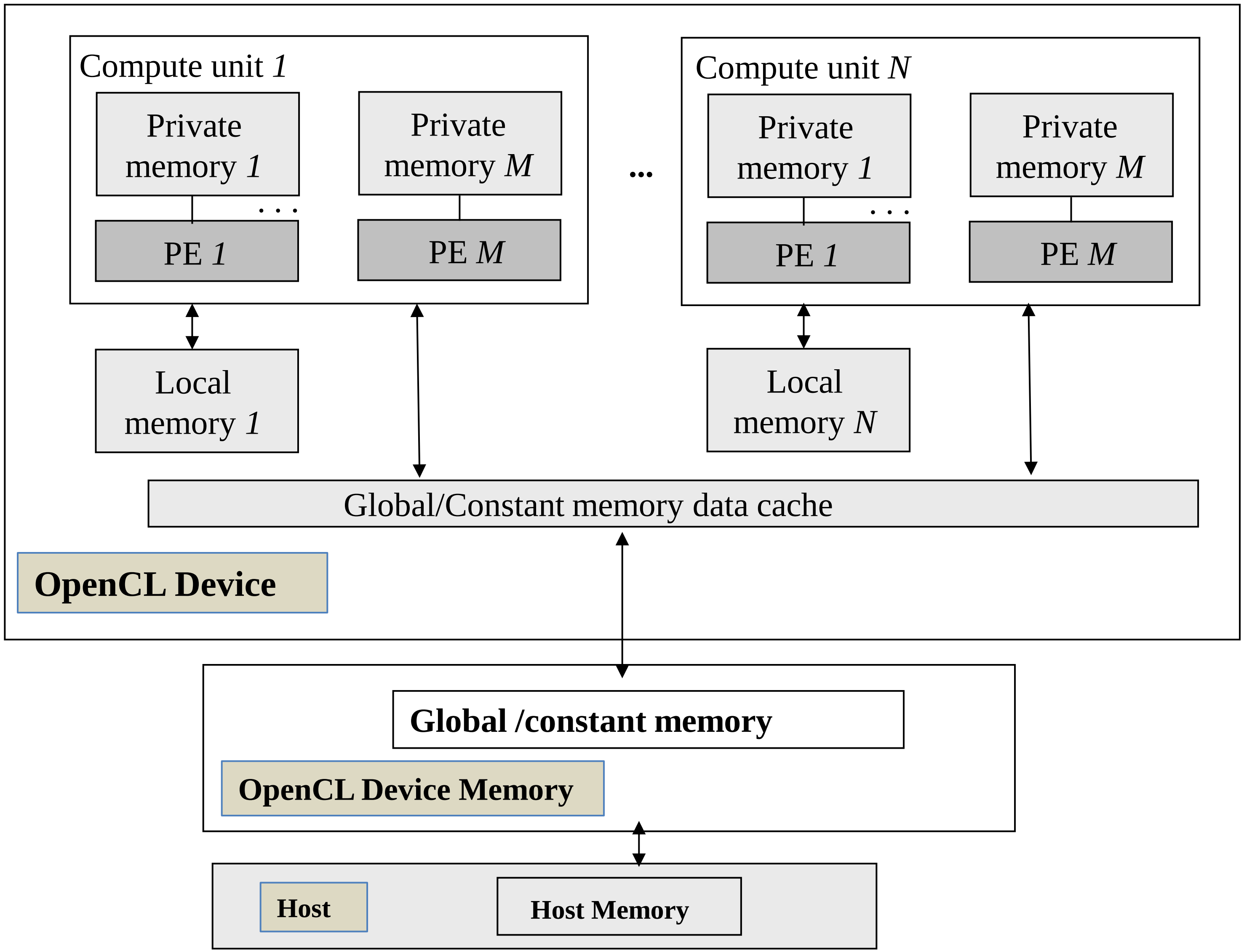
- **The Execution model tells**
  - How the kernel executes ?
  - How they interact with other kernels ?
- **Used "Memory Objects" for an associated command-queue**
  - How safe these memory objects can be used ?
- OpenCL defines two types of memory objects
  - Buffer Object
  - Image Object
- OpenCL – specify sub regions of memory objects as distinct memory objects

OpenCL : Specification : Heterogeneous Prog.

- **OpenCL Memory Model defines five distinct memory-regions**
  - Host memory
  - Global memory
  - Constant Memory
  - Local Memory
  - Private Memory

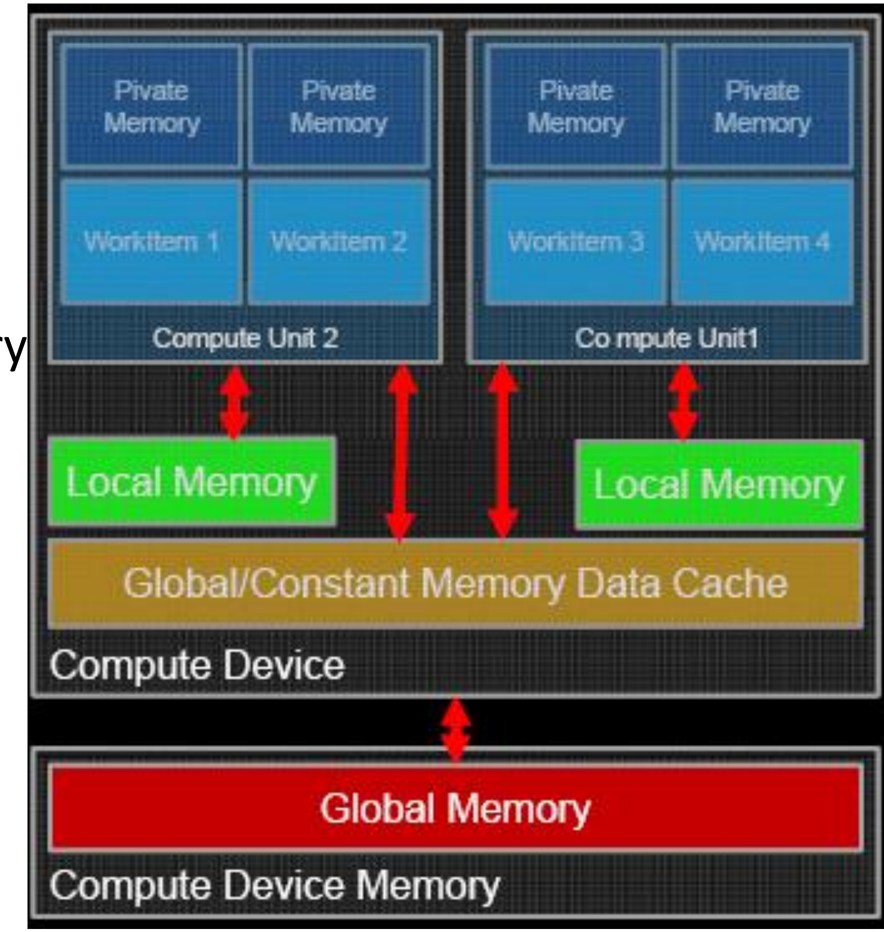
- **OpenCL Writing kernels**
  - Kernels begin with the **keyword _kernel** and must have a return type of **void**.

A summary of memory model to OpenCL



OpenCL Memory Model

- ❖ Implements a relaxed consistency, shared memory model
- ❖ Multiple distinct address spaces
  - ➢ Address spaces can be collapsed depending on the device's memory subsystem
  - ➢ Address qualifiers
  - ➢ _private
  - ➢ _local
  - ➢ _constant and_global
  - ➢ Example:
    - _global float4 *p;



Source : Khronos, References

OpenCL : Memory Model

The OpenCL : `Abstract Memory Model Defined`

- OpenCL'S approach is to define an abstract memory model

  - Programmers can target when writing code
  - Vendors can map to their actual memory hardware
  - The memory spaces defined by OpenCL :

    - Global Memory
    - Constant Memory
    - Local Memory
    - Private Memory

  - The key words associated with each space can be used to specify where a variable should be created or where the data that it points to resides.

Source : NVIDIA, Khronos AMD, References

The OpenCL : `Abstract Memory Model Defined`

❖ **Global Memory :**

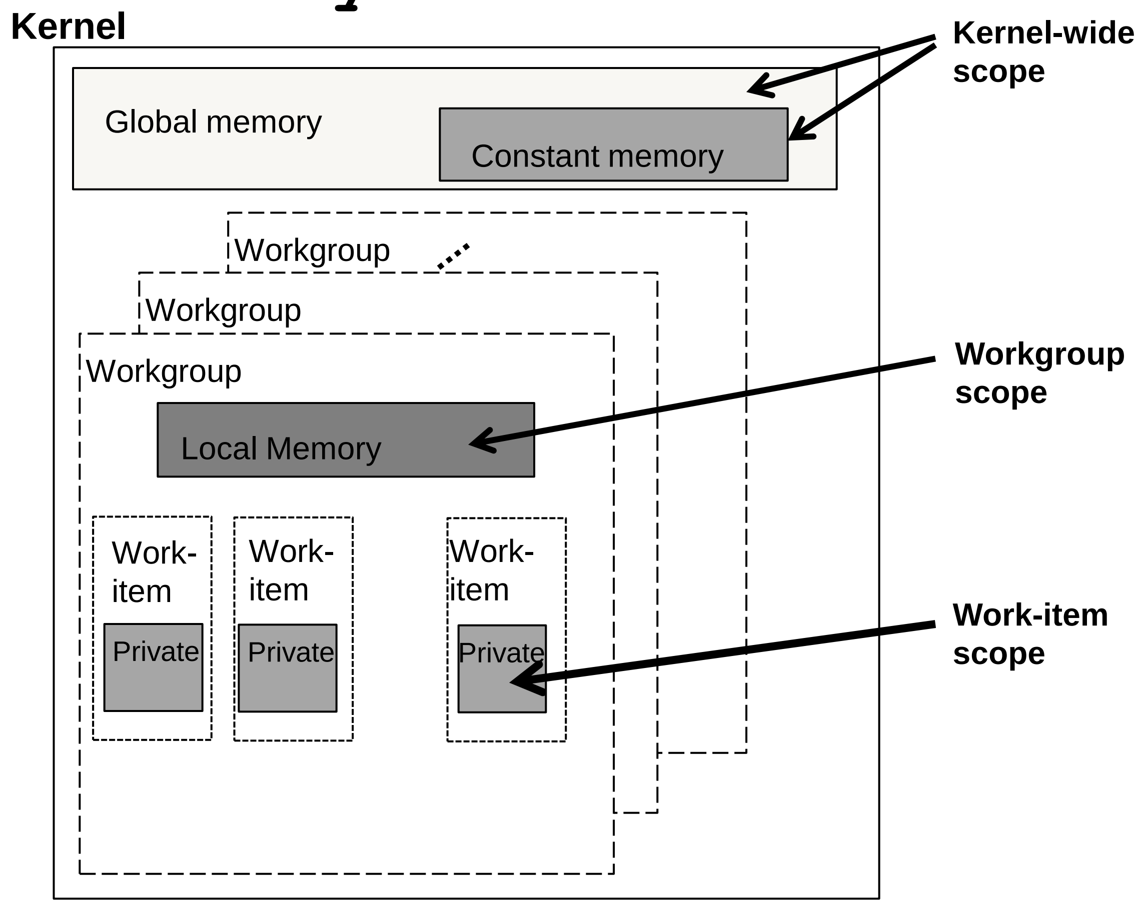
- Visible to all compute units on the device.
- Whenever the data is transferred from the host to device, the data will resides in global memory.
- And data transfer from the device to host must also reside in global memory :
  - The key-word **`__global`** is added to a pointer declaration to specify that data retrenched by the pointer, resides in global memory,

OpenCL : Memory Model

The OpenCL : `Abstract Memory Model Defined`

- Global Memory
- Constant Memory
- Local Memory
- Private Memory

Usually, the memory spaces of openCL closely model those of modern GPUs.



The abstract memory model defined by OpenCL.

Source : NVIDIA, Khronos AMD, References

OpenCL : Memory Model

The OpenCL : `Abstract Memory Model Defined`

❖ **Constant Memory :**

- Not specifically designed for every type of read-only data but, rather, for data where each element is accessed simultaneously by all **`work-items`**.
- Variables whose values never change also fall in the category.
- Constant memory is modeled as apart of global memory, so memory objects that are transferred to global memory can be specified as constant.
  - Data is mapped to constant memory by using the key-word **`__constant`**.

Source : NVIDIA, Khronos AMD, References

OpenCL : Memory Model

The OpenCL : `Abstract Memory Model Defined`

❖ **Local Memory :**

- Scratchpad memory whose address space is unique to each compute device :
- Local memory is modeled as being shared by a workgroup.
- Variables whose values never change also fall in the category.
  - Calling **`clSetKernelArg() with a size,`** but no argument allows local memory to be allocated at runtime, where a kernel parameter is defined as a **`__local pointer.`**
  - Data is mapped to constant memory by using the key-word **`__constant`**.
- Arrays can also be declared statically in local memory by appending the keyword **`_local`**, although this require specifying that array size at compile time.

OpenCL : Memory Model

The OpenCL : `Abstract Memory Model Defined`

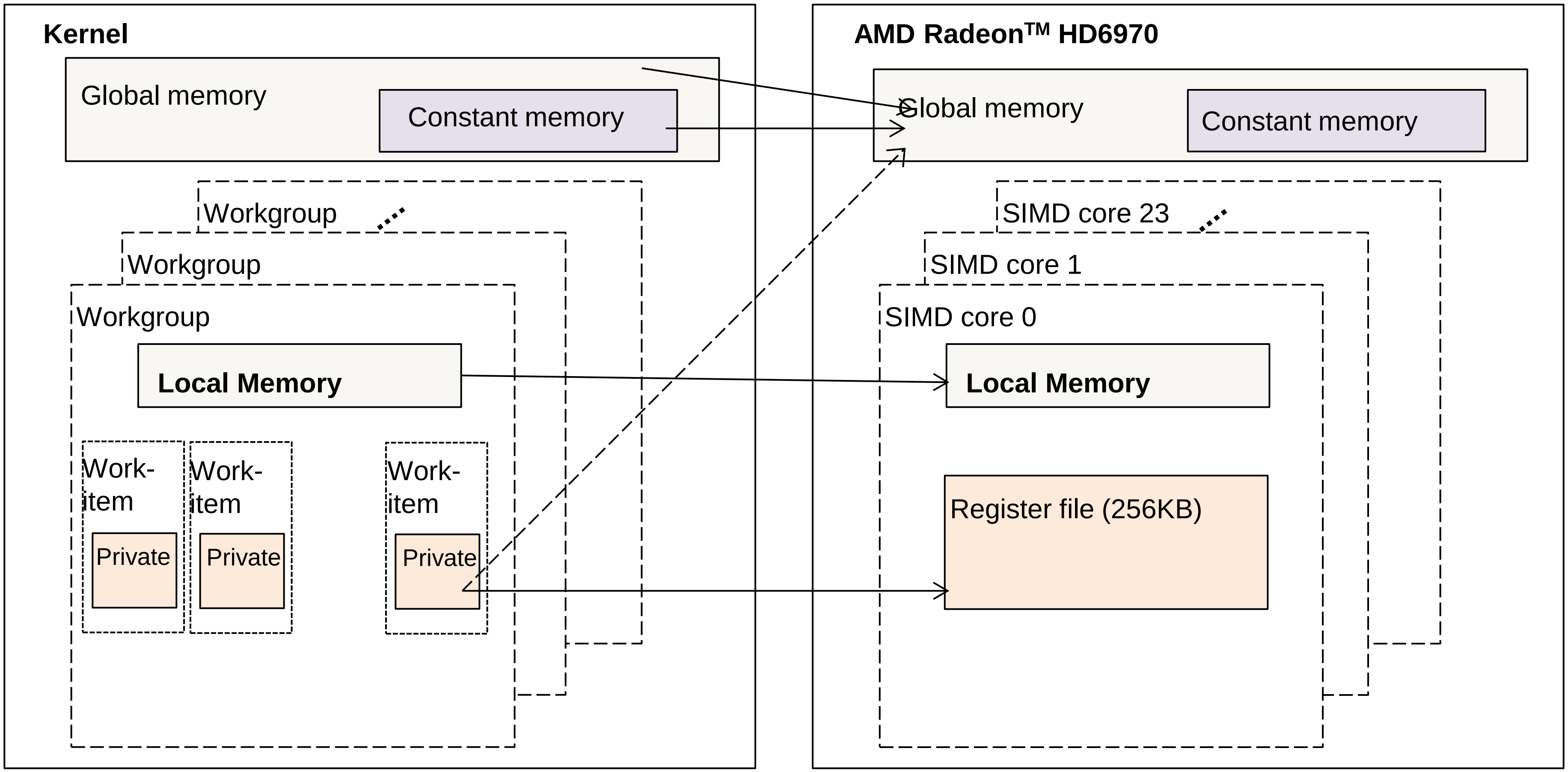
❖ **Private Memory :**

- Memory unique to an individual **`work-item.`**
- Local variables and non-pointer kernel arguments are private by default.
  - These variable are mapped to registers.

Source : NVIDIA, Khronos AMD, References

OpenCL : Memory Model

The OpenCL : Abstract Memory Model Defined



Mapping from the memory model defined by OpenCL to the architecture of an AMD Radeon 6970 GPU. Simple private memory will be stored in registers; complex addressing or excessive use will be stored in DRAM.

OpenCL : Writing Kernels

- OpenCL C kernels are similar to C functions and will be executed once for every work-item that is created. :
  - Buffers can be declared in global memory (**`_global`**) or constant memory **(`_constant`)** memory.
  - Images are assigned to global memory. Access qualifiers (**`_read_only`**, **`_write_only`**, and **`_read_write`**) can also be optimally specified
  - The **`__local`** qualifier is used to declare memory that is shared between all **`work-items`** in a **`workgroup`**.
  - Declare local memory allocations can be done differently using kernel-scope level..

OpenCL : Writing Kernels

- OpenCL devices, particularly GPUs, performance vary increase by using local memory to cache data that will be used multiple times by a **`work-item`** or by multiple **`work-items`** in the same workgroup.

- When developing a kernel, we can achieve this with an explicit assignment from a global memory pointer to a local memory pointer.

- Once **`work-item`** completes its execution, none of its state information or local memory storage is persistent.

- Any results that need to be kept must be transferred to global memory.

Part-III(I)

OpenCL Specification :

Memory Objects

Source : NVIDIA, Khronos AMD, References

OpenCL PLATFORM AND DEVICES: Memory Objects

❖ **Memory Objects**

- OpenCL applications often work with large arrays on multi-dimensional matrices. This data needs to be physically present on a device before execution can begin

1. First Step : Data must be encapsulated as a ***memory object***
2. Second Step : transfer the data to a device

- OpenCL define two types of memory objects
- **`clEnqueue`** also take a "**wait list**" of events as a parameter.

1. **Buffers** : equivalent to arrays in C, created using **malloc(),** where data elements are stored contiguously in memory.
2. **Images** : Designed as opaque objects, allowing data for padding and other optimizations that may improve performance on devices.

OpenCL PLATFORM AND DEVICES: Memory Objects

❖ **Memory Objects :**

➢ Memory object is valid only within a simple context, after creation of memory object.

1. To satisfy, the data Dependencies, **OpenCL** runtime manages movement to and from specific devices.

❖ **Memory Objects : Buffers**

➢ Buffers may help to visualize a memory object as a pointer that is valid on a device. (similar to call to malloc, in C or C++'s a new pointer

➢ The function **`clCreateBuffer()`** allocates the buffer and returns a memory object

❖ Memory Objects : Buffers

- Buffers may help to visualize a memory object as a pointer that is valid on a device. (similar to call to malloc, in C or C++'s a new pointer
- The function **`clCreateBuffer()`** allocates the buffer and returns a memory object
- Creating a buffer requires supplying the size of the **buffer** and a **context** in which the **buffer** will be allocated
- Buffer is visible for all devices associated with the context.
- Supply flags : Optionally, the caller can supply flags that specify that the data is read-only, write-only or read-write.

OpenCL PLATFORM AND DEVICES: Memory Objects

Memory Objects : Buffers

```
Cl_memclCreateBuffer(
    cl_context context,
    cl_mem_flags flags,
    Size_t size,
    void *host_ptr,
    cl_int *errcode_ret)
```

❖ **Memory Objects : Buffers**

- Supply flags : Creating and initializing a buffer with other flags (simple option is to supply a host pointer with data used to initialize the buffer)

OpenCL PLATFORM AND DEVICES: Memory Objects

- **Memory Objects : Buffers**
  - Data contained in host-memory is transferred to and from an OpenCL buffer using the command
    - **`ClEnqueueWriteBuffer()  and`**
    - **`ClEnqueueReadBuffer()`**

- Run-time determines the precise time the data is moved**.**
    - The buffer is linked to a context, not a device
    - If a kernel that is dependent on such a buffer is executed on a discrete accelerator device such as a GPU, the buffer may be transferred to the device.

OpenCL PLATFORM AND DEVICES: Memory Objects

Memory Objects : Buffers

```
Cl_int
clEnqueueWriteBuffer(
    cl_command_queue command_queue,
    cl_mem buffer,
    Cl_bool blocking_write,
    size_t  offset,
    Size_t cb
    const void *ptr,
    cl_unit num_events_in_wait_list,
    const cl_event *event_wait_list,
    cl_event *event)
```

OpenCL PLATFORM AND DEVICES: Memory Objects

❖ Memory Objects : Buffers

- Similar to other **`enqueue`** operations, reading or writing a buffer requires a command queue to manage the execution schedule.
- The **`enqueue`** function requires the buffer, the number of bytes to transfer, and an offset within the buffer.
- The **`block_write`** option should be set to **`CL_TRUE`** if the transfer into an openCL buffer until the operation has completed.
- Setting the **`block_write`** option to CL_FALSE allows **`clEnqueue-WriteBuffer`** to return before the write to **`CL_FALSE`** allows **`clEnqueueWriteBuffer()`** to return before the write operation has completed.

OpenCL PLATFORM AND DEVICES: Memory Objects

❖ **Memory Objects : Images**

- Images are type of OpenCL memory object that abstract the storage of physical data to allow for devices-specific optimization

- Use **`clGetDeviceInfo()`** to check the support of all OpenCL Devices.

- **Purpose of using Images** : to allow the hardware to take advantage of spatial locality and to utilize the hardware acceleration available on many devices.

  - Unlike buffers, images cannot be directly referenced as if they were arrays.

Source : NVIDIA, Khronos AMD, References

OpenCL PLATFORM AND DEVICES: Memory Objects

❖ **Memory Objects : Images**

- Images are type of OpenCL memory object that abstract the Images are an example of the OpenCL standard being dependent on the underlying hardware of a particular device.

- **Th**e elements of an image are represented by a format descriptor **`(cl_image_format)`**.

- The format descriptor specifies how the image elements are stored in memory based on the concepts of **channels**

  - The channels order specifies the number of elements that make up an image element (up to four elements, based on the traditional use of RGBA pixels), and the channel type specifies the size of each element.

  - These elements can be sized from 1 to 4 bytes and in various different formats (e.g., integer or floating point)

OpenCL PLATFORM AND DEVICES: Memory Objects

❖ Memory Objects : Images

- Creating an OpenCL image is done using the command (**`clCreateImage2D()`** or **`clCreateImage3D()`**
- Additional arguments are required when creating an image object versus those specified for creating a buffer.
  - First, the height and the width of the image must be given (and a depth for the three-dimensional case)
  - Image pitch (number of bytes between the start of one image and the start of the next.) may be specified if initialization data is provided.
- Additional parameters are required when reading or writing an image.
- Within a kernel, images are accessed with built-in functions specific to data type.

OpenCL PLATFORM AND DEVICES: Memory Objects

Memory Objects : Images

```
Cl_mem

clCreateImage2D(

    cl_context context,

    cl_mem_flags flags,

    const cl_image_format *image_format

    size_t  image_width,

    Size_t image_height,

    const image_row_pitch,

    void *host_ptr

    cl_int *errcode_ret,
```

OpenCL : Specification : Heterogeneous Prog.

- **Creating an OpenCL Program Object**
  - Process of creating a kernel (Character string, Character array, Program object
  - Intermediate OpenCL –ICD; NVIDIA –PTX, AMD-IL
  - Final and Intermediate representations

OpenCL : Specification : Heterogeneous Prog.

- **OpenCL Kernel**
  - Get kernel object
  - Execute kernels on a device
  - Extract a kernel from a program
    - To request from the compiled program object

Source : NVIDIA, Khronos AMD, References

Part-III(J)

OpenCL Specification :

Details on…. on Programming Model

The OpenCL Specification

- The OpenCL specification is defined in four parts, called models, that can be summarized as follows.

  - Platform Model
  - Execution Model
  - Memory Model
  - Programming Model

Source : Khronous, & References

The OpenCL Specification

❖ Programming model :

- Defines how the concurrency model is mapped to physical hardware. The hardware thread contexts that execute the kernel must be created and mapped to actual GPU hardware units.
- OpenCL C code (Written to run on an OpenCL device) called a ***program.*** A program is a collection of functions called ***kernels***, where kernels are units of execution that can be scheduled to run on a device
- OpenCL software links only to a common runtime layer (called the ICD); & uses dynamic library interface at runtime
- Compiled at runtime through a series of API calls (The source code is turned into a program object (**OpenCL program object**) & then compiled to the generate the **OpenCL Kernel object** that can be used to execute kernels on a device.

Source : Khronous, & References

The OpenCL Specification

❖ **Programming model :**

- The data within the **kernel** is allocated by the programmer to specific parts of an abstract memory hierarchy.
- The runtime and driver will **map** these abstract memory space to the physical memory.
- The hardware threads contexts that execute the kernel must be created and mapped to actual GPU hardware units
- Executing a kernel requires dispatching it through an **enqueue** function.

Source : Khronous, & References

The OpenCL Specification

❖ Programming model :

➢ The process of creating kernel involves three steps.

- **Step 1 :** The OpenCL source code is stored in a character string. If the source code is stored in a file on a disk, it must be read into the memory and stored as a character array.
- **Step 2 :** The source code is turned into a object, **`cl_program`**, by calling **`clCreateProgramWithSource()`**.
- **Step 3 :** The program object is then compiled, for one or more OpenCL devices, with **`clBuildPorgram()`**, If there are compile errors, they will be reported here.

➢ OpenCL provides APIs which takes a list of binaries that matches the device list.

Source : Khronous, & References

Important Steps in OpenCL Implementation

Source : NVIDIA, Khronos AMD, References

OpenCL Implementation Steps

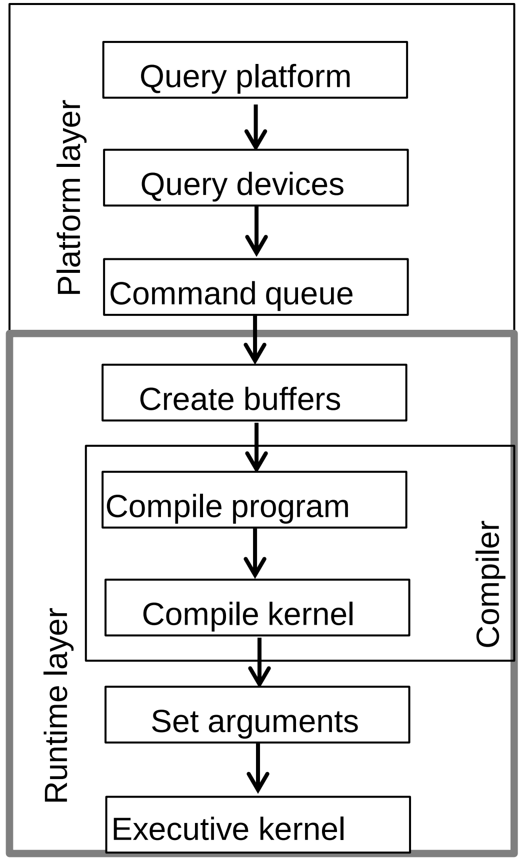


Figure 4.2 Programming steps to writing a complete OpenCL applications

OpenCL Important Steps – Implementation

Step 1 : Discover and initialize the platforms
Step 2 : Discover and initialize the devices
Step 3 : Create context
Step 4 : Create a command queue
Step 5 : Create device buffers
Step 6 : Write host data device buffers
Step 7 : Create and compile the program
Step 8 : Create the kernel
Step 9 : Set the kernel arguments
Step 10 : Configure the work -items structure
Step 11 : Enqueue the kernel for execution
Step 12 : Read the output buffer back to the host
Step 13 : Release OpenCL resources

OpenCL Important Steps – Implementation

Step 1 *: Discover and initialize the platforms*

Step 2 *: Discover and initialize the devices*

Step 3 *: Create context*

Step 4 *: Create a command queue*

Step 5 *: Create device buffers*

Step 6 *: Write host data device buffers*

The OpenCL specification in four parts, called models.

- **Platform Model**
- **Execution Model**
- **Memory Model**
- **Programming Model**

OpenCL Important Steps – Implementation

Step 7 : *Create and compile the program*

Step 8 : *Create the kernel*

Step 9 : *Set the kernel arguments*

Step 10 : *Configure the work -items structure*

Step 11 : *Enqueue the kernel for execution*

Step 12 : *Read the output buffer back to the host*

Step 13 : *Release OpenCL resources*

The OpenCL specification in four parts, called models.

- **Platform Model**
- **Execution Model**
- **Memory Model**
- **Programming Model**

OpenCL Important Steps – Implementation

- Create an OpenCL context on the first available device
- Create a command –queue on the first available device
- Load a kernel file (hello-world.cl) and build it into a program object
- Create a kernel object for the kernel function hello_world()
- Query the kernel for execution
- Read the results of the kernel back into the result buffer

OpenCL Important Steps – Implementation

```
_kernel void hello_kernel(_global *, *, )
{
     int gid = get_global_id(0);
     .........
 }

int main (int argc,  char** argv)
{
// Create an OpencL context on first available platform

// Create an command-queue on the first device
//  available on the created context
```

OpenCL Important Steps – Implementation

// Create OpenCL kernel

// Create memory objects that will be used as
// arguments to kernel.
// First create Host memory arrays that will be used to
// store the arguments to the kernel

// Set the kernel arguments

//Queue the kernel up for execution across the array
//Read the output buffer back to the Host
//Output the result buffer

OpenCL PLATFORM AND DEVICES: Flush & Finish

- The flush and finish commands are two different types of barrier operations for a command queue.
- The **`clFinish()`** function blocks until all of the commands in a command queue have completed.
- The **`clFlush()`** function blocks until all of the commands in a command queue have been removed from the queue.

```
cl_int clFlush(cl_command_queue command_queue)
cl_int clFinish(cl_command_queue command_queue)
```

Creating an OpenCL Program Object

- **What is an OpenCL C Code ?**

  - OpenCL C Code (Written to run on an OpenCL device) is called a *program*.
  - A program is a collection of functions called *kernels*, where kernels are units of execution that can be scheduled to run a device.
  - There is no need for an OpenCL application to have been *prebuilt* against the AMD, NVIDIA, or Intel runtime.
  - OpenCL software links to a command runtime layer (called the **ICD**); all platform-specific SDK activity is delegated to a vendor runtime through a dynamic library interface.

    - ICD: Installable Client Driver for OpenCL

Creating an OpenCL Program Object

What is an OpenCL^{TM} ICD ?

- The OpenCL ICD (Installable Client Driver) is a means of allowing multiple OpenCL implementations to co-exist and applications to select between them at runtime.
- User application is responsible for selecting which of the OpenCL platforms present on a system it wishes to use, instead of just requesting system default.
- Using

  `clGetPlatfromIDs()` & **`ClGetPlatfromInfo()`**

  functions to examine the list of available OpenCL implementations and selecting the one which best suites user requirements.

Creating an OpenCL Program Object

❖ About OpenCL™ ICD - Vendor Platform

- At this point, OpenCL Studio selects either the NVIDIA or AMD platform.
- There is **no support** for multiple platforms yet, but that will likely be another abstraction to manage multiple platforms and devices.
- The AMD driver lets you choose between the CPU and the GPU
- NVIDIA however only supports the “CUDA enabled NVIDIA GPU”

Creating an OpenCL Program Object

❖ About OpenCL^{TM} ICD - Vendor Platform

- At this point, OpenCL Studio selects either the NVIDIA or AMD platform.
- There is no support for multiple platforms yet, but that will likely be another abstraction to manage multiple platforms and devices
- The AMD driver lets you choose between the CPU and the GPU
- NVIDIA however only supports the “CUDA enabled NVIDIA GPU”

Source : NVIDIA, Khronos AMD, References

OpenCL : The Execution Environment

Creating an OpenCL Program Object

- **How to create OpenCL Kernel ?**
- **What is the process of creating a kernel ?**

1. The OpenCL C source code is stored in a character string. If the source code is stored in a file on a disk, it must be read into memory and stored as a character array.
2. The source code is turned into a program object **`cl_program`**, by calling **`clCreateProgramWithSource()`**.
3. The program object is then compiled, for one or more OpenCL devices, with **`clBuildProgram()`**, If there are compile errors, they will be reported here.

Creating an OpenCL Program Object

- **Is "Binary Representation " very vendor specific ?**

  - **AMD**: In the AMD runtime, there are two main classes of devices : x86 CPUs and GPUs

    - X86 CPUs **`clBuildProgram()`** generates x86 instructions that can be directly executed on the device.

    - For the GPUs, it will create AMD's GPU intermediate language (IL), a high-level intermediate language that represents a single **`work-item`** & compiled for a specific GPU's architecture later. (Generating ISA -code specific instruction set architecture)

  - The advantage of using such an IL is to allow the GPU ISA itself to change from one device or generation to another in what is still a very rapidly developing architectural space

Creating an OpenCL Program Object

- **Is "Binary Representation " very vendor specific ?**

  - **Additional Feature :** Build process is the ability to generate both the final binary format and various intermediate representations

  - Serialize these binaries (Write them to out to disk)

  - **OpenCL** provides a function to return information about program objects, **`clGetPrograminfo()`**

    - Flags to this function : **`cl_PROGRAM_BINARIES,`** which returns a vendor-specific set of binary objects generated by **`clBuildProgram()`**

  - OpenCL provides **`clCreateProgramWithBinary(),`** which takes a list of binaries that matches its device list.

Binary Representation on GPUs

- **Is "Binary Representation " very vendor specific ?**
  - **NVIDIA:** calling its intermediate representation PTX (PTX is an intermediate assembly language for NVIDIA GPUs) NVCC is the NVIDIA compiler driver
  - **PTX:** a low-level parallel thread execution virtual machine and instruction set architecture (ISA). PTX exposes the GPU as a data-parallel computing device.
  - PTX defines a virtual machine and ISA for general purpose parallel thread execution. . (ISA - code specific instruction set architecture)

Binary Representation on GPUs

❖ **Is “Binary Representation “ very vendor specific ?**

NVIDIA :

- PTX programs are translated at install time to the target hardware instruction set.
- PTX-to-GPU translator and driver enable NVIDIA GPUs to be used as programmable parallel computers.
- Provide a stable ISA that spans multiple GPU generations.
- Achieve performance in compiled applications comparable to native GPU performance.

OpenCL : The Execution Environment

The OpenCL Kernel

❖ **How to obtain "`cl_kernel`" object that can be used to execute kernels on a device ?**

- Extract kernel from the **`cl_program`**
- Similar to obtaining an exported function from a dynamic Lib.
  - The name of the kernel that the program exports is used to request it from the compiled program object.
  - The name of the kernel is passed to **`clCreateKernel()`**, along with the program object, and the kernel object will be returned if the program object was valid and the particular kernel is found.
- A few more steps are required before the kernel can actually be executed.

The OpenCL Kernel

- **What are the steps required before the kernel can actually be executed ?**

  - Executing a kernel requires dispatching it through an **enqueue** function

  - Specify each kernel argument individually using the function **clSetKernelArg()**

    - This function takes kernel object, an index specifying the argument number, the size of the argument, and a pointer to the argument..

The OpenCL Kernel

- **What are the steps required before the kernel can actually be executed ?**
  - When a kernel is executed, this information is used to transfer arguments to the device
  - After any required memory objects are transferred to the device and the kernel arguments are set, the kernel is ready to be executed.
  - Requesting that a device begin executing a kernel is done with a call to **clEngueueNDRangeKernel()**

OpenCL : The Execution Environment

```
cl_int
clEngueueNDRangeKernel(
   cl_command_queue command_queue
   cl_kernel kernel,
   cl_uint work_dim
   const size_t *global_work_offset,
   const size_t *global_work_size,
   const size_t *local_work_size,
   cl_unit num_events_in_wait_list,
   const cl_event *event_wait_list,
   cl_event *event)
```

OpenCL : The Execution Environment

The OpenCL Kernel : `clEngueueNDRangeKernel()`

- A command queue should be specified so that the target device is known

- The **`clEngueueNDRangeKernel()`** call is asynchronous

  - It will return immediately after the command is enqueued in the command queue and likely before the kernel has even started exeuction.

  - Either **`clWaitForEvent()`** or **`clFinfish()`** can be used to block execution on the host until the kernel completes.

The OpenCL Kernel : `clEngueueNDRangeKernel()`

- At this point, we have presented all the required host API commands needed to enable the reader to run a complete OpenCL Program

Part-III(K)

OpenCL Example Programs

Example Program -1

Kernels and the OpenCL Execution Model

Addition of two vectors : How to define workgroups & work-items

- work-items within a workgroup can perform “barrier synchronization”
- work-items within a workgroup can access to a shared memory address space. (Does not affect the scalability of a large concurrent dispatch)

Example Program : <u>Addition of two vectors of size 1024</u>

- The workgroup size might be specified as

```
size_t workGroupSize(3) = (64,1,1);
```

- Total number of work-items for array : 1024
- Total number of workgroups : 1024/64 = 64 workgroups

Kernels and the OpenCL Execution Model

- **Example Program** : Addition of two vectors (**Sequential**)
  - Algorithm executes a loop with as many iterations as there are elements to compute.
  - Each loop iterations adds the corresponding locations in the **input** arrays together and stores the result into the **output** array.

```
//Perform element-wise addition of A & B and
//Stores in C - There are N elements per array
void vecadd(int *C, int *A, int *B, int N)
{
  for(int =0; i < n; i++)
  {
    C[i] = A[i] + B[i];
  }
}
```

Source : Khronous, & References

Kernels and the OpenCL Execution Model

- **Example Program** : Addition of two vectors (**Multi-Core Device**)
  - Use low level coarse-grained threading API (POSIX threads) (One can use Data Parallel model such as OpenMP).
    - Divide the work (i.e., loop iterations) between the threads
    - Work per iteration is (loop counter) may be small or large. Use Strip mining to chunk the loop iterations into a large granularity.

Source : NVIDIA, Khronos AMD, References

Kernels and the OpenCL Execution Model

❖ **Example Program** : Addition of two vectors (**Multi-Core Device**)

```
//Perform element-wise addition of A & B and
//Stores in C - There are N elements per array
//and NP CPU Cores

void vecadd(int *C, int *A, int *B, int N,int NP,int tid)
{
  int Elept = N/NP;  // elements per thread
  for(int = tid*Elept; i < (tid+1)*Elept; i++)
  {
    C[i] = A[i] + B[i];
  }
}
```

Source : Khronous, & References

Kernels and the OpenCL Execution Model

Example Program : Addition of two vectors (**OpenCL**)

❖ When an OpenCL device begins executing a kernel, it provides intrinsic function that allow a ***work-item*** to identify itself

- Current work-item position is given by OpenCL Intrinsic function **get_global_id(0)**

```
// Perform element-wise addition of A & B & Stores in C
// N work items will be created to execute this kernel.
__kernel
void vecadd(_global int *C,
                 _global int *A,
                 _global int *B)
{
 int tid = get_global_id(0);
  C[tid] = A[tid] + B[tid];
}
```

Source : NVIDIA, Khronos AMD, References

Example Program -2

OpenCL : to write data-parallel programs

❖ Simple Matrix Multiplication Example:

- OpenCL host programs can be written in either C or using the OpenCL, C++ Wrapper API.

- Serial implementation : C or C++

  - The code iterates over three nested for loops, multiplying Matrix **A** by Matrix **B** and storing the result in Matrix **C**.
  - The **two** outer loops are used to iterative over each element of the output matrix
  - The **innermost** loop will iterate over the individual elements of the input elements of the input matrices to calculate the result of each output location.

OpenCL : to write data-parallel programs

❖ Simple Matrix Multiplication Example:

➢ OpenCL host programs can be written in either C or using the OpenCL, C++ Wrapper API.

➢ Serial implementation : C or C++

- The code iterates over three nested for loops, multiplying Matrix **A** by Matrix **B** and storing the result in Matrix **C**.
- The **two** outer loops are used to iterative over each element of the output matrix
- The **innermost** loop will iterate over the individual elements of the input elements of the input matrices to calculate the result of each output location.

OpenCL PLATFROM AND DEVICES

Serial Implementation

```
// Iteration over the rows of Matrix A
for ( int i = 0; i< heightA;  i++)
{
    // Iteration over the columns of MatrixB
    for ( int j = 0; j< widthB;  j++) {
            C[i][j] = 0;

            // Multiply and accumulate  over values in the current row
            // of A and column of B
            for ( int k = 0; k< widthA;  k++) {
            C[i][j] += A[i][k] * B[k][j];
            }
   }
}
```

OpenCL : to write data-parallel programs

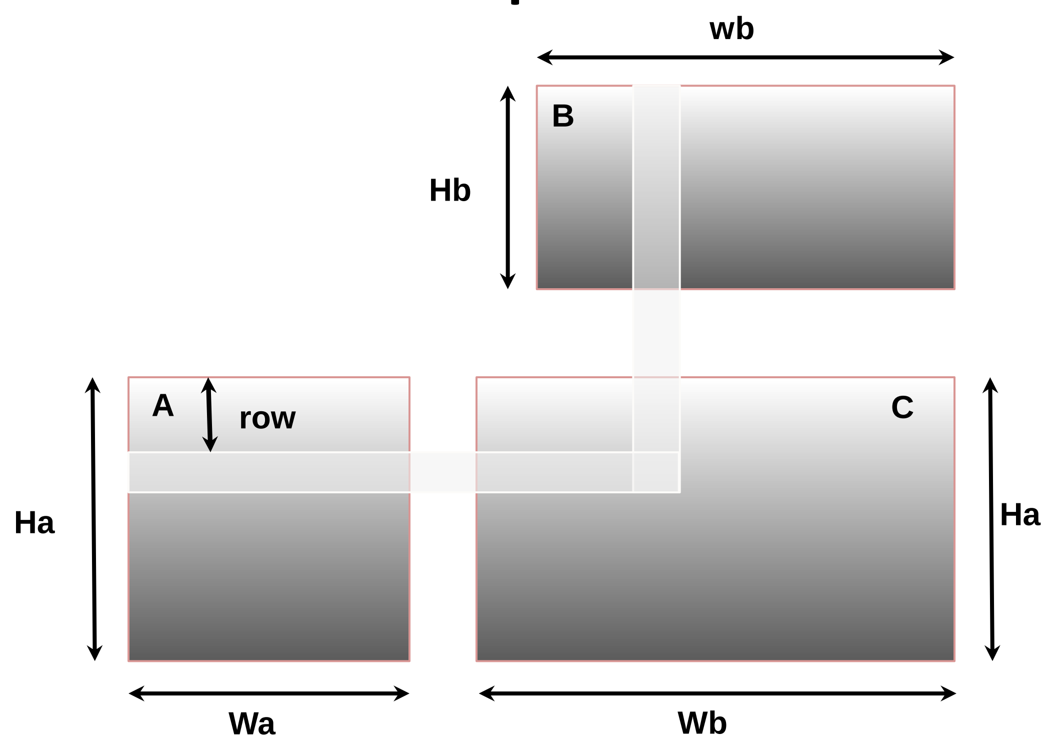
❖ OpenCL Simple Implementation : Matrix Multiplication

- **Two** outer loops work independently of each other
  - Separate **work-item** can be created for **each** output element of the matrix
  - The **two** outer for-loops are mapped to the **two** dimensional range of **work-item** for the kernel.

- Serial implementation : C or C++
  - The code iterates over three nested for loops, multiplying Matrix **A** by Matrix **B** and storing the result in Matrix **C**.
  - The **two** outer loops are used to iterative over each element of the output matrix
  - The **innermost** loop will iterate over the individual elements of the input elements of the input matrices to calculate the result of each output location.

OpenCL Simple Implementation : Matrix Multiplication

- Each **work-item** reads in its own row of **Matrix A** and its column of **Matrix B**.
- The data being read is multiplied and written at the appropriate location of the output **Matrix C**

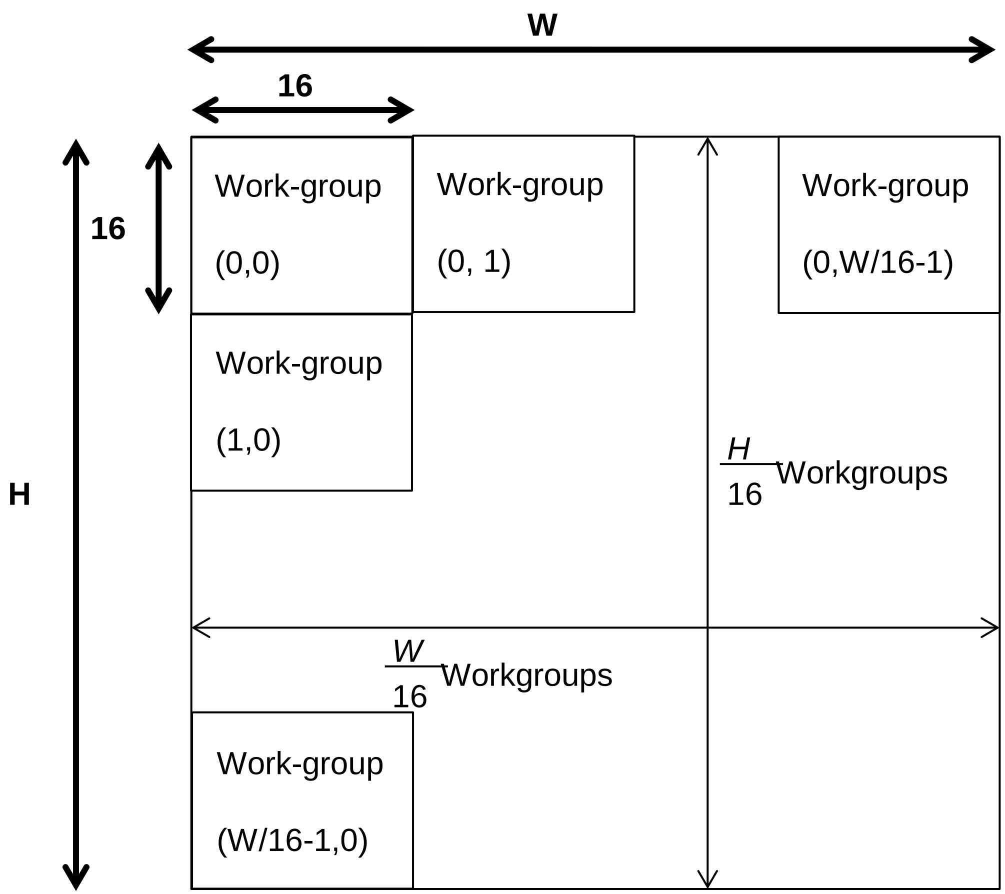


Each output value in a matrix multiplication is generated independently of all others.

Data Parallel Kernel Implementation

```
// Iteration over the rows of Matrix A
for ( int i = 0; i< heightA;  i++)
{
    // Iteration over the columns of MatrixB
    for ( int j = 0; j< widthB;  j++) {
            C[i][j] = 0;

            // Multiply and accumulate  over values in the current row
            // of A and column of B
            for ( int k = 0; k< widthA;  k++) {
            C[i][j] += A[i][k] * B[k][j];
            }
   }
}
```

OpenCL : Work-items & work-Groups



Input image workgroup configuration

Each element of the input image is handled by one work-item. Each work-item calculates its data’s coordinates and writes image out.

Simple Matrix Multiplication Example

Step 1: Set Up Environment

In this step, we declare a context, choose a device type, and create the context and a command queue. Throughout this example, the ci ErrNum variable should always be checked to see if an error code is returned by the implementation.

```
cl_int ciErrNum;

//Use the first platform
cl_platform_i d platform;
ci ErrNum = clGet PIatformIDs (1, &platform, NULL);

//Use the first device
cl_device_id device;
ciErrNum = clGetDevice IDs(
   piatform,
   CL_DEVICE_TYPE_ALL,
   1,
   &device,
   NULL);
```

Source : NVIDIA, Khronos AMD, References

Simple Matrix Multiplication Example

```
context_properties cps[3]={;
   CL_CONTEXT_PLATFORM, (cl_context_properties)platform, 0};

//Create the context
cl_context ctx = clCreateContext(
   cps,
   1,
   &device,
   NULL,
   NULL,
   &ciErrNum);

//Create the command queue
cl_command_queue myqueue = clCreateCommandQueue{
   ctx,
   device,
   0,
   &ciErrNum0;
```

Source : NVIDIA, Khronos AMD, References

Simple Matrix Multiplication Example

Step 2: Declare Buffers and Move Data

Declare buffers on the device and enqueue copies of input matrices to the device. Also declare the output buffer.

```
// We assume that A, B, C are float arrays which
// have been declared and initialized

// Allocate space for Matrix A on the device
 cl_mem buf ferA = cl CreateBuf fer(
  ctx,
  CL_MEM_READ_ONLY,
  wA*hA*si zeof(float),
  NULL,
  &ci ErrNum);

// Copy Matrix A to the device
  ci ErrNum = clEnqueueWriteBuffe-(
  myqueue,
  bufferA,
  CL_TRUE, 0,
 wA*hA*sizeof(float), (void *)A, 0.
 NULL, NULL);
```

Simple Matrix Multiplication Example

```
// Copy Matrix A to the device
   ci ErrNum = clEnqueueWriteBuffe-(
   myqueue,
   bufferA,
   CL_TRUE,
   0,
   wA*hA*sizeof(float),
  (void *)A,
   0.
   NULL,
   NULL);

// Allocate space for Matrix B on the device
   cl_mem bufferB = clCreateBuffer(
   ctx,
   CL_MEM_READ_ONLY,
   wB*hB*si zeof(float),
   NULL,
   &ci ErrNum);
```

Simple Matrix Multiplication Example

```
// Copy Matrix B to the device
cl ErrNum = clEnqueueWri teBuffer(
   myqueue,
   bufferB,
   CL_TRUE,
   0,
   wB*hB*si zeof(float),
   (void *)B,
   0,
   NULL,
   NULL);

// A1 locate space for Matrix C on the device
 cl_mem bufferC = cl CreateBuffer(
   ctx,
   CL_MEM_READ_ONLY,
   hA*wB*sizeof(float),
   NULL.
   &ci ErrNum);
```

Simple Matrix Multiplication Example

Step 3: Runtime Kernel Compilation

Compile the program from the kernel array, build the program, and define the kernel.

```
 // We assume that the program source is stored in the variable
 // 'source' and is NULL terminated
cl_programiriyprog = clCreateProgramWithSource (
   ctx,
   1,
   (const char**)&source,
   NULL,
   &ci ErrNum);

// Compi 1 p the program. Passing NULL for the 'devi ce_]1st'
// argume.it targets all devices in the context ciErrNum=clBuildProgram(myprog, 0,
NULL, NULL, NULL, NULL);

// Create, the kernel
cl_kernel mykernel = clCreateKernel(
   myprog,
   "simpleMulti ply",
   &ci ErrNum);
```

Simple Matrix Multiplication Example

Step 4: Run the Program

Set kernel arguments and the workgroup size. We can then enqueue kernel onto the command queue to execute on the device.

```
//Set the kernel arguments
clSetKernelArg(my kernel, 0, si zeof(cl_mem), (void *)&d_C); clSetKernelArg(mykernel, 1,
sizeof(cl_int), (void *)&wA);

cl Set Kernel Arg( my kernel , 2 , sizeof(cl_int), (void *)&hA); clSetKerne1Arg(my kernel, 3,
sizeof(cl_int), (void * )&wB); clSetKernelArg(my kernel, 4. si zeof(cl_i nt), (void*)&hB);

cl SetKernel Arg( mykernel , 5, sizeof (cl__mem), (void *)&d_A);

cl Set Kernel Arg( my kerne 1 , 6, sizeof( cl__mem) , (void *)&d_B );

// Set local and global workgroup sizes
//We assume the matrix dimensions are divisible by 16

size_t1ocalws[2] = 116 ,161 ;
size„t globalws[2] = iwC, hC};
```

Simple Matrix Multiplication Example

```
// Execute the kernel
ciErrNum = clEnqueueNDRangeKernel(
   myqueue,
   mykernel ,
   2,
   NULL,
   globalws,
   1ocalws,
   0,
   NULL,
   NULL);
```

Simple Matrix Multiplication Example

Step 5: Obtain Results to Host

After the program has run, we enqueue a read back of the result matrix from the device buffer to host memory.

```
// Read the output data back to the host
ciErrNum = cl EnqueueReadBuffer(
   myqueue,
   d_C,
   CL_TRUE,
   0,
   wC*hC*si zeof(float),
   (void *)C, 0,
   NULL,
   NULL);
```

The steps outlined here show an OpenCL implementation of matrix multiplication that can be used as a baseline. In later chapters, we use our understanding of data-parallel architectures to improve the performance of particular data-parallel algorithms.

OpenCL Summary

- History of OpenCL
- Easing cross-platform development with major enhancements for stream software strategy
- GPU Programming – OpenGL, DirectX, NVIDIA (CUDA), AMD (Brook+)
- Aggressively expanding stream strategy to consumer segment

OpenCL Summary

- ❖ A new computer language that works across GPUs and CPUs
  - ➢ C /C++ with extensions
  - ➢ Familiar to developers
  - ➢ Includes a rich set of built-in functions
  - ➢ Makes it easy to develop data- and task- parallel compute programs
- ❖ Defines hardware and numerical precision requirements
- ❖ Open standard for heterogeneous parallel computing

References

1. Randi J. Rost, OpenGL – shading Language, Second Edition, Addison Wesley 2006
2. GPGPU Reference http://www.gpgpu.org
3. NVIDIA http://www.nvidia.com
4. NVIDIA tesla http://www.nvidia.com/object/tesla_computing_solutions.html
5. RAPIDMIND http://www.rapidmind.net
6. Peak Stream - Parallel Processing (Acquired by Google in 2007) http:/www.google.com
7. guru3d.com http://www.guru3d.com/news/sandra-2009-gets-gpgpu-support/
   ATI & AMD http://ati.amd.com/products/radeon9600/radeon9600pro/index.html
8. AMD http:www.amd.com
9. AMD Stream Processors http://ati.amd.com/products/streamprocessor/specs.html
10. RAPIDMIND & AMD http://www.rapidmind.net/News-Aug4-08-SIGGRAPH.php
11. General-purpose computing on graphics processing units (GPGPU) http://en.wikipedia.org/wiki/GPGPU
12. Khronous Group, OpenGL 3, December 2008 URL : http://www.khronos.org/opencl
13. *OpenCL - The open standard for parallel programming of heterogeneous systems* URL : http://www.khronos.org/opencl
14. Programming the GPU and a brief intro to the OPENGL shading language – Marcel Cohan & VVR Talk
15. David B Kirk, Wen-mei W. Hwu nvidia corporation, 2010, Elsevier, Morgan Kaufmann Publishers, 2011
16. Benedict R Gaster, Lee Howes, David R Kaeli, Perhadd Mistry Dana Schaa, Heterogeneous Computing with OpenCL, Elsevier, Moran Kaufmann Publishers, 2011
17. The OpenCL 1.2 Specification (Document Revision 15) Last Released November 15, 2011 Editor : Aaftab Munshi Khronos OpenCL Working Group
18. The OpenCL 1.1 Quick Reference card

References

19. http://developer.amd.com/sdks/AMDAPPSDK/downloads/Pages/default.aspx AMD APP SDK with OpenCL 1.2 Support
20. http://developer.amd.com/sdks/AMDAPPSDK/downloads/Pages/default.aspx#oneAMD-APP-SDKv2.7 (Linux) with OpenCL 1.2 Support
21. http://icl.cs.utk.edu/magma/software/ MAGMA OpenCL
22. http://developer.amd.com/zones/OpenCLZone/pages/GettingStarted.aspx Getting Started with OpenCL
23. http://developer.amd.com/openclforum AMD Developer OpenCL FORUM
24. http://developer.amd.com/zones/OpenCLZone/programming/pages/benchmarkingopenclperformance.aspx AMD Developer Central - Programming in OpenCL - Benchmarks performance
25. http://developer.amd.com/sdks/AMDAPPSDK/assets/opencl-1.2.pdf OpenCL 1.2 (pdf file)
26. http://developer.amd.com/zones/opensource/pages/ocl-emu.aspx AMD OpenCL Emulator-Debugger
27. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf The OpenCL 1.2 Specification (Document Revision 15) Last Released November 15, 201 Editor : Aaftab Munshi <I> Khronos OpenCL Working Group
28. http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/ OpenCL1.1 Reference Pages

Source : NVIDIA, Khronos AMD, References

Thank you

Any Questions ?

Source : NVIDIA, Khronos AMD, References

References Acknowledgement s

References :

1. OpenACC: www.openacc-standard.org/
2. GPU Computing with OpenACC Directives Presented by John Urbanic, Pittsburgh Supercomputing Center Authored by Mark Harris NVIDIA Corporation
3. Cray OpenACC http://www.openacc-standard.org/content/cray-even
4. CAPS – OpenACC : http://www.caps-entreprise.com/index.php
5. http://www.caps-entreprise.com/fr/page/index.php?id=148&p_p=36 CAPS OpenACC COMPILER
6. PGI OpenACC : www.pgroup.com/resources/accel.htm
7. http://www.opengpu.net/EN/attachments/154_HiPEAC2012_OpenGPU_nVidia.pdf OPENACC DIRECTIVES FOR ACCELERATORS –NVIDIA
8. http://www.pgroup.com/doc/openACC_gs.pdf PGI OpenACC Compilers Getting Started Guide Version 12.3
9. *Introduction to OpenACC;* Oscar Hernandez, Richard Graham, Computer Science and Mathematics (CSM), Application Performance Tools Group,Oak Ridge National Laboratories, U.S Dept. of Energy
10. *GPU Programming with CUDA and OpenACC*; Axel Koehler – NVIDIA
11. http://www.nvidia.com/docs/IO/116711/OpenACC-API.pdf The OpenACC™ API QUICK RE FEREN CE GUIDE
12. http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf The OpenACC™ Application Programming Interface Version 1.0 November, 2011

Source : NVIDIA & References given in the presentation

References

1. Randi J. Rost, OpenGL – shading Language, Second Edition, Addison Wesley 2006
2. GPGPU Reference http://www.gpgpu.org
3. NVIDIA http://www.nvidia.com
4. NVIDIA tesla http://www.nvidia.com/object/tesla_computing_solutions.html
5. NVIDIA CUDA Reference http://www.nvidia.com/object/cuda_home.html
6. CUDA sample source code: http://www.nvidia.com/object/cuda_get_samples.html
7. List of NVIDIA GPUs compatible with CUDA: The href://www.nvidia.com/object/cuda_learn_products.html
8. Download the CUDA SDK: www.nvidia.com/object/cuda_get.html
9. Specifications of nVIDIA GeForce 8800 GPUs:
10. RAPIDMIND http://www.rapidmind.net
11. Peak Stream - Parallel Processing (Acquired by Google in 2007) http:/www.google.com
12. guru3d.com http://www.guru3d.com/news/sandra-2009-gets-gpgpu-support/
    ATI & AMD http://ati.amd.com/products/radeon9600/radeon9600pro/index.html
13. AMD http:www.amd.com
14. AMD Stream Processors http://ati.amd.com/products/streamprocessor/specs.html
15. RAPIDMIND & AMD http://www.rapidmind.net/News-Aug4-08-SIGGRAPH.php
16. Merrimac - Stream Architecture Standford Brook for GPUs
    http://www-graphics.stanford.edu/projects/brookgpu/
17. Standford : Merrimac - Stream Architecture http://merrimac.stanford.edu/
18. ATI RADEON - AMD http://www.canadacomputers.com/amd/radeon/
19. ATI & AMD - Technology Products http://ati.amd.com/products/index.html
20. Sparse Matrix Solvers on the GPU ; conjugate Gradients and Multigrid by *Jeff Bolts, Ian Farmer, Eitan Grinspum, Peter Schroder* , Caltech Report (2003); Supported in part by NSF, nVIDIA, etc....
21. Scan Primitives for GPU Computing by *Shubhabrata Sengupta, Mark Harris*, Yao Zhang and John D Owens* University of California Davis & *nVIDIA Corporation *Graphic Hardware (2007).*
22. Horm D; *Stream reduction operations for GPGPU applciations in GPU Genes 2 Phar M.,* (Ed.) Addison Weseley, March 2005; Chapter 36, pp. 573-589 Graphic Hardware (2007).
23. *Bollz J., Farmer I., Grinspun F., Schroder F :* Sparse Matris Solvers on the GPU ; Conjugate Gradients and multigrid ACM Transactions on Graphics *(Proceedings of ACM SIGRAPH 2003) 22, 2 (Jul y2003) pp 917-924 Graphic Hardware (2007).*
24. NVIDIA CUDA Compute Unified Device Architecture - Programming Guide - Version 1.1 November 2007

References

25. Tom R. Halfhill, *Number crunching with GPUs PeakStream Math API Exploits Parallelism in Graphics Processors, Ocotober 2006; Microprocessor http://www.mdronline.com*
26. Tom R. Halfhill, *Parallel Processing with CUDA Nvidia's High-Performance Computing Platform Uses Massive Multithreading ;* Microprocessors, Volume 22, Archive 1, January 2008 *http://www.mdronline.com*
27. J. Tolke, M.Krafczyk *Towards Three-dimensional teraflop CFD Computing on a desktop PC using graphics hardware Institute for Computational Modeling in Civil Engineering,* TU Braunschweig (2008)
28. I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Hoston, P.Hanrahan, Brook for GPUs ; *Stream Computing on GRaphics Hadrware, ACM Tran. GRaph (SIGGRAPH) 2008*
29. Z. Fan, F. Qin, A.E. Kaufamm, S. Yoakum-Stover*, GPU cluster for Hgh Performance Computing in : Proceedings of ACM/IEEE Superocmputing Conference 2004 pp. 47-59.*
30. J. Kriiger, R. Wetermann, *Linear Algeria operators for GPU implementation of Numerical Algorithms ACm Tran, Graph (SIGGRAPH) 22 (3) pp. 908-916. (2003)*
31. Tutorial SC 2007 SC05 : *High Performance Computing with CUDA*
32. *FASTRA http://www.fastra.ua.ac.bc/en/faq.html*
33. *AMD Stream Computing software Stack ; http://www.amd.com*
34. *BrookGPU : http://graphics standafrod.edu/projects/brookgpu/index.html*
35. *FFT – Fast Fourier Transform www.fftw.org*
36. BLAS – *Basic Linear Algebra Suborutines – www.netlibr.org/blas*
37. LAPACK : *Linear Algebra Package – www.netlib.org/lapack*
38. Dr. Larry Seller, Senipr Principal Engineer; Larrabee : A Many-core Intel Architecture for Visual computing, Intel Deverloper FORUM 2008
39. *Tom R Halfhill,* Intel's Larrabee Redefines GPUs – Fully Programmable Many core Processor Reaches Beyond Graphics, Microprocessor Report September 29, 2008
40. Tom R Halfhill AMD's Stream Becomes a River – Parallel Processing Platform for ATI GPUs Reaches More Systems, Microprocessor Report December 2008
41. AMD's ATI Stream Platform http://www.amd.com/stream
42. General-purpose computing on graphics processing units (GPGPU) http://en.wikipedia.org/wiki/GPGPU
43. Khronous Group, OpenGL 3, December 2008 URL : http://www.khronos.org/opencl

References

44. NVIDA CUDA C Programming Guide Version V4.0, May 2012 (5/6/2012) http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf
45. NVIDIA Developer Zone http://developer.nvidia.com/category/zone/cuda-zone
46. NVIDIA CUDA Toolkit 4.0 (May 2012) http://developer.nvidia.com/cuda-toolkit-4.0
47. NVIDIA CUDA Toolkit 4.0 Downloads http://developer.nvidia.com/cuda-toolkit
48. NVIDIA Developer ZONE – GPUDirect http://developer.nvidia.com/gpudirect
49. NVIDIA OpenCL Programming Guide for the CUDA Architecture version 4.0 Feb, 2012 (2/14,2012) http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Programming_Guide.pdf
50. Optimization : NVIDIA OpenCL Best Practices Guide Version 1.0 Feb 2012 http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Best_Practices_Guide.pdf
51. NVIDIA OpenCL JumpStart Guide - Technical Brief http://developer.download.nvidia.com/OpenCL/NVIDIA_OpenCL_JumpStart_Guide.pdf
52. NVIDA CUDA C BEST PRACTICES GUIDE (Design Guide) V4.0, May 2012
53. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf
54. NVIDA CUDA C Programming Guide Version V5.0, May 2012 (5/6/2012)
55. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf
56. Programming Massively Parallel Processors - A Hands-on Approach, David B Kirk, Wen-mei W. Hwu, Nvidia corporation, 2010, Elsevier, Morgan Kaufmann Publishers, 2011
57. Aftab Munshi Benedict R Gaster, timothy F Mattson, James Fung, Dan Cinsburg, Addison Wesley, OpenCL Progrmamin Guide, Pearson Education, 2012
58. The OpenCL 1.2 Specification Khronos OpenCL Working Group
59. http://www.khronos.org/files/opencl-1-2-quick-reference-card.pdf‟ The OpenCL 1.2 Quick-reference-card ; Khronos OpenCL Working Group

References

60. Mary Fetcher and Vivek Sarkar, Introduction to GPGPUS – Seminar on Heterogeneous Processors, Dept. of computer Science, Rice University, October 2007
61. OpenCL - The open standard for parallel programming of heterogeneous systems URL : http://www.khronos.org/opencl
62. Tom R. Halfhill, Parallel Processing with CUDA Nvidia's High-Performance Computing Platform Uses Massive Multithreading ; Microprocessors, Volume 22, Archive 1, January 2008 http://www.mdronline.com
63. Matt Pharr (Author), Randima Fernando, GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation ,Addison Wesley , August 2007
64. NVIDIA GPU Programming Guide http://www.nvidia.com
65. Perry H. Wang1, Jamison D. Collins1, Gautham N. Chinya1, Hong Jiang2, Xinmin Tian3 , EXOCHI: Architecture and Programming Environment for A Heterogeneous Multi-core Multithreaded System, PLDI'07
66. Karl E. Hillesland, Anselmo Lastra GPU Floating-Point Paranoia, University of North Carolina at Chapel Hill
67. KARPINSKI, R. 1985. Paranoia: A floating-point benchmark. Byte Magazine 10, 2 (Feb.), 223–235.
68. GPGPU Web site : http://www.ggpu.org
69. Graphics Processing Unit Architecture (GPU Arch) With a focus on NVIDIA GeForce - 6800 GPU, Ajit Datar, Apurva Padhye Computer Architecture
70. Nvidia 6800 chapter from GPU Gems 2 http://download.nvidia.com/developer/GPU_Gems_2/GPU_Gems2_ch30.pdf
71. OpenGL design http://graphics.stanford.edu/courses/cs448a-01-fall/design_opengl.pdf
72. OpenGL programming guide (ISBN: 0201604582)
73. Real time graphics architectures lecture notes http://graphics.stanford.edu/courses/cs448a-01-fall/
74. GeForce 256 overview http://www.nvnews.net/reviews/geforce_256/gpu_overviews.html
75. GPU Programming "Languages http://www.cis.upenn.edu/~suvenkat/700/
76. Programming the GPU and a brief intro to the OPENGL shading language – Marcel Cohan & VVR Talk
77. Johan Seland, GPU Programming and Computing, Workshop on High-Performance and Parallel Computing Simula Research Laboratory October 24, 2007
78. Daniel Weiskopf, Basics of GPU-Based Programming, Institute of Visualization and Interactive Systems, Interactive Visualization of Volumetric Data on Consumer PC Hardware: Basics of Hardware-Based Programming University of Stuttgart, **VIS 2003**

Source & Acknowledgements : NVIDIA, References

References

79. http://www.nvidia.com/object/nvidia-kepler.html NVIDIA Kepler Architecture 2012
80. http://developer.nvidia.com/cuda-toolkit NVIDIA CUDA toolkit 5.0 Preview Release April 2012
81. http://developer.nvidia.com/category/zone/cuda-zone NVIDIA Developer Zone
82. http://developer.nvidia.com/gpudirect RDMA for NVIDIA GPUDirect coming in CUDA 5.0 Preview Release, April 2012
83. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf NVIDIA CUDA C Programming Guide Version 4.2 dated 4/16/2012 (April 2012)
84. http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf Dynamic Parallelism in CUDA Tesla K20 Kepler GPUs - Prelease of NVIDIA CUDA 5.0
85. http://developer.nvidia.com/cuda-downloads NVIDIA Developer ZONE - CUDA Downloads CUDA TOOLKIT 4.2
86. http://developer.nvidia.com/gpudirect NVIDIA Developer ZONE – GPUDirect
87. http://developer.nvidia.com/openacct OpenACC - NVIDIA
88. http://developer.nvidia.com/cuda-toolkit Nsight, Eclipse Edition Pre-release of CUDA 5.0, April 2012
89. The OpenCL Specification, Version 1.1,</B> Published by Khronos OpenCL Working Group, Aaftab Munshi (ed.), 2010.
90. NVIDA CUDA C Programming Guide Version V4.0, May 2012 (5/6/2012) http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf
91. http://www.khronos.org/files/opencl-1-1-quick-reference-card.pdf The OpenCL 1.1 Quick Reference card.
92. NVIDIA Developer Zone http://developer.nvidia.com/category/zone/cuda-zone
93. NVIDIA CUDA Toolkit 4.0 (May 2012) http://developer.nvidia.com/cuda-toolkit-4.0

References

94. NVIDIA CUDA Toolkit 4.0 Downloads http://developer.nvidia.com/cuda-toolkit
95. NVIDIA Developer ZONE – GPUDirect http://developer.nvidia.com/gpudirect
96. NVIDIA OpenCL Programming Guide for the CUDA Architecture version 4.0 Feb, 2012 (2/14,2012) http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Programming_Guide.pdf

97. Optimization : NVIDIA OpenCL Best Practices Guide Version 1.0 Feb 2012 http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Best_Practices_Guide.pdf
98. NVIDIA OpenCL JumpStart Guide - Technical Brief http://developer.download.nvidia.com/OpenCL/NVIDIA_OpenCL_JumpStart_Guide.pdf
99. NVIDA CUDA C BEST PRACTICES GUIDE (Design Guide) V4.0, May 2012
100. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf
101. NVIDA CUDA C Programming Guide Version V5.0, May 2012 (5/6/2012)
102. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

Source & Acknowledgements : NVIDIA, References

References

1. Randi J. Rost, OpenGL – shading Language, Second Edition, Addison Wesley 2006
2. GPGPU Reference http://www.gpgpu.org
3. NVIDIA http://www.nvidia.com
4. NVIDIA tesla http://www.nvidia.com/object/tesla_computing_solutions.html
5. NVIDIA CUDA Reference http://www.nvidia.com/object/cuda_home.html
6. CUDA sample source code: http://www.nvidia.com/object/cuda_get_samples.html
7. List of NVIDIA GPUs compatible with CUDA: The href://www.nvidia.com/object/cuda_learn_products.html
8. Download the CUDA SDK: www.nvidia.com/object/cuda_get.html
9. Specifications of nVIDIA GeForce 8800 GPUs:
10. RAPIDMIND http://www.rapidmind.net
11. Peak Stream - Parallel Processing (Acquired by Google in 2007) http:/www.google.com
12. guru3d.com http://www.guru3d.com/news/sandra-2009-gets-gpgpu-support/
    ATI & AMD http://ati.amd.com/products/radeon9600/radeon9600pro/index.html
13. AMD http:www.amd.com
14. AMD Stream Processors http://ati.amd.com/products/streamprocessor/specs.html
15. RAPIDMIND & AMD http://www.rapidmind.net/News-Aug4-08-SIGGRAPH.php
16. Merrimac - Stream Architecture Standford Brook for GPUs
    http://www-graphics.stanford.edu/projects/brookgpu/
17. Standford : Merrimac - Stream Architecture http://merrimac.stanford.edu/
18. ATI RADEON - AMD http://www.canadacomputers.com/amd/radeon/
19. ATI & AMD - Technology Products http://ati.amd.com/products/index.html
20. Sparse Matrix Solvers on the GPU ; conjugate Gradients and Multigrid by *Jeff Bolts, Ian Farmer, Eitan Grinspum, Peter Schroder* , Caltech Report (2003); Supported in part by NSF, nVIDIA, etc....
21. Scan Primitives for GPU Computing by *Shubhabrata Sengupta, Mark Harris*, Yao Zhang and John D Owens* University of California Davis & *nVIDIA Corporation *Graphic Hardware (2007).*
22. Horm D; *Stream reduction operations for GPGPU applciations in GPU Genes 2 Phar M.,* (Ed.) Addison Weseley, March 2005; Chapter 36, pp. 573-589 Graphic Hardware (2007).
23. *Bollz J., Farmer I., Grinspun F., Schroder F :* Sparse Matris Solvers on the GPU ; Conjugate Gradients and multigrid ACM Transactions on Graphics *(Proceedings of ACM SIGRAPH 2003) 22, 2 (Jul y2003) pp 917-924 Graphic Hardware (2007).*
24. NVIDIA CUDA Compute Unified Device Architecture - Programming Guide - Version 1.1 November 2007

References

25. Tom R. Halfhill, *Number crunching with GPUs PeakStream Math API Exploits Parallelism in Graphics Processors, Ocotober 2006; Microprocessor http://www.mdronline.com*
26. Tom R. Halfhill, *Parallel Processing with CUDA Nvidia's High-Performance Computing Platform Uses Massive Multithreading ;* Microprocessors, Volume 22, Archive 1, January 2008 *http://www.mdronline.com*
27. J. Tolke, M.Krafczyk *Towards Three-dimensional teraflop CFD Computing on a desktop PC using graphics hardware Institute for Computational Modeling in Civil Engineering,* TU Braunschweig (2008)
28. I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Hoston, P.Hanrahan, Brook for GPUs ; *Stream Computing on GRaphics Hadrware, ACM Tran. GRaph (SIGGRAPH) 2008*
29. Z. Fan, F. Qin, A.E. Kaufamm, S. Yoakum-Stover*, GPU cluster for Hgh Performance Computing in : Proceedings of ACM/IEEE Superocmputing Conference 2004 pp. 47-59.*
30. J. Kriiger, R. Wetermann, *Linear Algeria operators for GPU implementation of Numerical Algorithms ACm Tran, Graph (SIGGRAPH) 22 (3) pp. 908-916. (2003)*
31. Tutorial SC 2007 SC05 : *High Performance Computing with CUDA*
32. *FASTRA http://www.fastra.ua.ac.bc/en/faq.html*
33. *AMD Stream Computing software Stack ; http://www.amd.com*
34. *BrookGPU : http://graphics standafrod.edu/projects/brookgpu/index.html*
35. *FFT – Fast Fourier Transform www.fftw.org*
36. BLAS – *Basic Linear Algebra Suborutines – www.netlibr.org/blas*
37. LAPACK : *Linear Algebra Package – www.netlib.org/lapack*
38. Dr. Larry Seller, Senipr Principal Engineer; Larrabee : A Many-core Intel Architecture for Visual computing, Intel Deverloper FORUM 2008
39. *Tom R Halfhill,* Intel's Larrabee Redefines GPUs – Fully Programmable Many core Processor Reaches Beyond Graphics, Microprocessor Report September 29, 2008
40. Tom R Halfhill AMD's Stream Becomes a River – Parallel Processing Platform for ATI GPUs Reaches More Systems, Microprocessor Report December 2008
41. AMD's ATI Stream Platform http://www.amd.com/stream
42. General-purpose computing on graphics processing units (GPGPU) http://en.wikipedia.org/wiki/GPGPU
43. Khronous Group, OpenGL 3, December 2008 URL : http://www.khronos.org/opencl

References

44. *Mary Fetcher and Vivek Sarkar,* Introduction to GPGPUS – Seminar on Heterogeneous Processors, Dept. of computer Science, Rice University, October 2007
45. *OpenCL - The open standard for parallel programming of heterogeneous systems* URL : http://www.khronos.org/opencl
46. Tom R. Halfhill, Parallel Processing with CUDA Nvidia's High-Performance Computing Platform Uses Massive Multithreading ; Microprocessors, Volume 22, Archive 1, January 2008 http://www.mdronline.com
47. *Matt Pharr (Author), Randima Fernando,* GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation *,Addison Wesley , August 2007*
48. *NVIDIA GPU Programming Guide* http://www.nvidia.com
49. *Perry H. Wang1, Jamison D. Collins1, Gautham N. Chinya1, Hong Jiang2, Xinmin Tian3 ,* EXOCHI: Architecture and Programming Environment for A Heterogeneous Multi-core Multithreaded System, PLDI'07
50. Karl E. Hillesland, Anselmo Lastra GPU Floating-Point Paranoia, University of North Carolina at Chapel Hill
51. KARPINSKI, R. 1985. Paranoia: A floating-point benchmark. Byte Magazine 10, 2 (Feb.), 223–235.
52. GPGPU Web site : http://www.ggpu.org
53. Graphics Processing Unit Architecture (GPU Arch) With a focus on NVIDIA GeForce - 6800 GPU, Ajit Datar, Apurva Padhye Computer Architecture
54. Nvidia 6800 chapter from GPU Gems 2 http://download.nvidia.com/developer/GPU_Gems_2/GPU_Gems2_ch30.pdf
55. OpenGL design http://graphics.stanford.edu/courses/cs448a-01-fall/design_opengl.pdf
56. OpenGL programming guide (ISBN: 0201604582)
57. Real time graphics architectures lecture notes http://graphics.stanford.edu/courses/cs448a-01-fall/
58. GeForce 256 overview http://www.nvnews.net/reviews/geforce_256/gpu_overviews.html
59. GPU Programming "Languages http://www.cis.upenn.edu/~suvenkat/700/
60. Programming the GPU and a brief intro to the OPENGL shading language – Marcel Cohan & VVR Talk
61. *Johan Seland,* GPU Programming and Computing, Workshop on High-Performance and Parallel Computing Simula Research Laboratory October 24, 2007
62. *Daniel Weiskopf,* Basics of GPU-Based Programming, Institute of Visualization and Interactive Systems, Interactive Visualization of Volumetric Data on Consumer PC Hardware: Basics of Hardware-Based Programming University of Stuttgart, **VIS 2003**

References

1. AMD Accelerated Parallel Processing (APP) SDK (formerly ATI Stream) with OpenCL 1.1 Support http://developer.amd.com/sdks/AMDAPPSDK/Pages/default.aspx
2. AMD Accelerated Parallel Processing (APP) SDK (formerly ATI Stream) with AMD APP Math Libraries (APPML); AMD Core Math Library (ACML); AMD Core Math Library for Graphic Processors (ACML-GPU) http://developer.amd.com/sdks/AMDAPPSDK/Pages/default.aspx
3. AMD Accelerated Parallel Processing (AMD APP) Programming Guide OpenCL : August 2012 http://developer.amd.com/sdks/AMDAPPSDK/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf
4. AMD Developer Central - OpenCL Zone, http://developer.amd.com/zones/OpenCLZone/Pages/default.aspx
5. AMD Developer Central - Programming in OpenCL http://developer.amd.com/zones/OpenCLZone/programming/Pages/default.aspx
6. AMD Developer Central - Programming in OpenCL - Benchmarks performance http://developer.amd.com/zones/OpenCLZone/programming/pages/benchmarkingopenclperformance.aspx
7. *The open standard for parallel programming of heterogeneous systems* URL : http://www.khronos.org/opencl
8. OpenGL design http://graphics.stanford.edu/courses/cs448a-01-fall/design_opengl.pdf
9. OpenGL programming guide (ISBN: 0201604582)
10. Real time graphics architectures lecture notes http://graphics.stanford.edu/courses/cs448a-01-fall/
11. GeForce 256 overview http://www.nvnews.net/reviews/geforce_256/gpu_overviews.html
12. GPU Programming “Languages http://www.cis.upenn.edu/~suvenkat/700/
13. Programming the GPU and a brief intro to the OPENGL shading language – Marcel Cohan & VVR Talk
14. *Johan Seland,* GPU Programming and Computing, Workshop on High-Performance and Parallel Computing Simula Research Laboratory October 24, 2007

References

1. NVIDA CUDA C Programming Guide Version V4.0, May 2012 (5/6/2012) http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf
2. NVIDIA Developer Zone http://developer.nvidia.com/category/zone/cuda-zone
3. NVIDIA CUDA Toolkit 4.0 (May 2012) http://developer.nvidia.com/cuda-toolkit-4.0
4. NVIDIA CUDA Toolkit 4.0 Downloads http://developer.nvidia.com/cuda-toolkit
5. NVIDIA Developer ZONE – GPUDirect http://developer.nvidia.com/gpudirect
6. NVIDIA OpenCL Programming Guide for the CUDA Architecture version 4.0 Feb, 2012 (2/14,2012) http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Programming_Guide.pdf

7. Optimization : NVIDIA OpenCL Best Practices Guide Version 1.0 Feb 2012 http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Best_Practices_Guide.pdf
8. NVIDIA OpenCL JumpStart Guide - Technical Brief http://developer.download.nvidia.com/OpenCL/NVIDIA_OpenCL_JumpStart_Guide.pdf
9. NVIDA CUDA C BEST PRACTICES GUIDE (Design Guide) V4.0, May 2012
10. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf
11. NVIDA CUDA C Programming Guide Version V4.0, May 2012 (5/6/2012)

12. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf