C-DAC Four Days Technology Workshop

ON

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

hyPACK-2013
(Mode-4 : GPUs)

Lecture Topic:
An Overview of CUDA enabled GPUs

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

An Overview of CUDA enabled NVIDIA GPUs

Lecture Outline

Following topics will be discussed

- An overview of CUDA enabled NVIDIA GPU
- Tuning & Performance Issues on NVIDIA GPUs
- An Overview of CUDA 4.x/5.0 & -Fermi /Kepler GK110

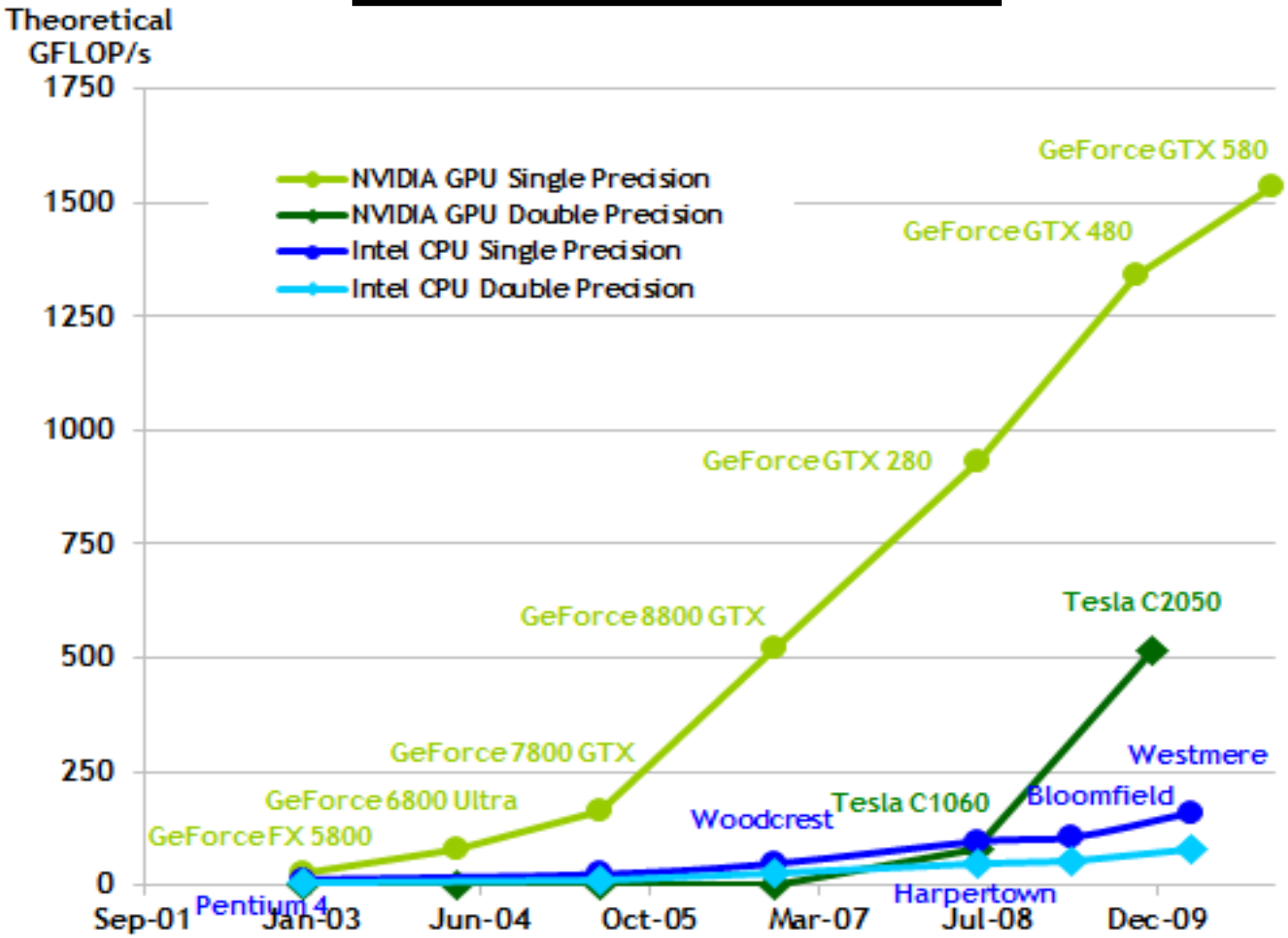
Source : NVIDIA, References given in the presentation

Part-1

CUDA enabled NVIDIS GPUs

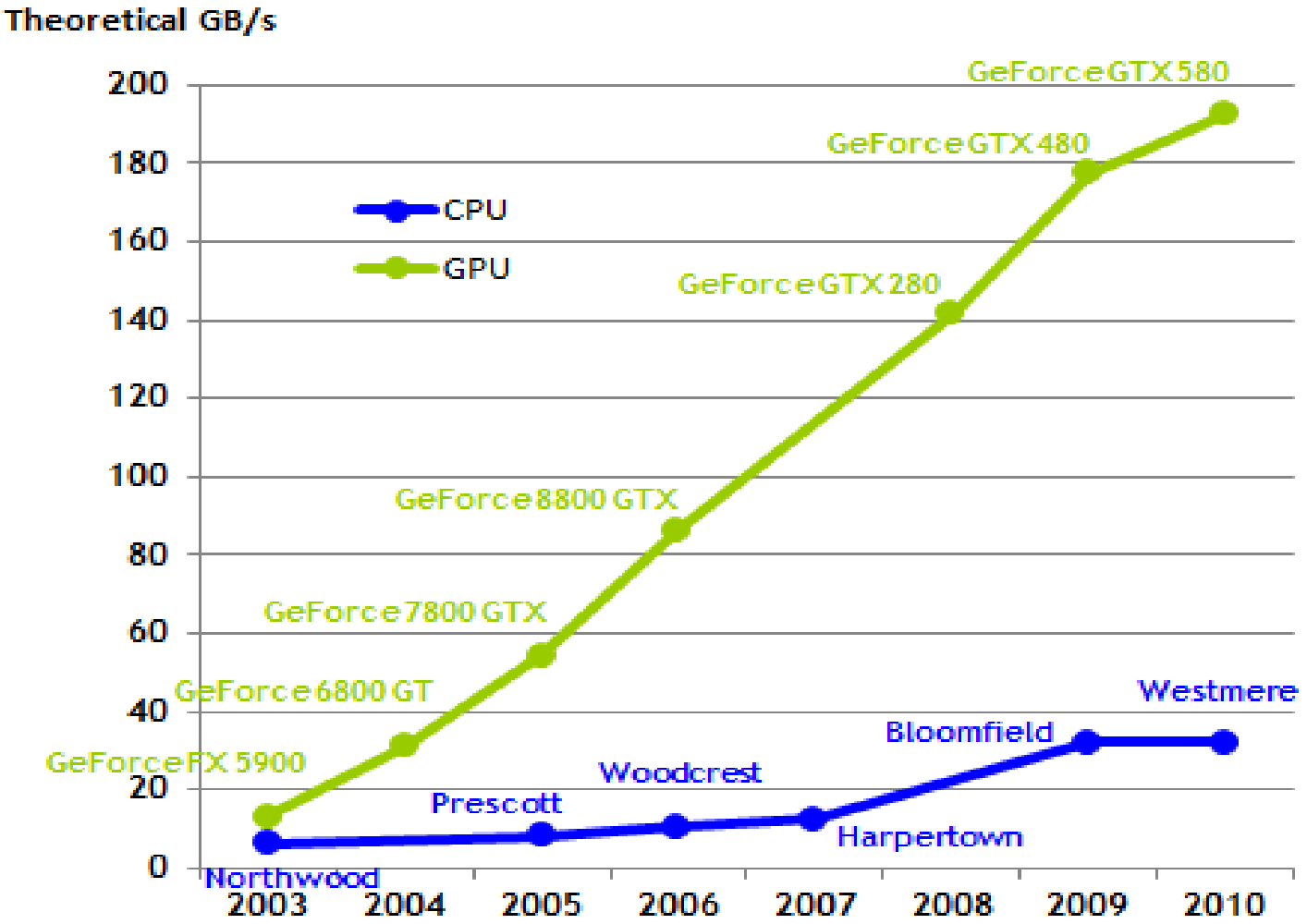
Source & Acknowledgements : NVIDIA, References

Computing - CPU/GPU



Source & Acknowledgements : NVIDIA, References

Computing - CPU/GPU



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

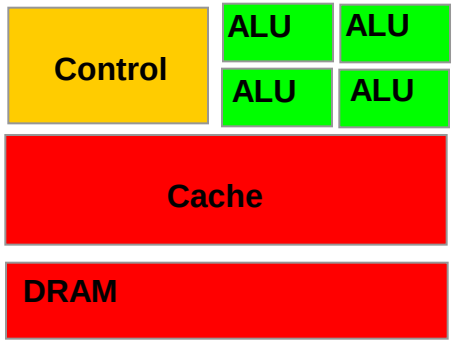
Source & Acknowledgements : NVIDIA, References

Why Are GPUs So Fast?

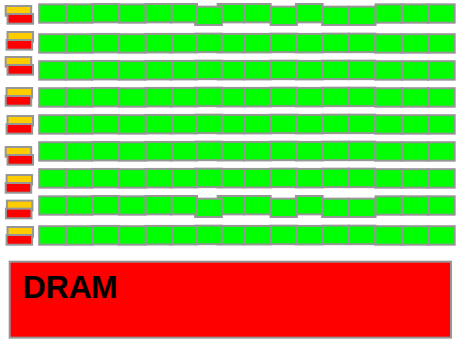
- GPU originally specialized for math-intensive, highly parallel computation
- So, more transistors can be devoted to data processing rather than data caching and flow control

AMD

NVIDIA



CPU



GPU

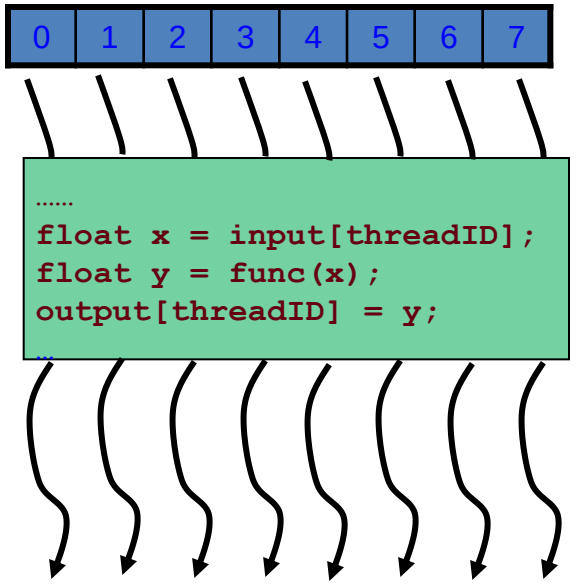
- Commodity industry: provides economies of scale
- Competitive industry: fuels innovation

Source : NVIDIA, References

GPU Computing : Think in Parallel

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



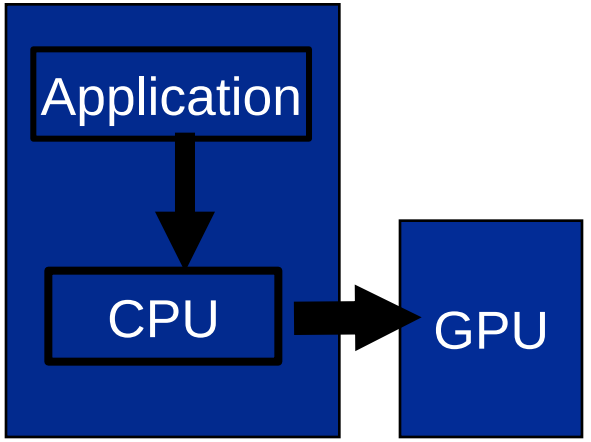
GPU Computing : Think in Parallel

- Performance = parallel hardware

  +

  scalable parallel program

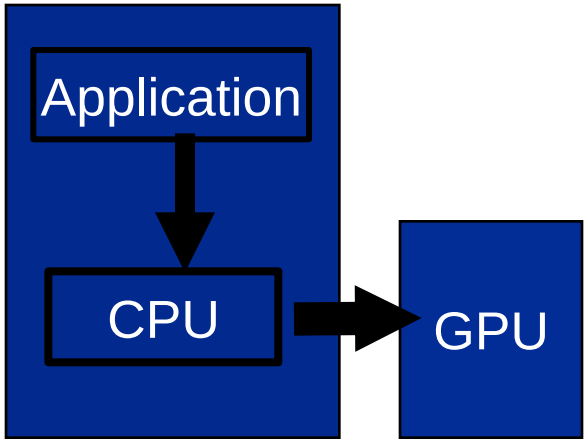
- GPU Computing drives new applications
  - Reducing “Time to Discovery”
  - 100 x Speedup changes science & research methods
- New applications drive the future of GPUs
  - Drives new GPU capabilities
  - Drives hunger for more performance



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

GPU Computing : Think in Parallel

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 : 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 & Acknowledgements : NVIDIA, References

GPU Computing : Think in Parallel

GPU Computing : Use Parallelism Efficiently

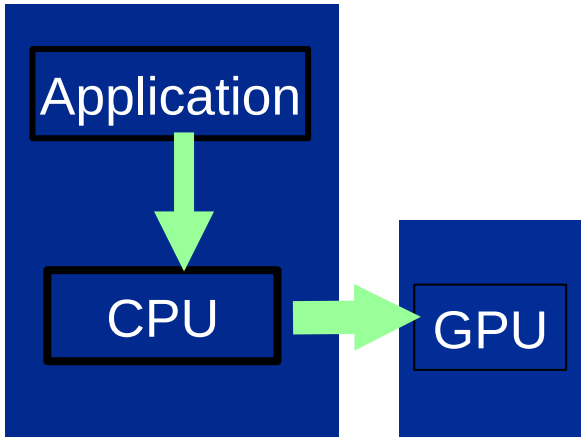
- Partition your computation to keep the GPU multiprocessors equally busy
  - Many threads, many thread blocks
- Keep resource usage low enough to support multiple active thread blocks per multiprocessor
  - Registers, shared memory

Source : NVIDIA,AMD, References

GPU 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



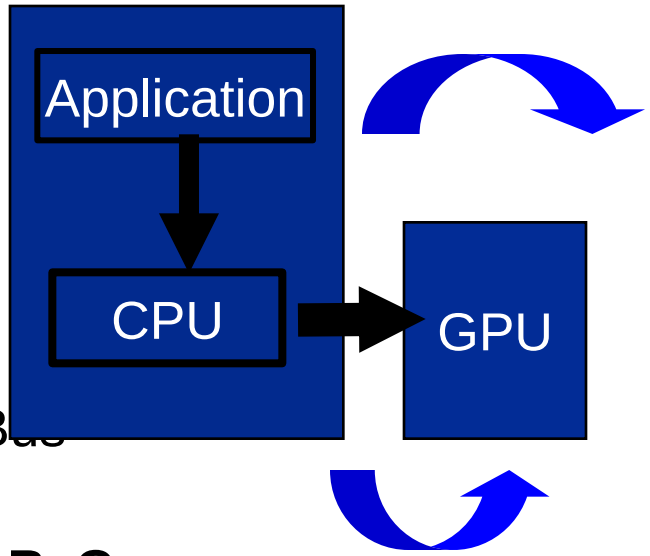
Source & Acknowledgements : NVIDIA, References

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.

Source : NVIDIA GPUs

GPU Programming : Two Main Challenges

Challenge 2 : Accuracy

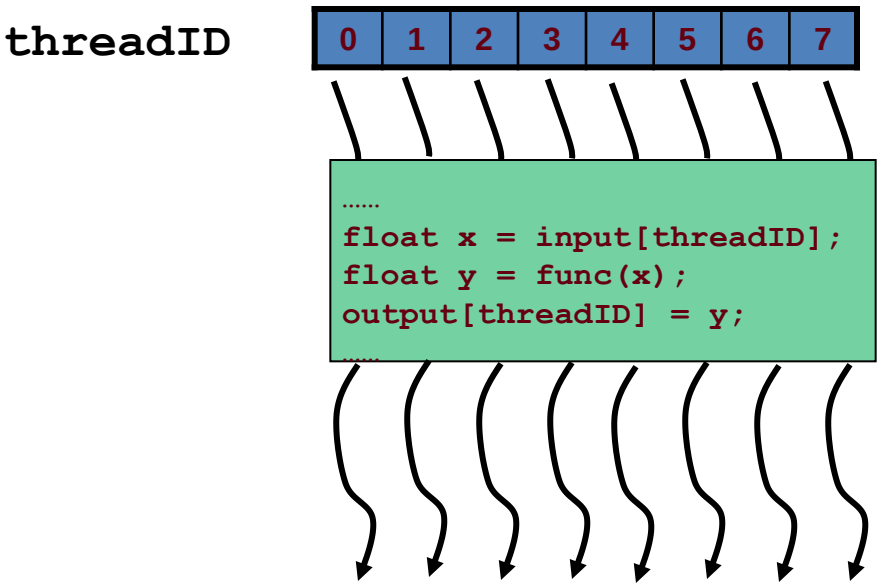
- Example : Non-Scientific Computation - Video Games (Frames) (A single bit difference in a rendered pixel in a real-time graphics program may be discarded when generating subsequence frames)
- Scientific Computing : Single bit error - Propagates overall error
- **Past History :** Most GPUs support single/double 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

Source : NVIDIA

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

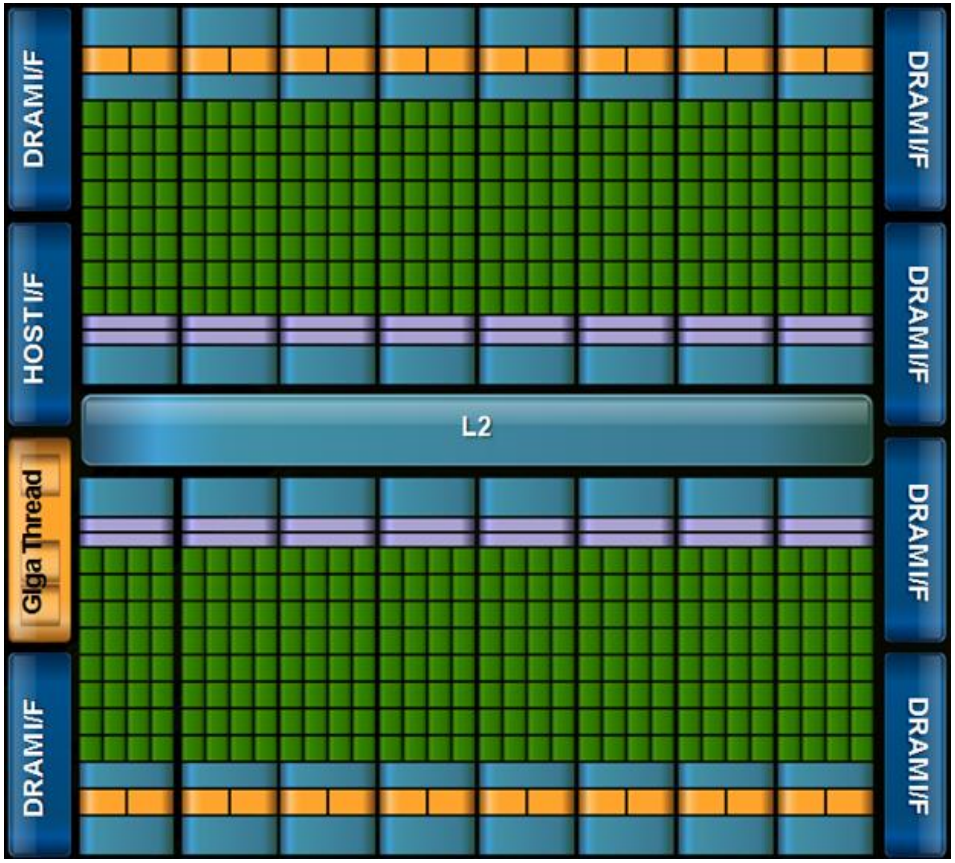
GPU : Architecture

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

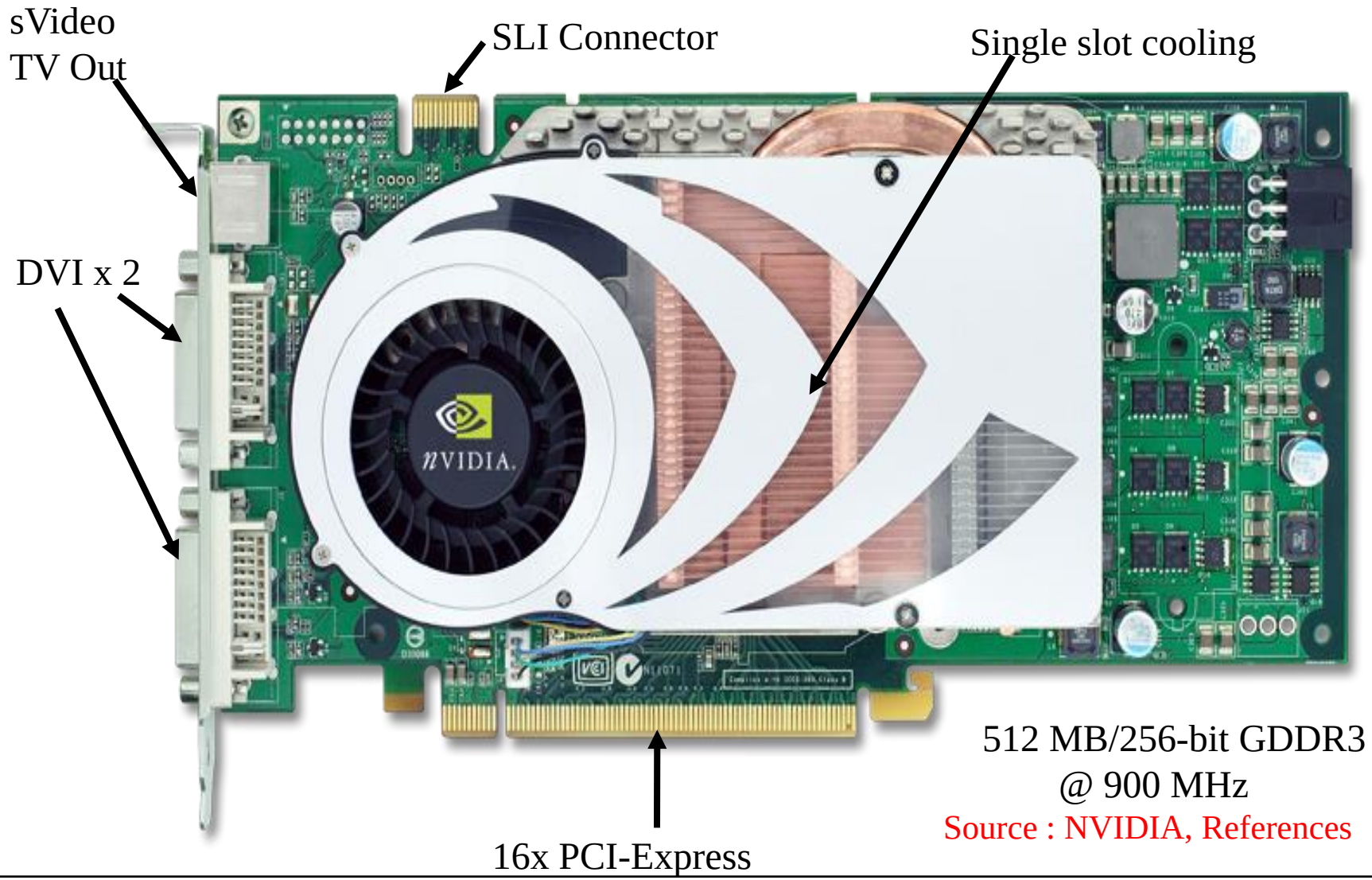
GeForce 8800

Tesla D870

Tesla S870

Source : NVIDIA, References

GeForce 8800 GT Card



512 MB/256-bit GDDR3
@ 900 MHz

Source : NVIDIA, References

GPU Thread Organisation

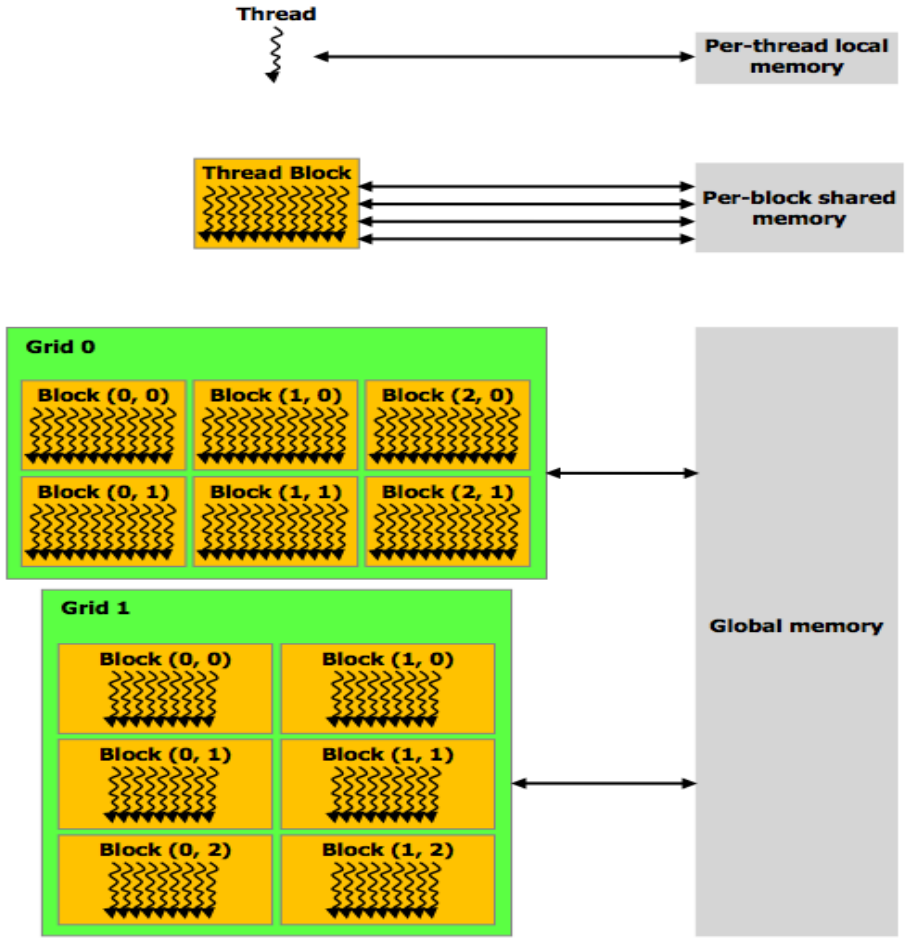
Reflects the memory hierarchy of the device

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

❖ ***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

NVIDA :CUDA – Data Parallelism

❖ ***Data Parallelism***

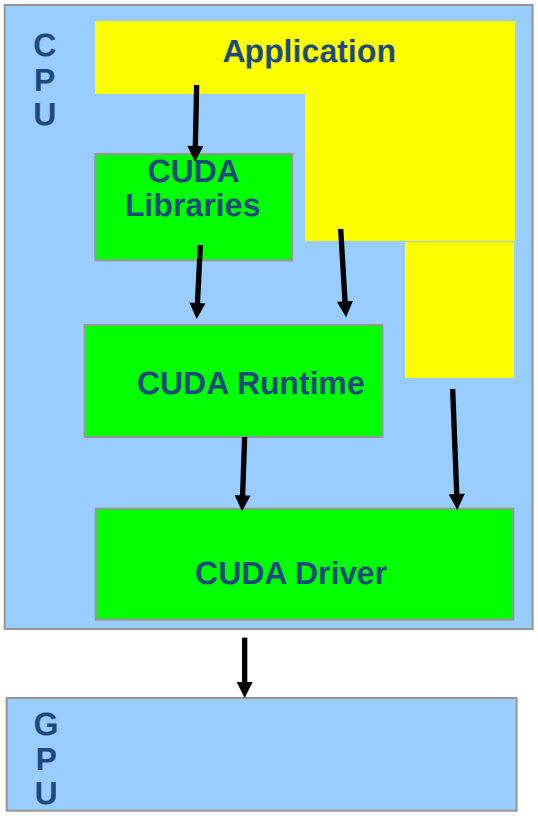
- It refers to the program property whereby many arithmetic operations can be safely performed on the data structure in a simultaneous manner.

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

Source & Acknowledgements : NVIDIA, References

NVIDIA GPU Computing - CUDA Kernels and Threads

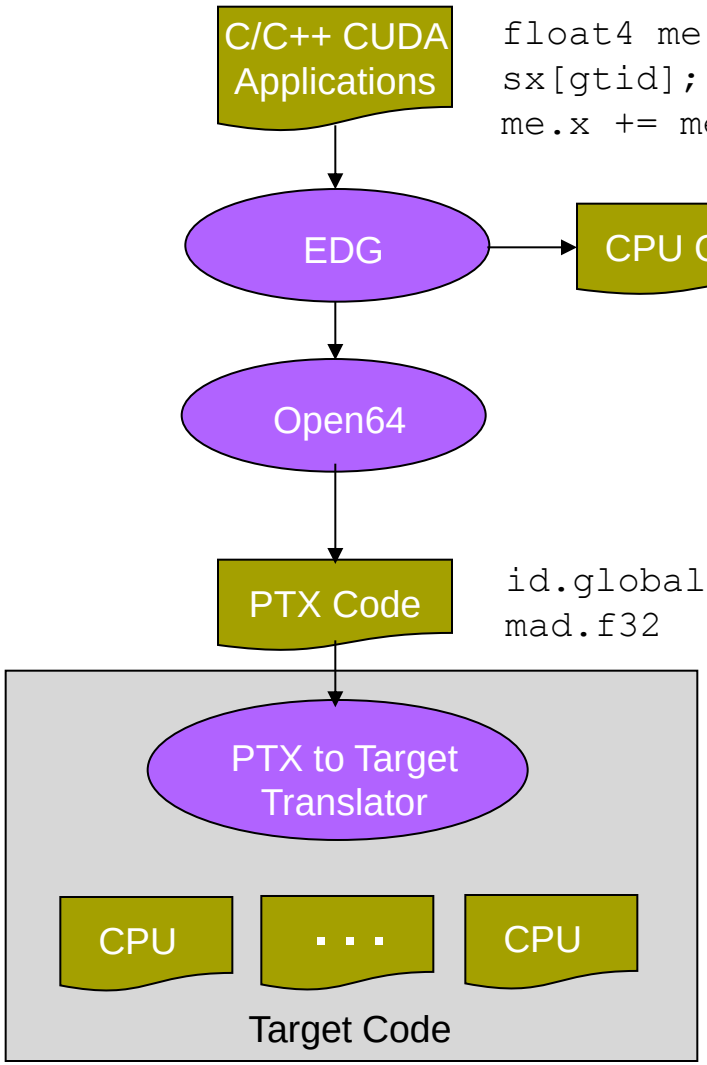
- **NEW:** *GPU Computing* with CUDA
  - CUDA = Compute Unified Device Architecture
  - Co-designed hardware & software for direct GPU computing
- *Hardware: fully general data-parallel architecture*
  - General thread launch; Global load-store
  - Parallel data cache
- *Software: program the GPU in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

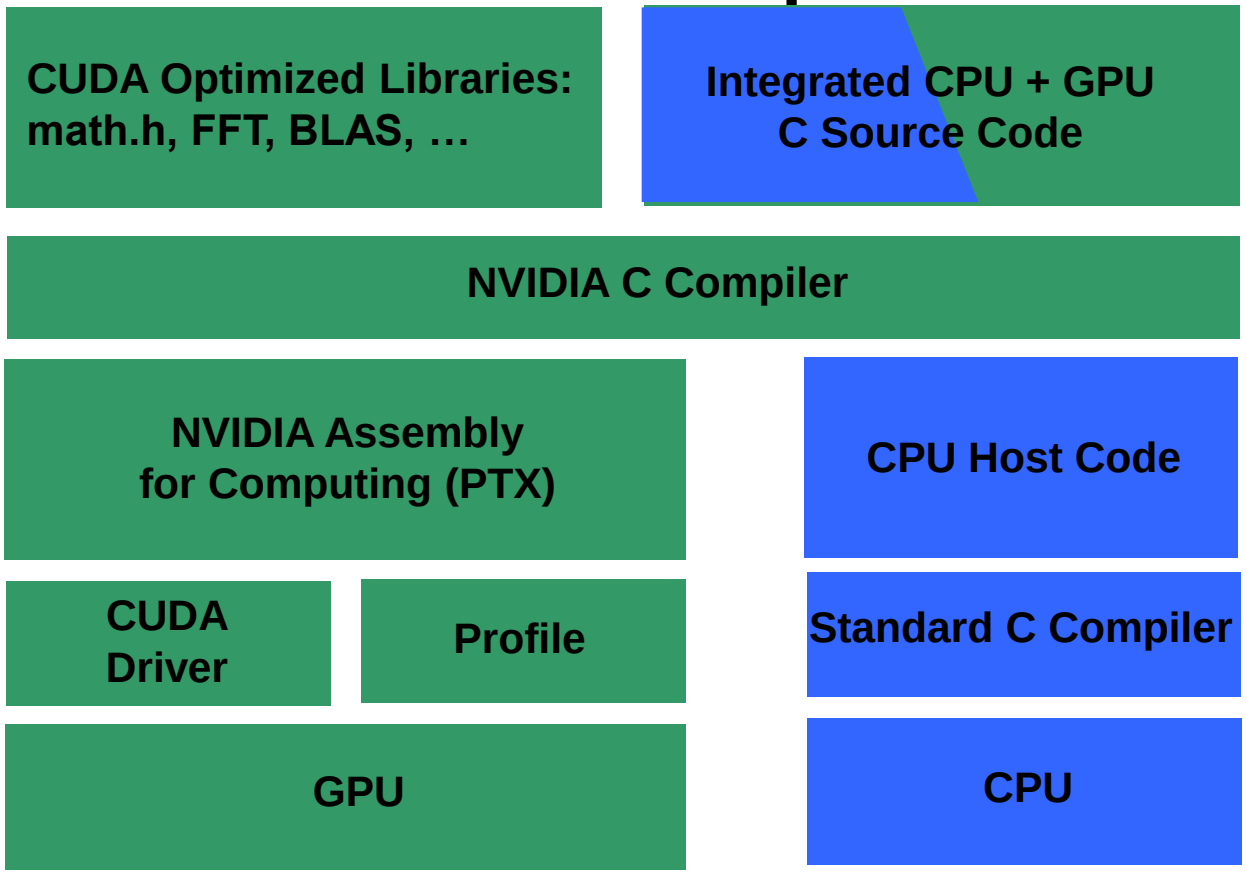
NVIDIA GPU Computing - CUDA Kernels and Threads



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.

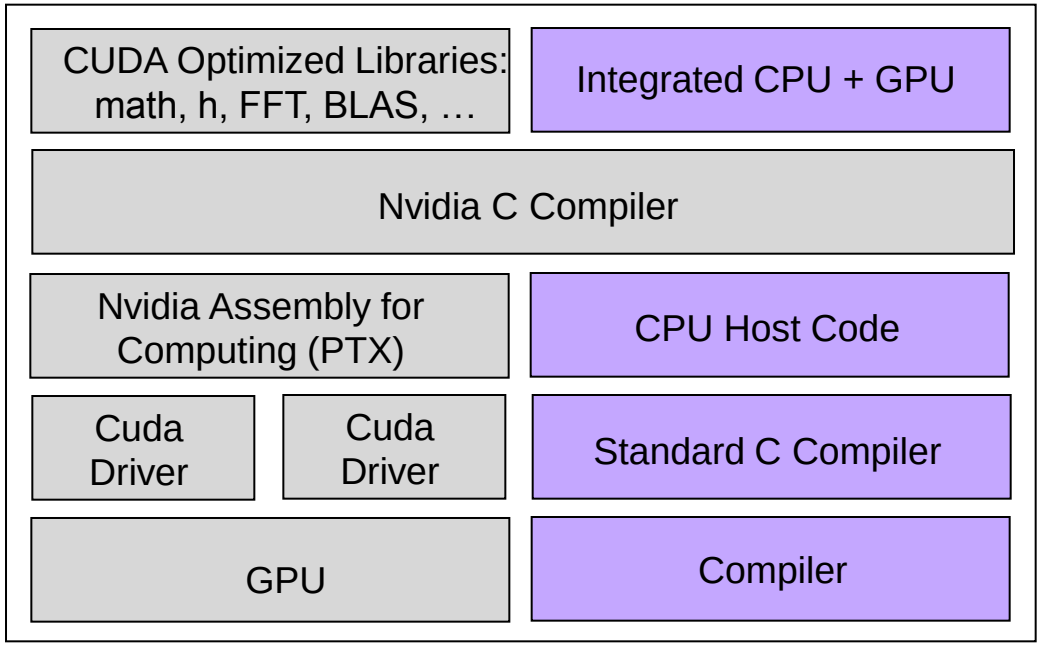
Source & Acknowledgements : NVIDIA, References

CUDA Software Development



Source & Acknowledgements : NVIDIA, References

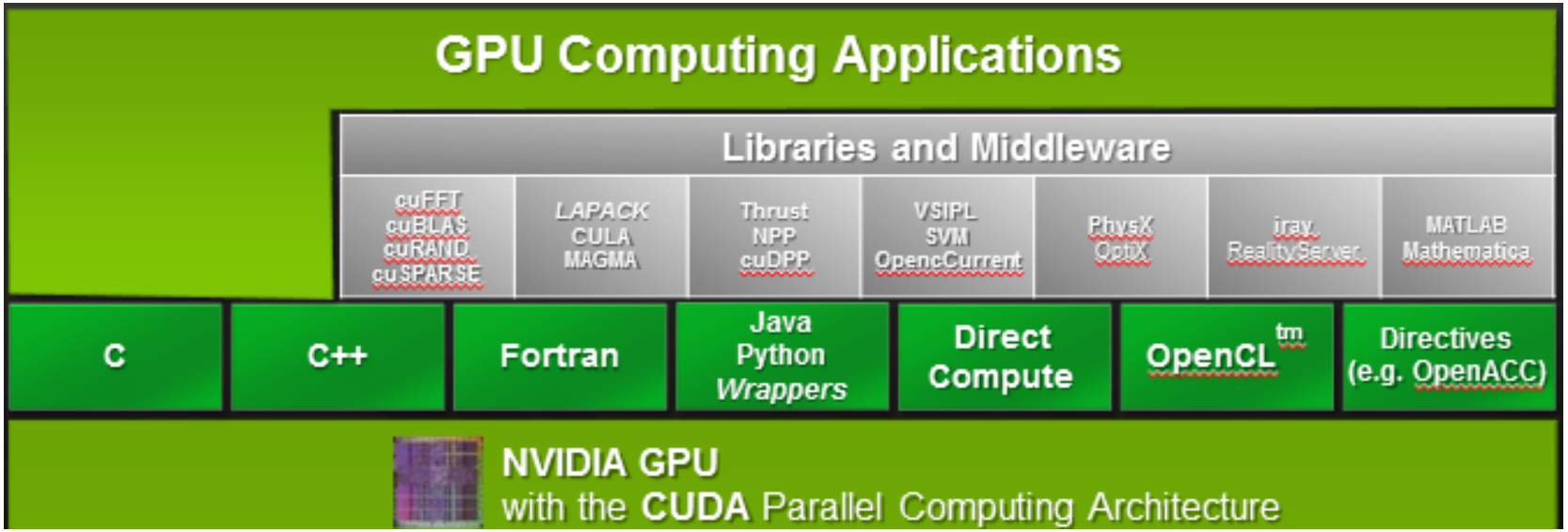
CUDA Performance Advantage



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

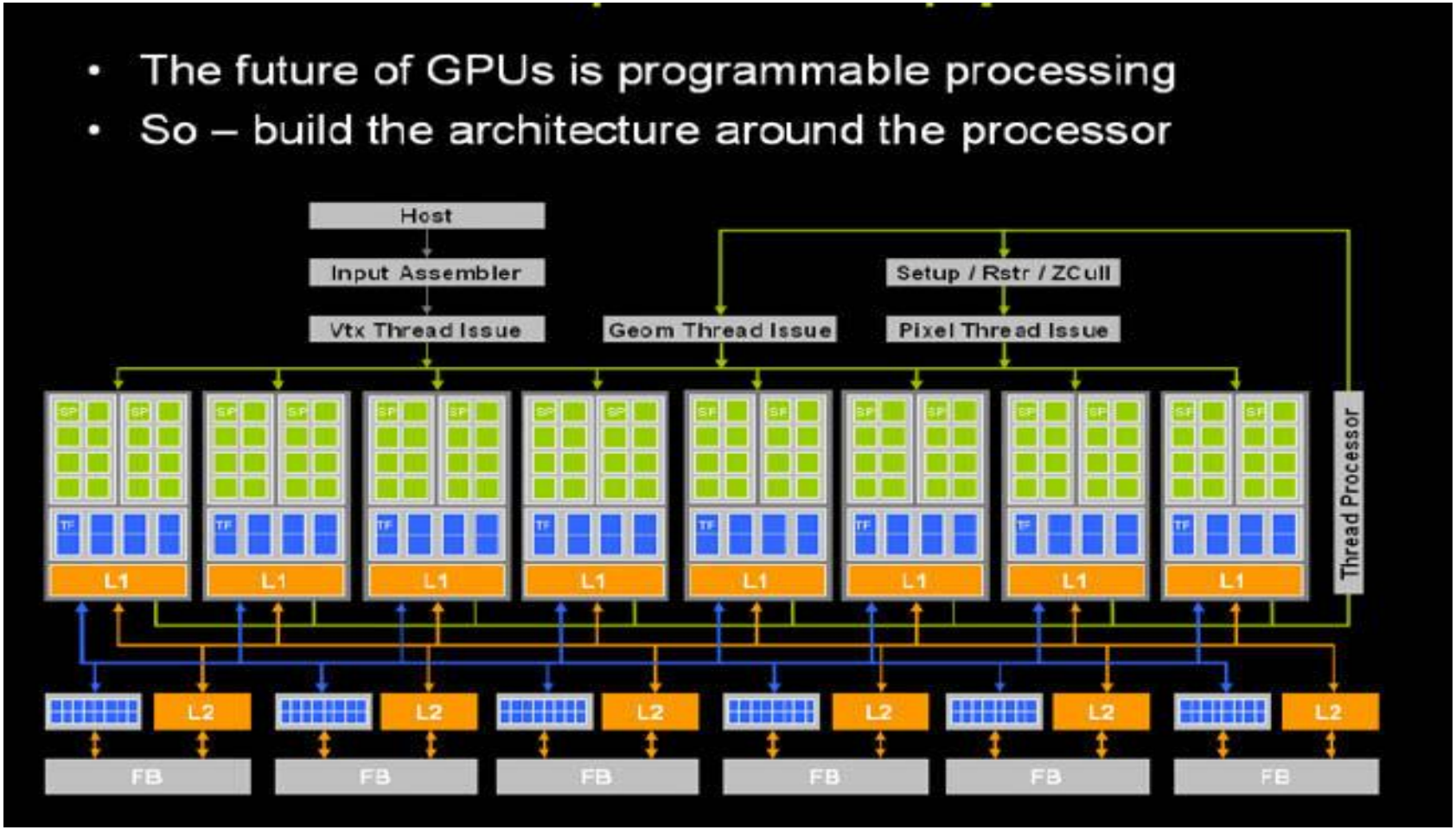
NVIDIA GPU Computing - CUDA Kernels and Threads



CUDA is Designed to Support Various Languages and Application Programming Interfaces

Source & Acknowledgements : NVIDIA, References

NVIDIA GeForce GPU



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

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

- ***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.

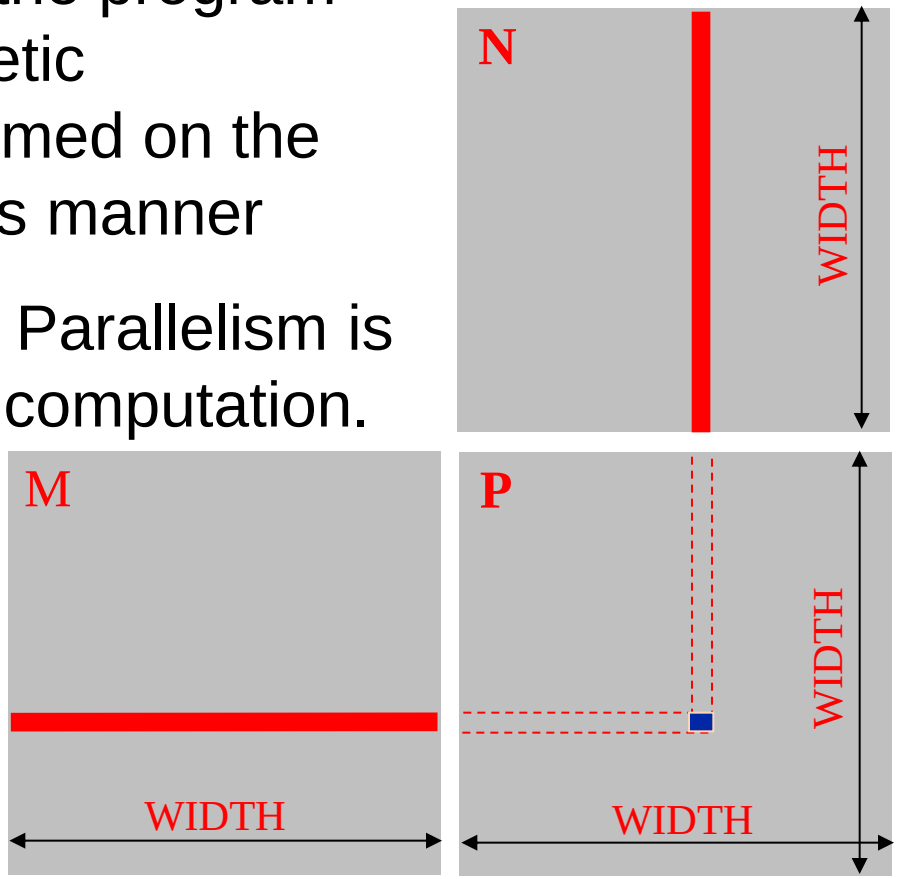


Figure Data parallelism in matrix multiplication.

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.

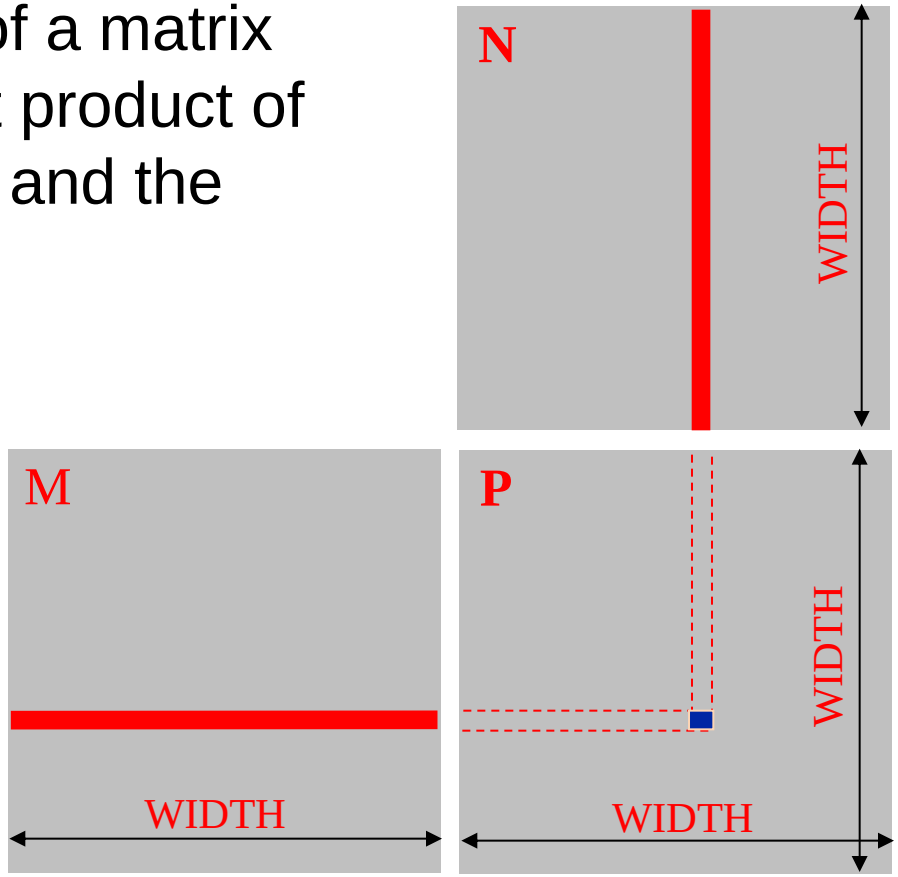


Figure Data parallelism in matrix multiplication.

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

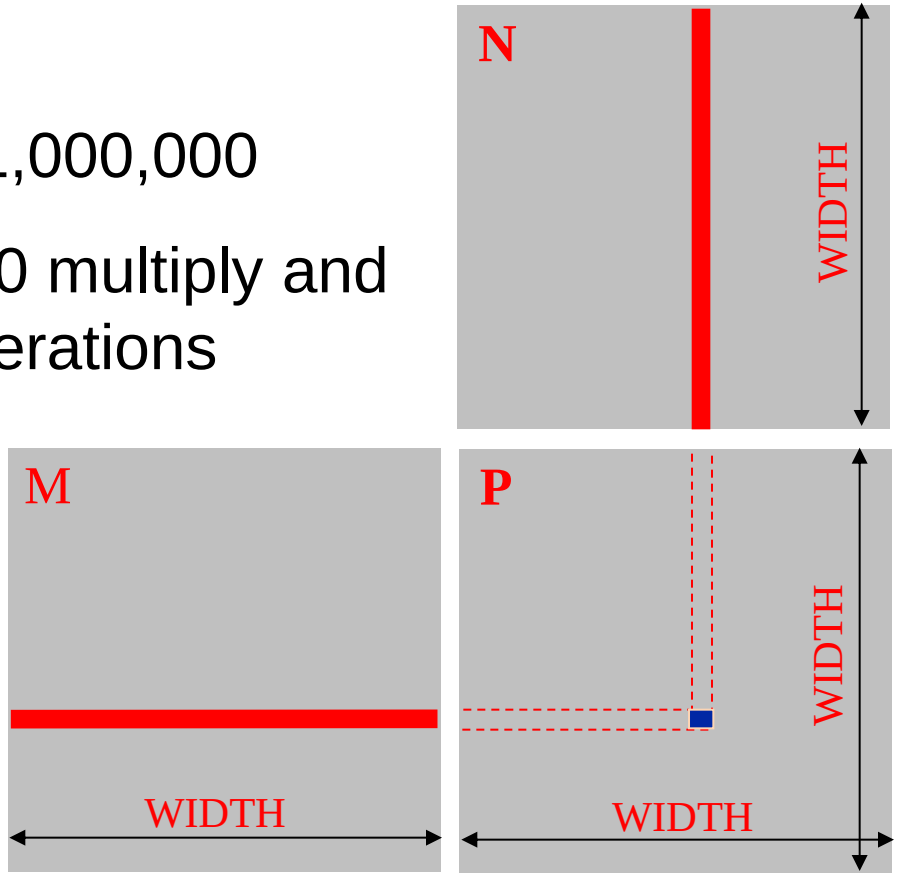


Figure : Data parallelism in matrix Multiplication.

Source:

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 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

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 :

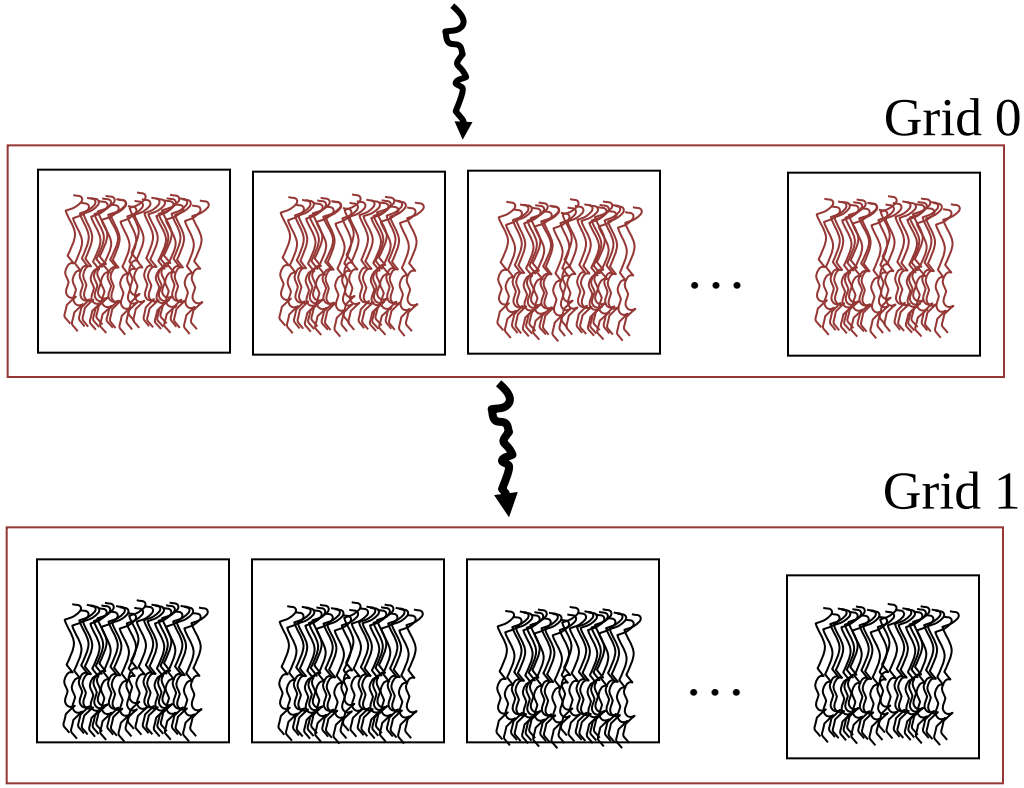
- CUDA threads are of much lighter weight than the CPU threads
- It can be assumed that these threads take **very few cycles** to generate and schedule due to efficient hardware support.
  - Note : CPU threads that typically require thousands of clock cycles to generate and schedule.
  - When kernel 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

CUDA PROGRAM STRCUTURE

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 :CUDA STRUCTURE

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;
    }
}
```

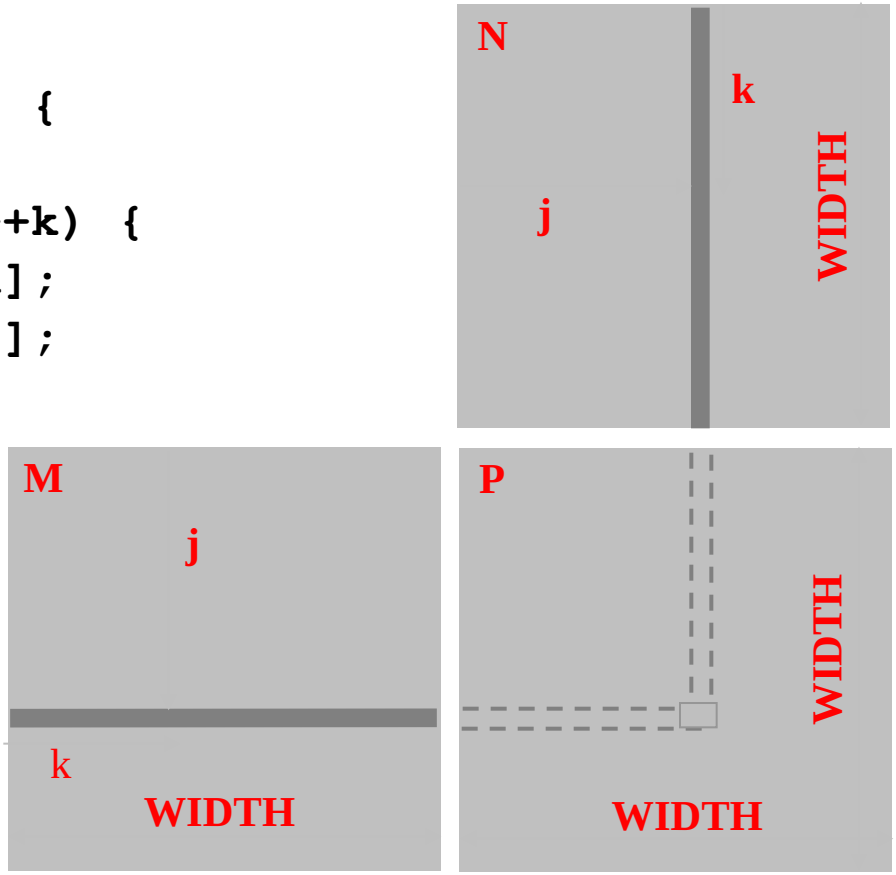


Figure A simple matrix multiplication function with only host code.

NVIDIA :CUDA STRUCTURE

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.

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

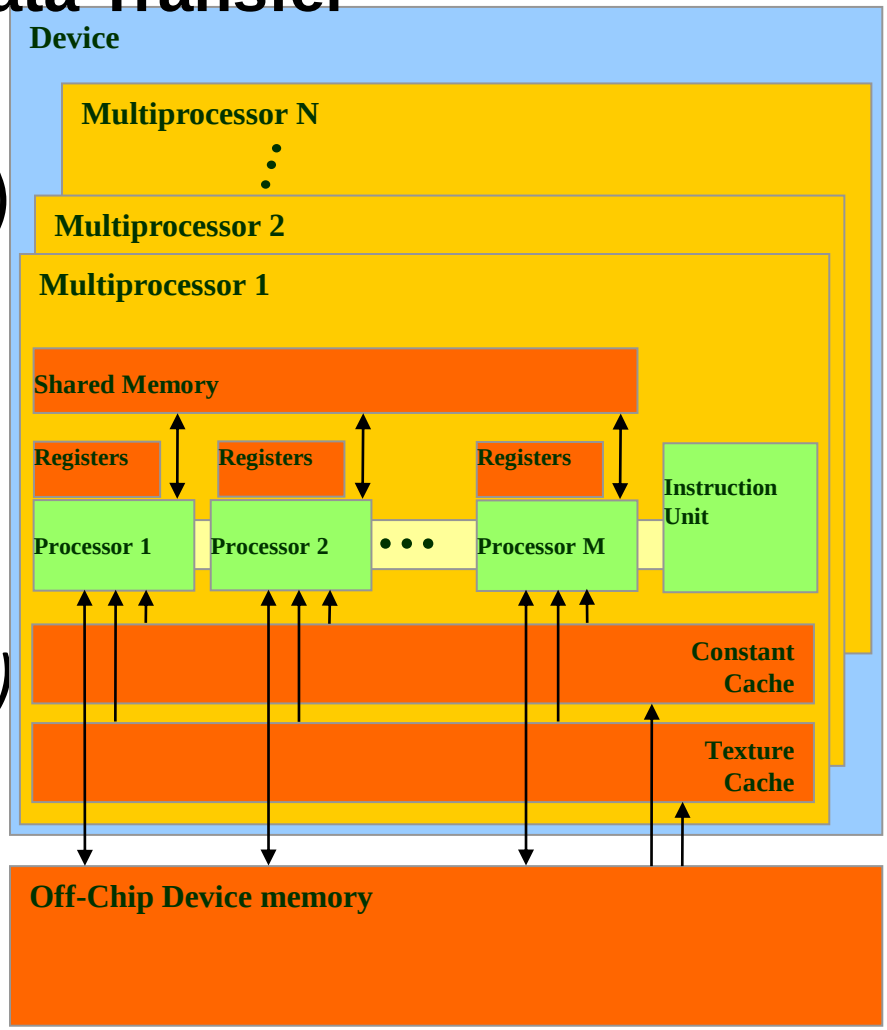
CUDA Architecture

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

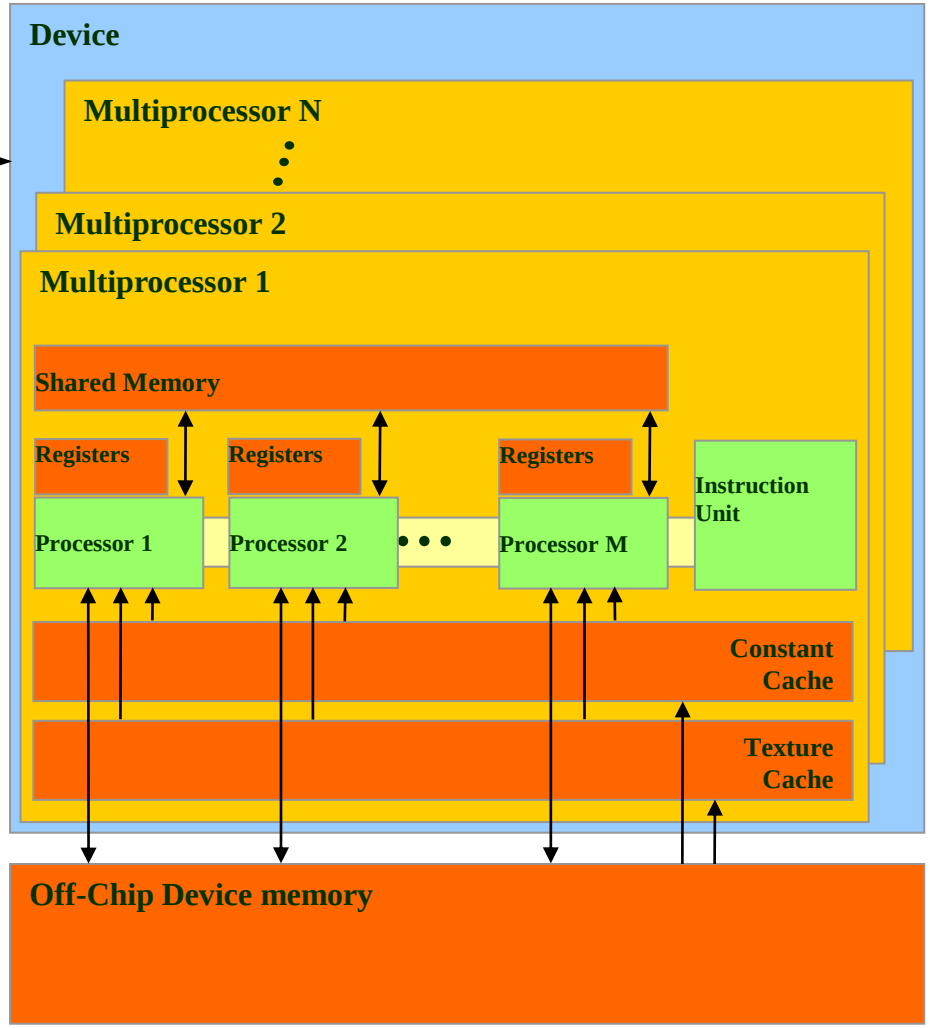
CUDA Device Memories and Data Transfer

Host CPU

CPU initialize data

Launches kernel

Threads work on sub-streams



Source & Acknowledgements : NVIDIA, References

Source & Acknowledgements : NVIDIA, References

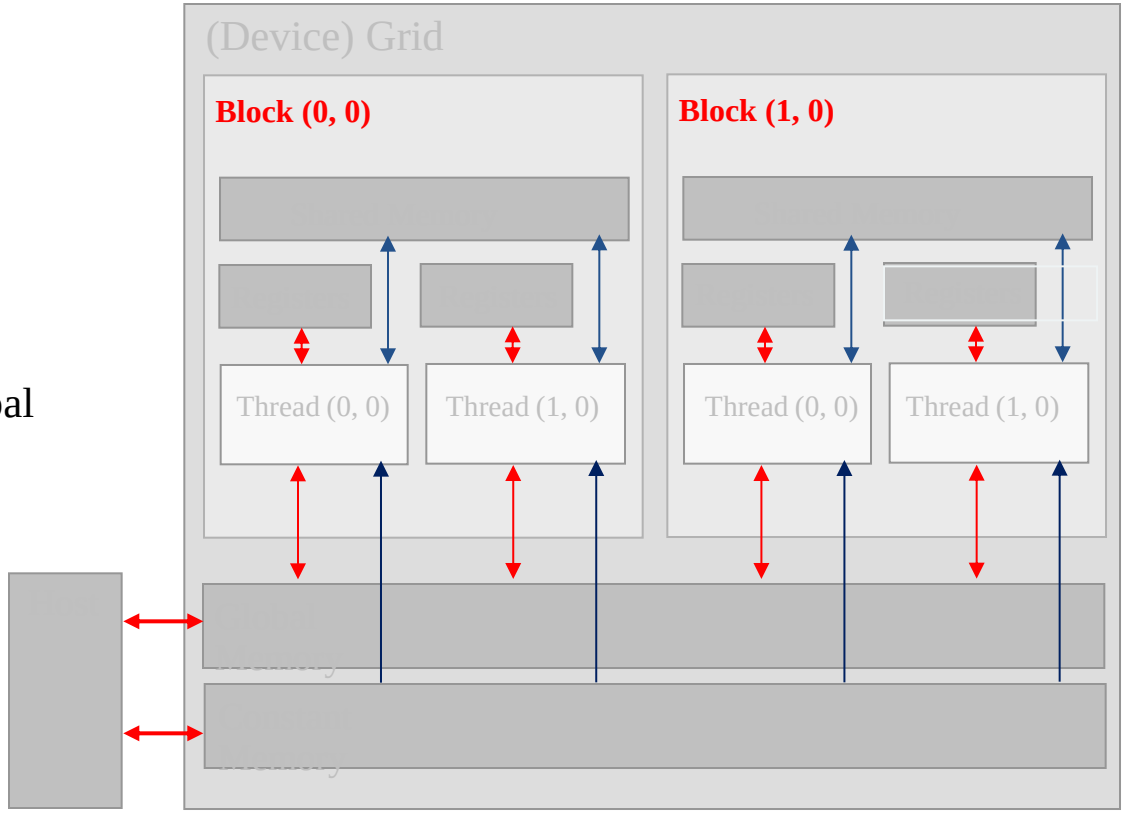
NVIDA :CUDA DEVICE MEMORIES & DATA TRANSFER

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



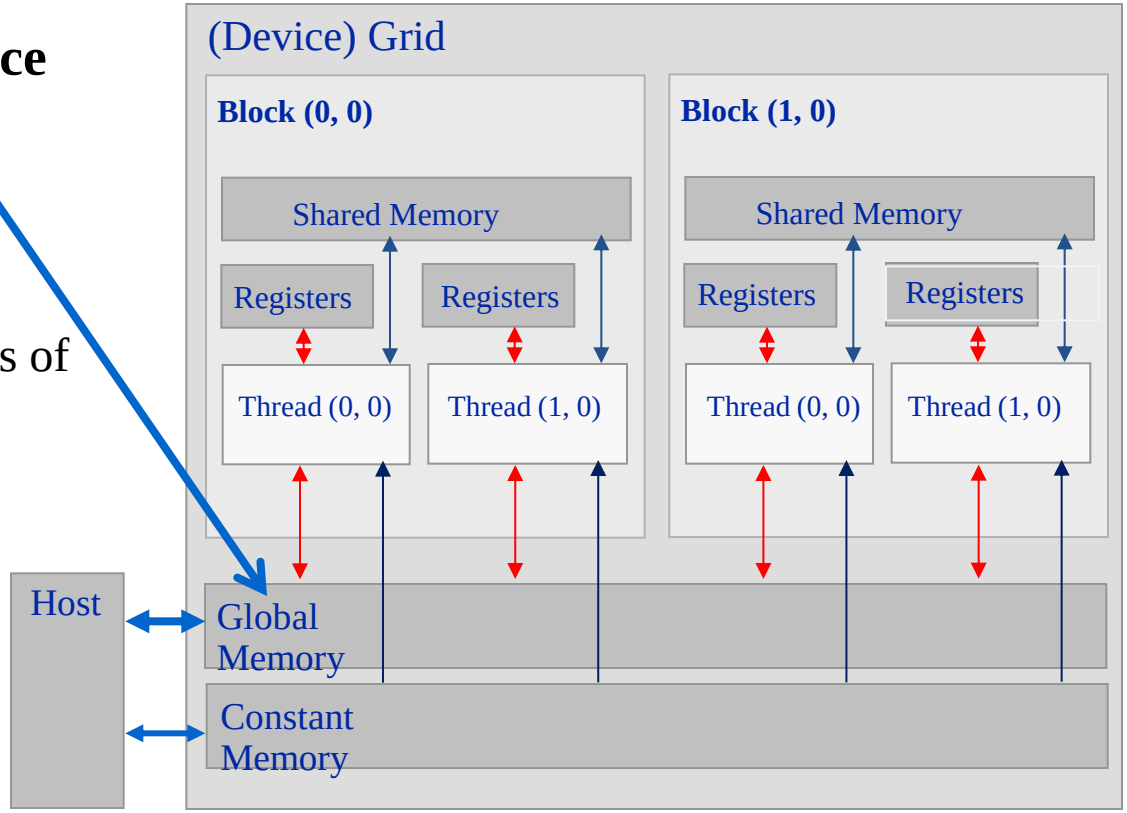
Host memory is not shown in the figure

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

NVIDIA :CUDA STRUCTURE

Example : Matrix Multiplication

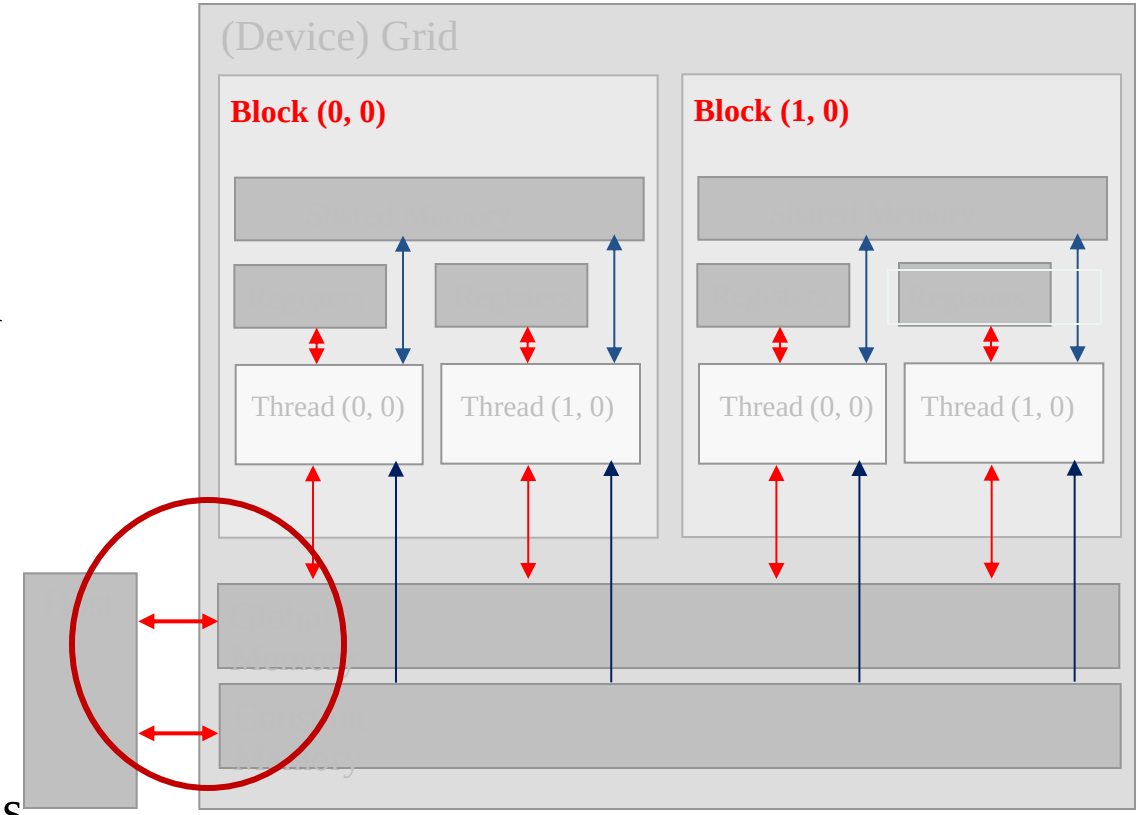
```
Void MatrixMultiplication(float* M,float* N,float* P,int Width)
{
   int size = Width * Width *sizeof(float);
   float* Md, Nd, Pd;
   ……..……
   Step 1: // Allocate device memory for M, N, and P
           // copy M and N to allocate device memory locations

   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

NVIDIA :CUDA STRUCTURE

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.

NVIDA :CUDA STRUCTURE

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

NVIDIA :CUDA STRUCTURE

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

NVIDA :CUDA STRUCTURE

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( )

NVIDIA :CUDA STRUCTURE

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

NVIDIA :CUDA THREAD ORGANIZATION

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
```

NVIDIA :CUDA THREAD ORGANIZATION

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

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 **`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

NVIDIA :CUDA THREAD ORGANIZATION

KERNEL FUNCTIONS AND THREADING

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

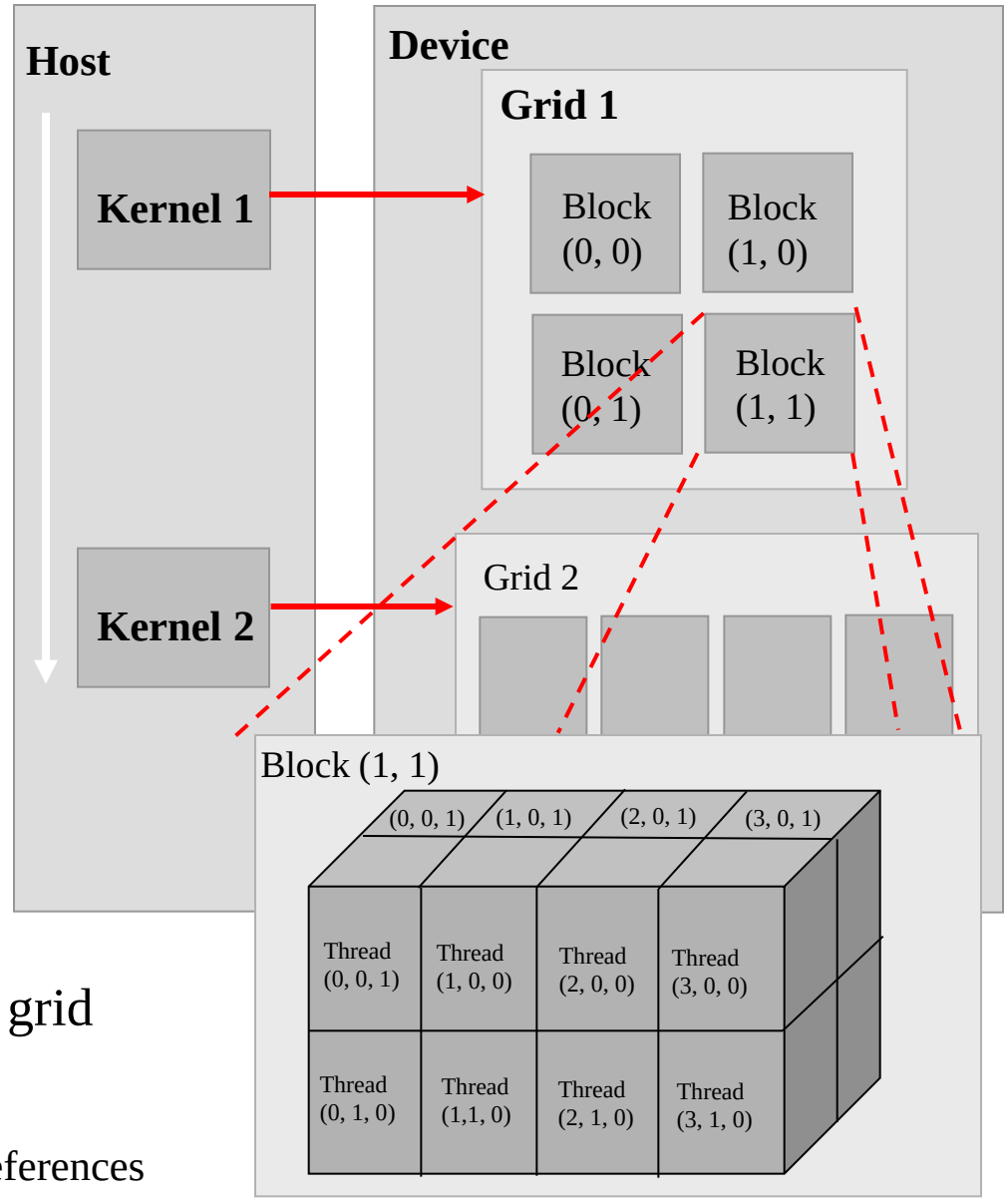
- Each thread uses its **`threadIdx.x`** and **`threadIdx.y`** to identify the row of **Md** and the column of **Nd** to perform the dot product operation.
- Each thread also uses its ***`threadIdx.x`*** and ***`threadIdx.y`*** values to select the **Pd** element that it is responsible for; for example $threadId_{2,2}$ 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

NVIDIA :KERNEL FUNCTIONS AND THREADING

- **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

NVIDIA :CUDA Thread Organisation

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

NVIDIA :CUDA THREAD ORGANIZATION

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

Source : NVIDIA

  - This gives a **Grid** one (1) a total of 4 x 16 = 64 threads

NVIDIA :CUDA THREAD ORGANIZATION

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

NVIDIA :CUDA THREAD ORGANIZATION

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

NVIDIA :CUDA THREAD ORGANIZATION

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-2

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

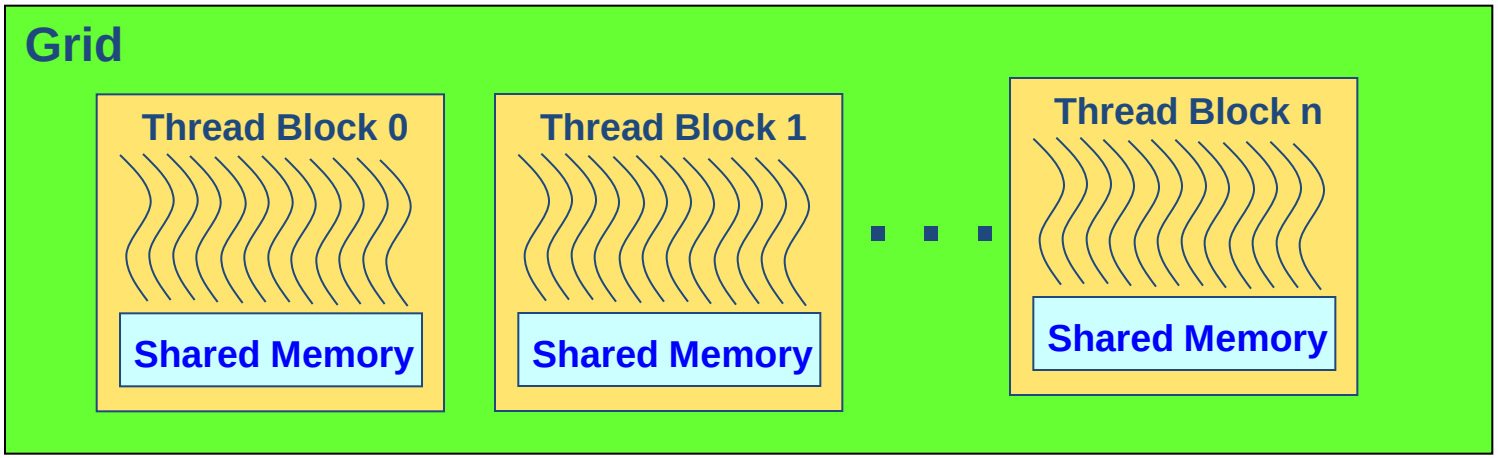
```
Kernel Function << 128, 32 >>> (…);
```

- 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 GPUs

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

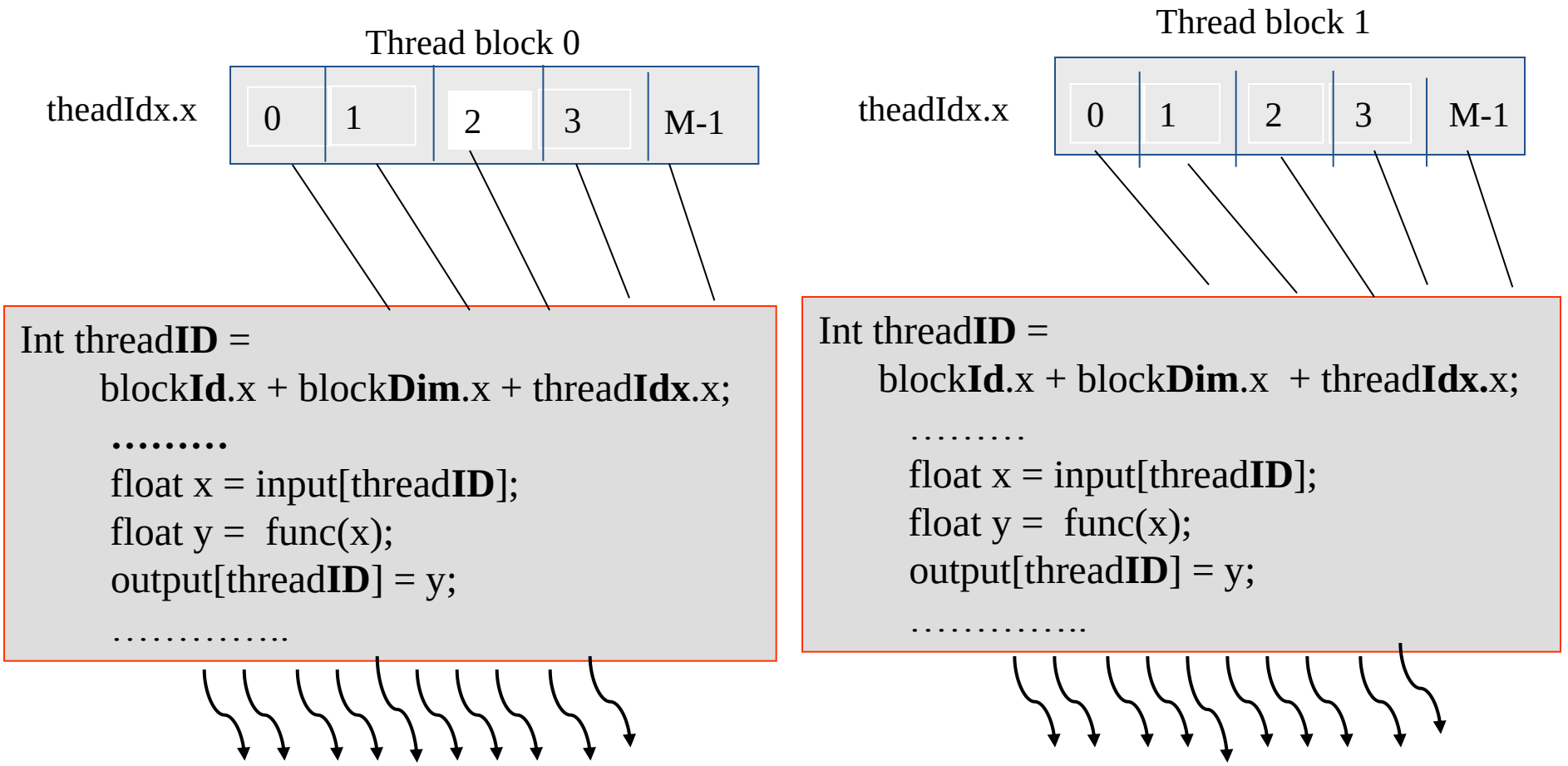
- The example figure consists of **N** thread blocks, each with a **blockIdx.x** value ranges from **0** to ***N-1***
  - Each block in-turn consists of M threads, each with a **threadIDx.x** value ranges from **0** to **M-1**.
- Each grid has a total of **N*M** threads

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

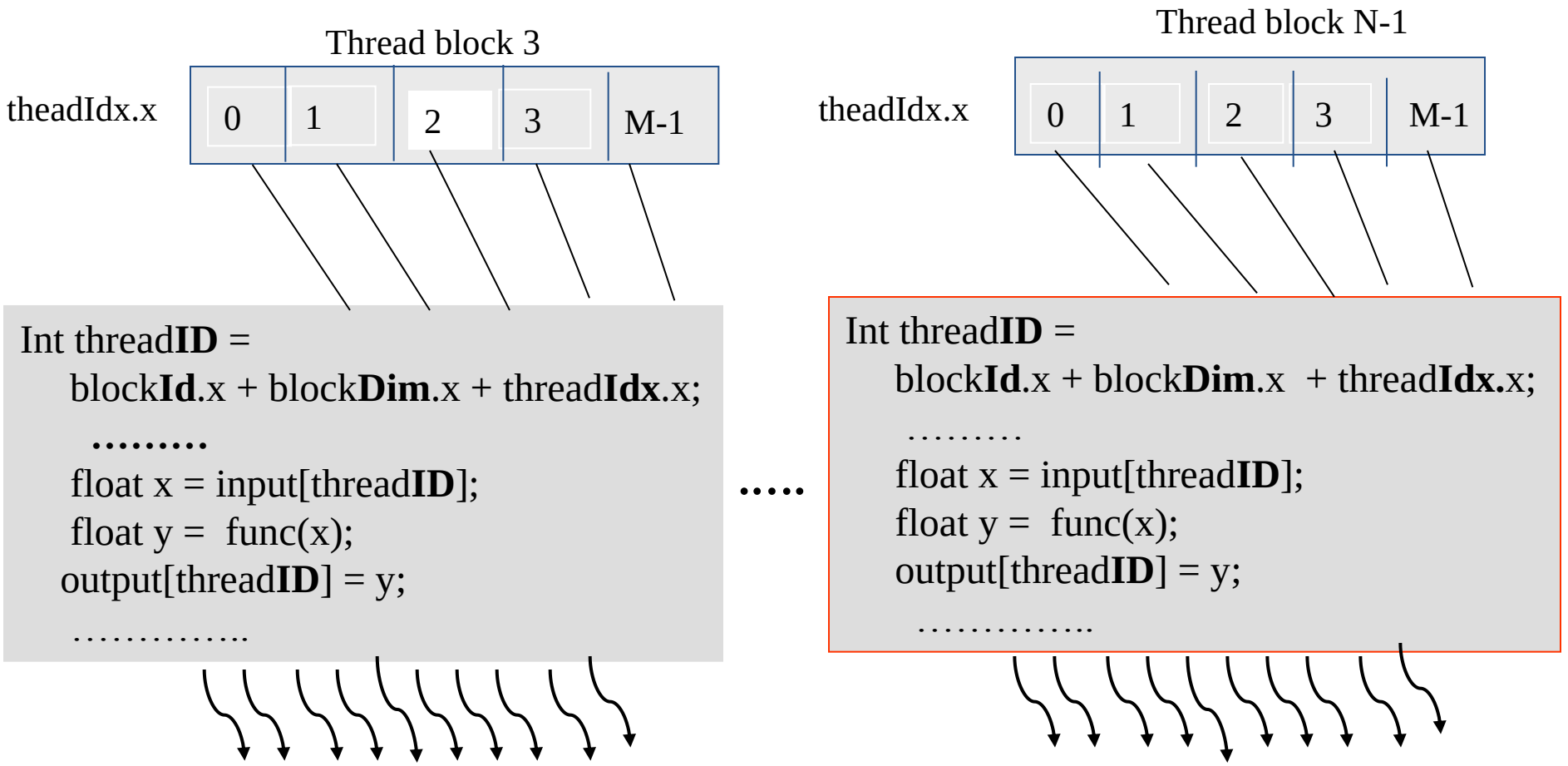
Thread **3** of Block **0** has a **threadId** value of **0*32 + 3 = 3**
Thread **3** of Block **4** has a **threadId** value of **4*32 + 3 = 131**
Thread **3** of Block **20** has a **threadId** value of **20*32 + 3 = 643**
Thread **3** of Block **40** has a **threadId** value of **40*32 +3 = 1283**
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

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 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);
```

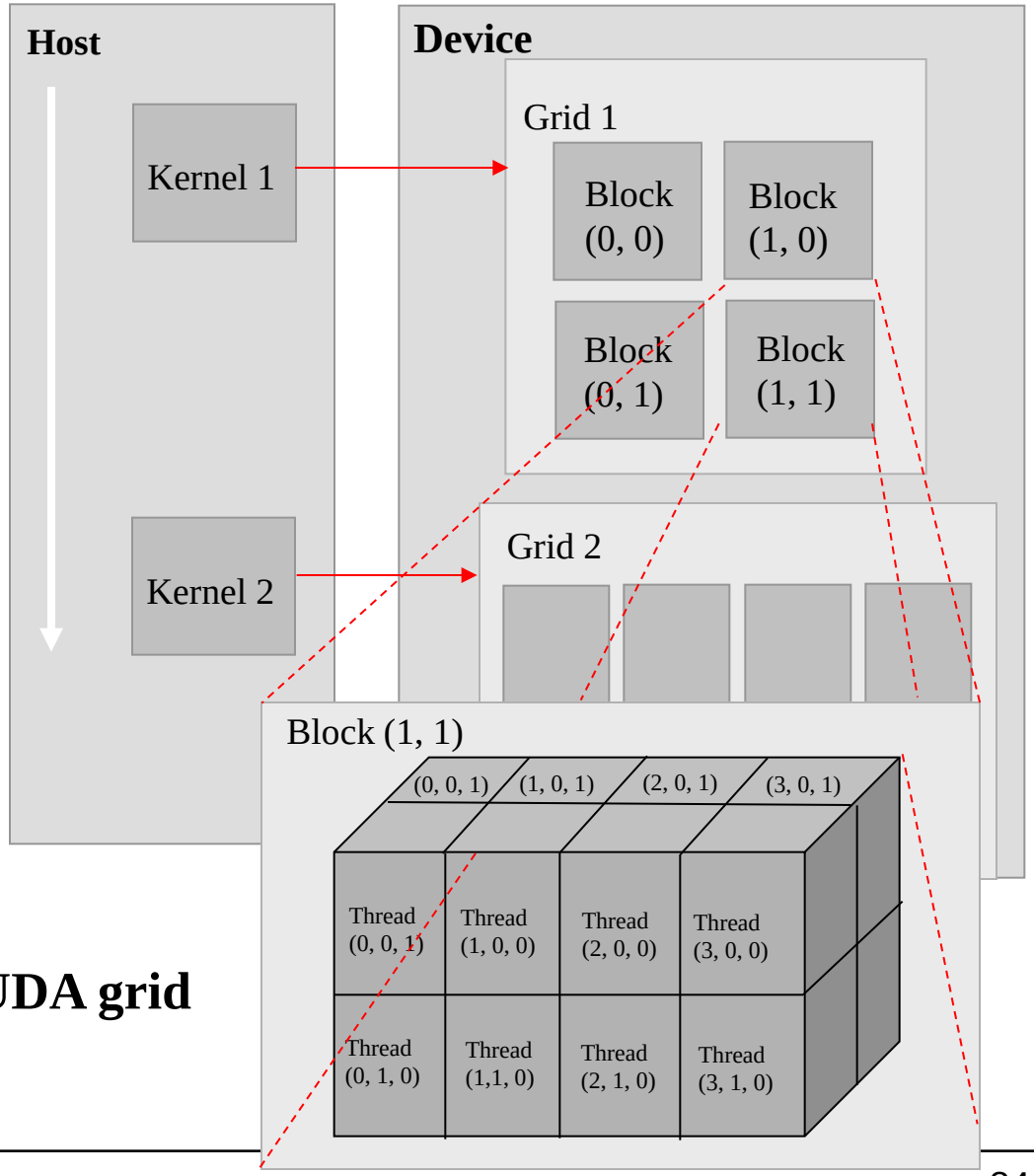
- The **first** parameter - specifies the total terms of number of blocks
- The **second** and **third** parameter specifies the number of threads in each dimension
- The configuration parameter can be accessed as a pre-defined `C struct` variable, **`blockDim`**

- 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

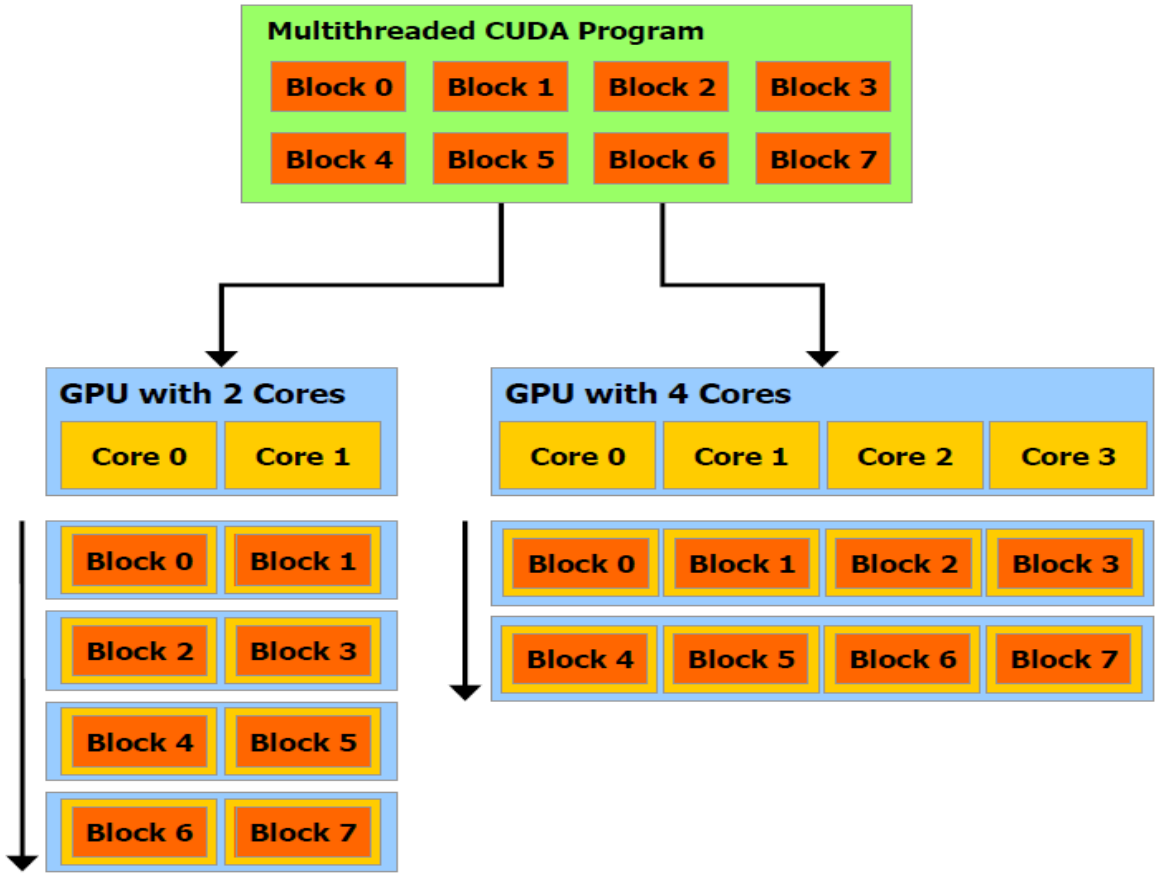
```
Dim3 dimGrid(2, 1,1);
Dim3 dimBlock(4,2,1,);
Kernel Function
 <<<
   dimGrid, dimBlock
    >>>
 (……);
```



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

NVIDIA :CUDA – Thread Organization

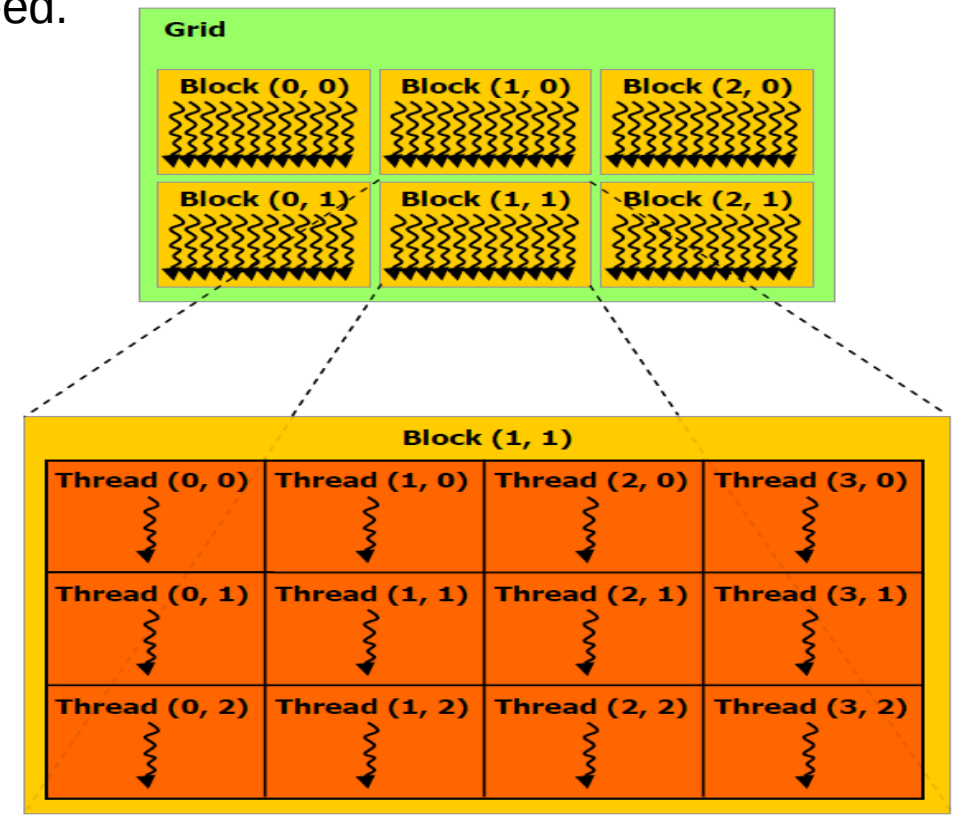
Automatic Scalability : A multi-threaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more cores will automatically execute the program in less time than a GPU with fewer cores.



Source & Acknowledgements : NVIDIA, References

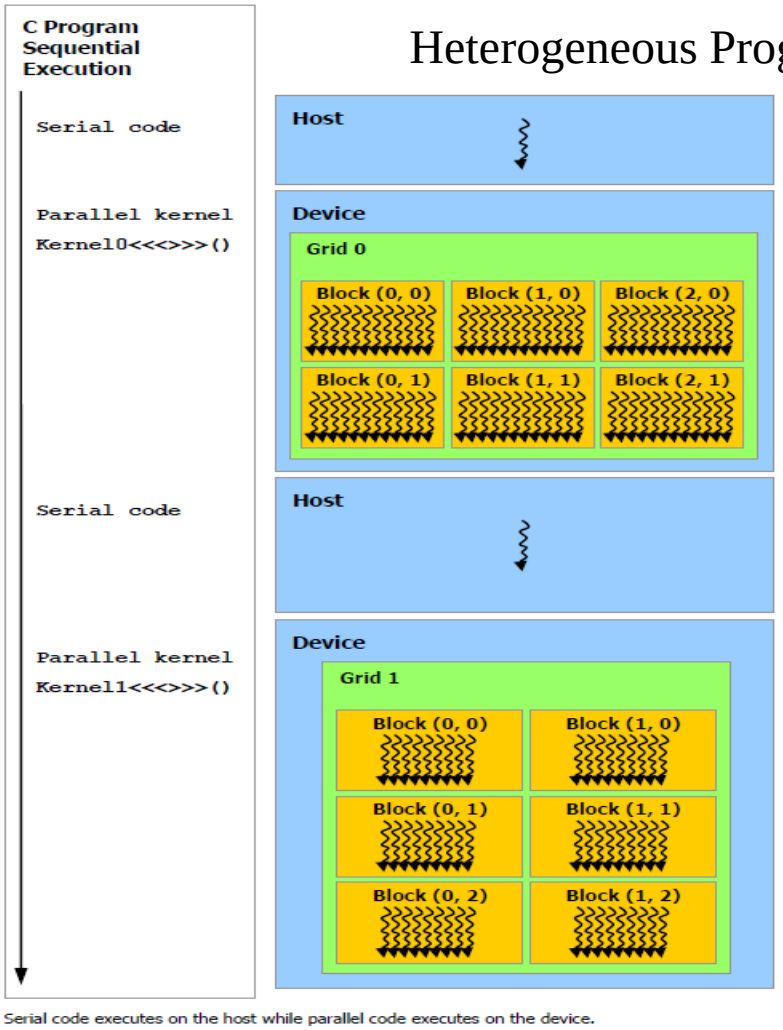
NVIDIA :CUDA – Thread Organization

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



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

Source & Acknowledgements : NVIDIA, References

Part-3

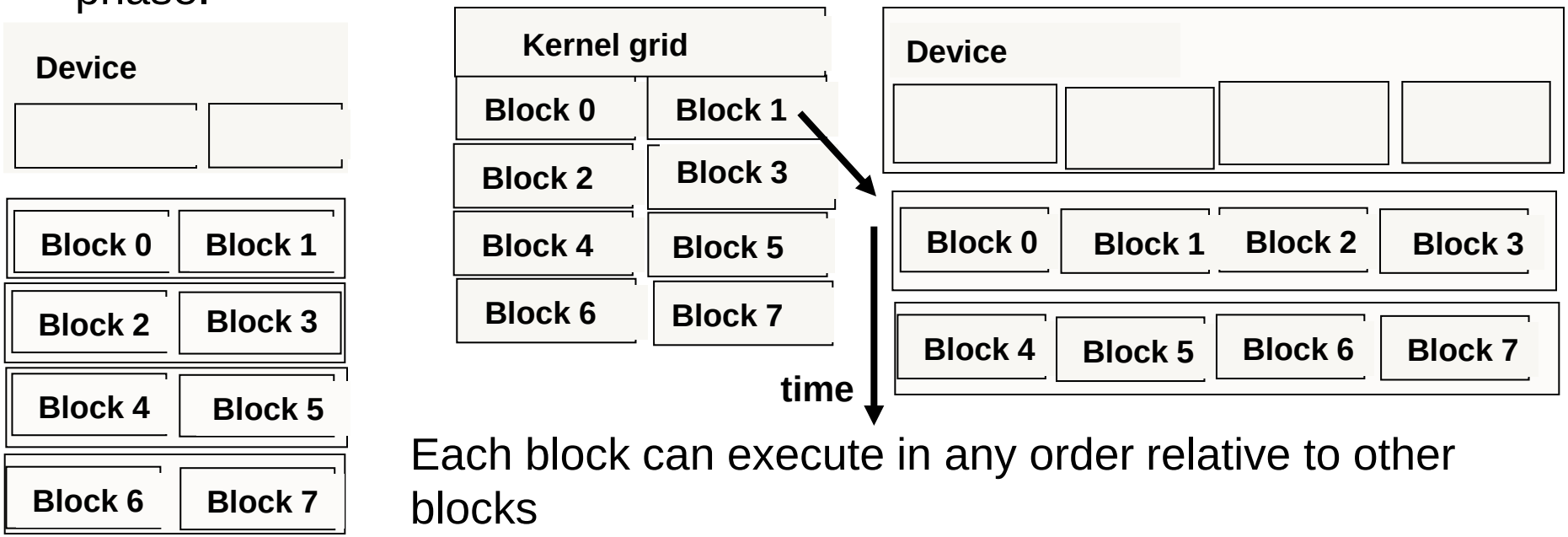
CUDA Synchronization

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

NVIDIA : CUDA Threads Organisation

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.

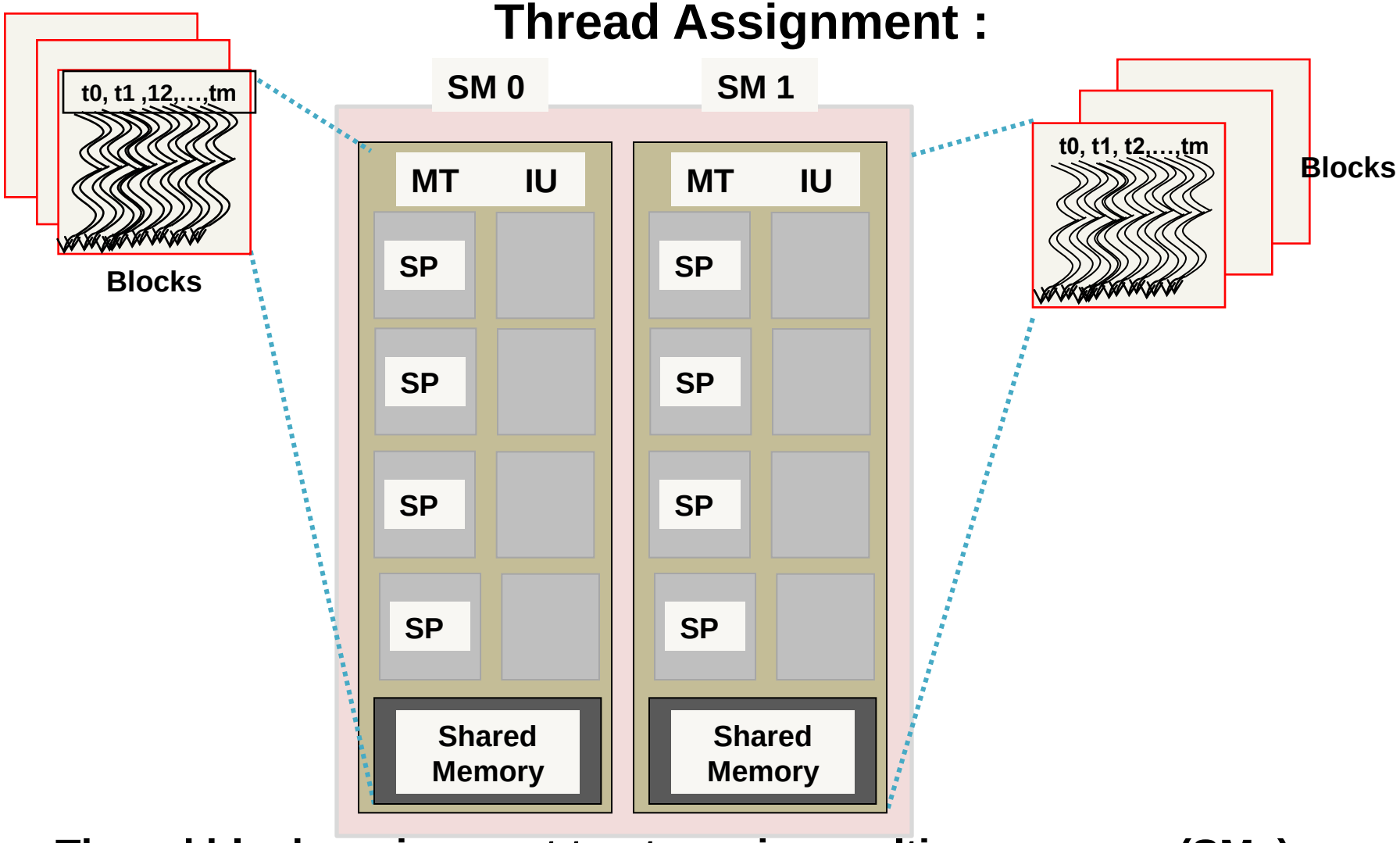
- These threads are assigned to execution resources on a block-by-block basis.

- Thread block assignment to streaming multiprocessors (SMs)

Source & Acknowledgements : NVIDIA, References

NVIDIA : CUDA Threads Organisation

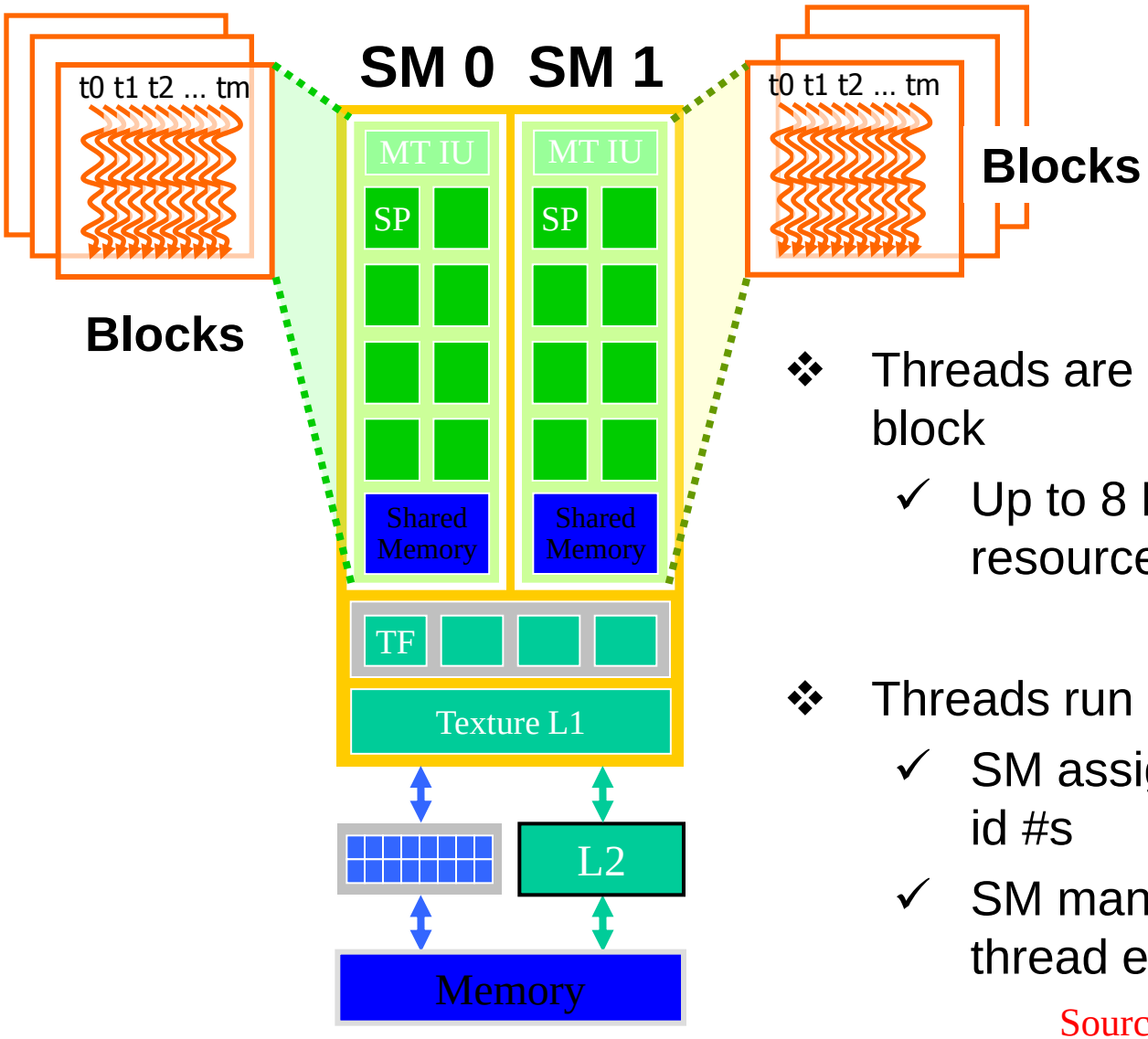
Thread Assignment :



Thread block assignment to streaming multiprocessors (SMs)

Source & Acknowledgements : NVIDIA, References

NVIDIA : CUDA Threads Organisation

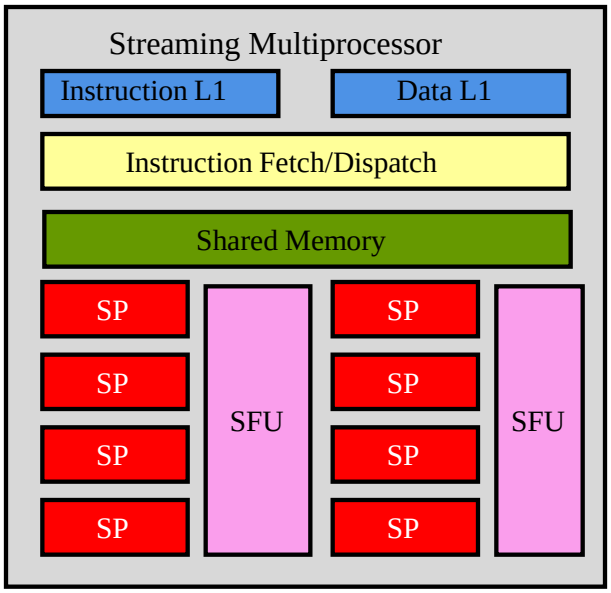


- Threads are assigned to SMs in block
  - ✓ Up to 8 Blocks to each SM as resource allows
- Threads run concurrently
  - ✓ SM assigns/maintains thread id #s
  - ✓ SM manages/schedules thread execution

Source : NVIDIA, References

Streaming Multiprocessor (SM)

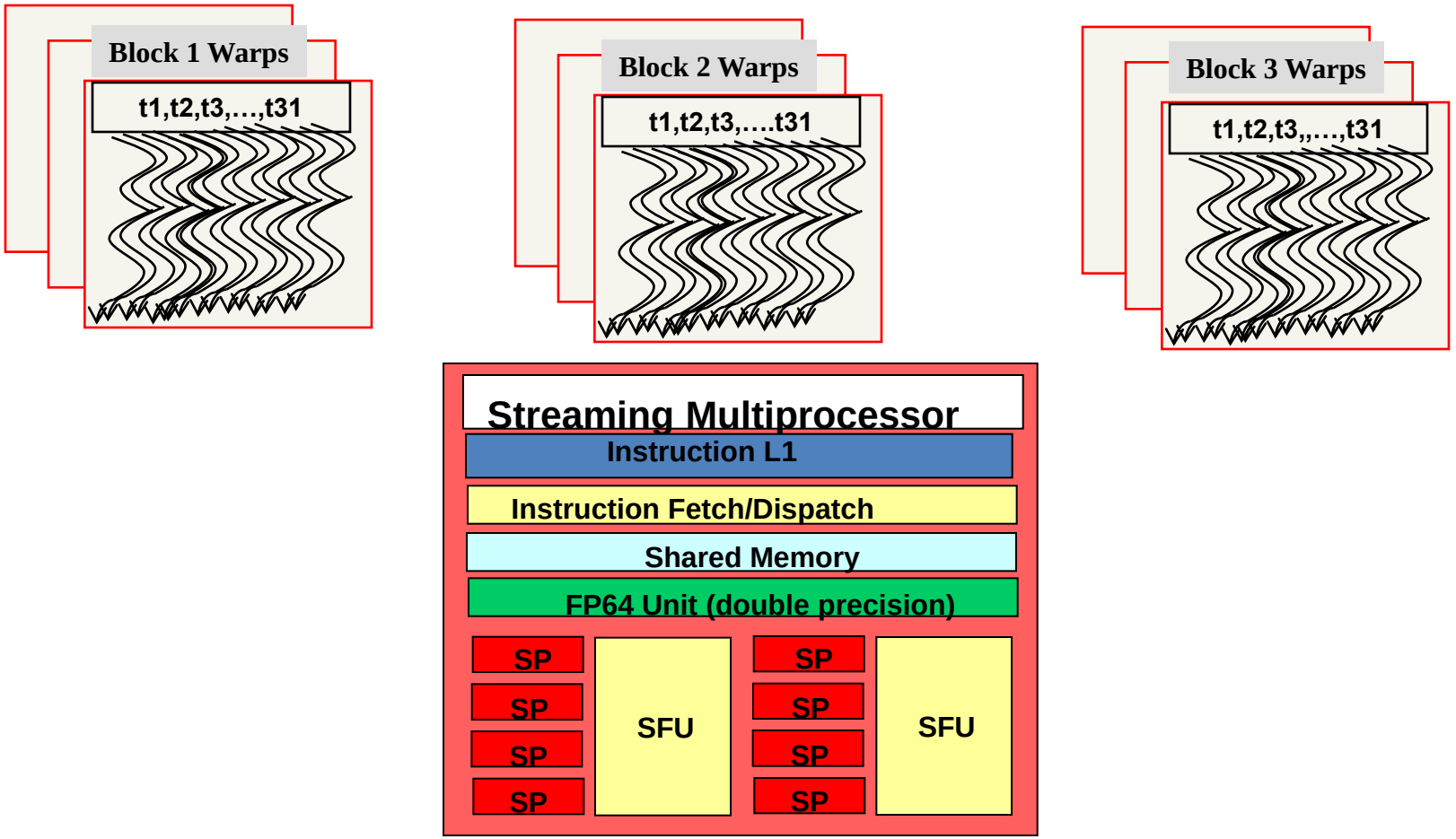
- Streaming Multiprocessor (SM)
  - ✓ 8 Streaming Processors (SP)
  - ✓ 2 Super Function Units (SFU)
- Multi-threaded instruction dispatch
  - ✓ 1 to 512 threads active
  - ✓ Shared instruction fetch per 32 threads
  - ✓ Cover latency of texture/memory loads
- 20+ GFLOPS (24 GFLOPS in G92)
- 16 KB shared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

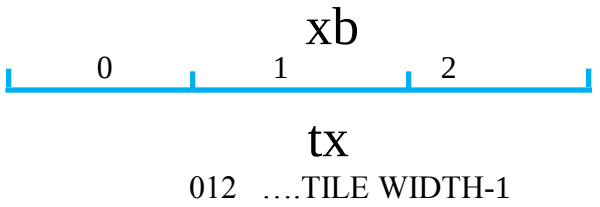
KERNEL FUNCTIONS AND THREADING

❖ **`threadIdx.x & threadIdx.y`**

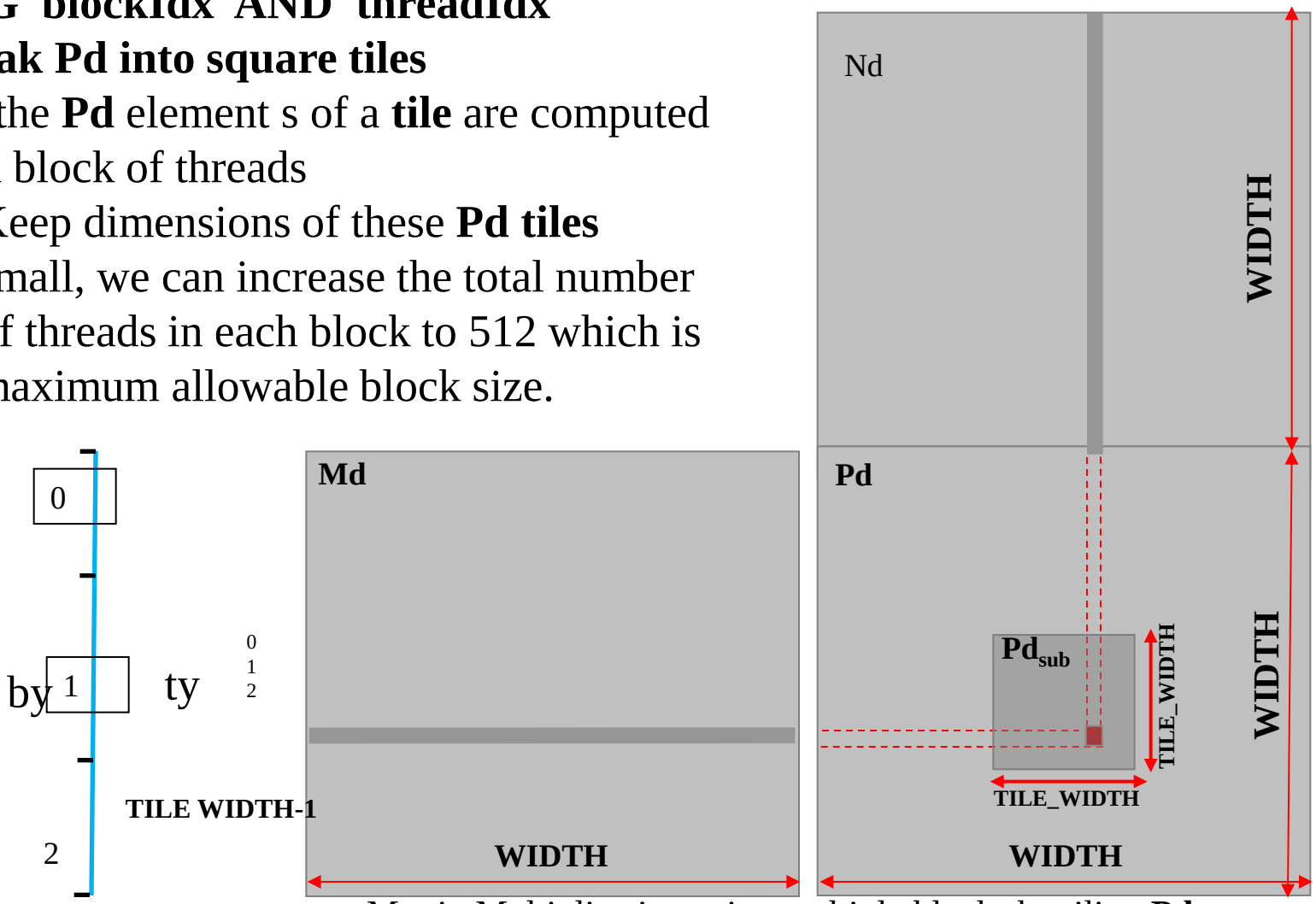
- Refer to the thread indices of a thread (Different threads will see different values in their **`threadIdx.x`** and t**`hreadIdx.y`** variables)
- Refer thread as **`Thread`***threadIdx.x, threadIdx,y* Coordinates reflect a multi-dimensional organization for the threads.
- CUDA threading hardware generates 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
```

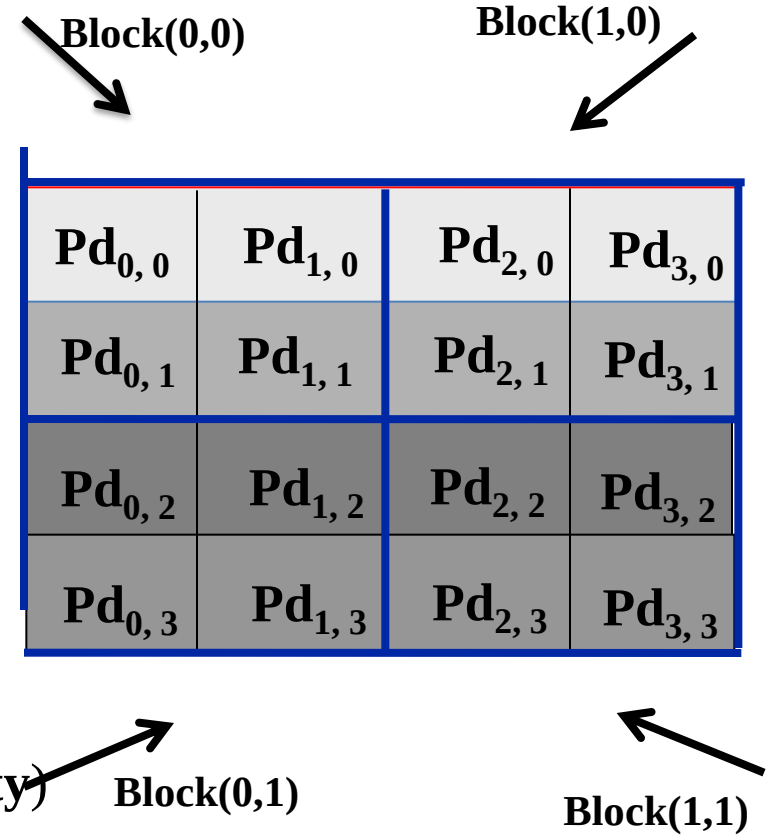
`Pd` element at respective column & row can be computed.

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**
- 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 Multiple blocks to calculate **Pd**.

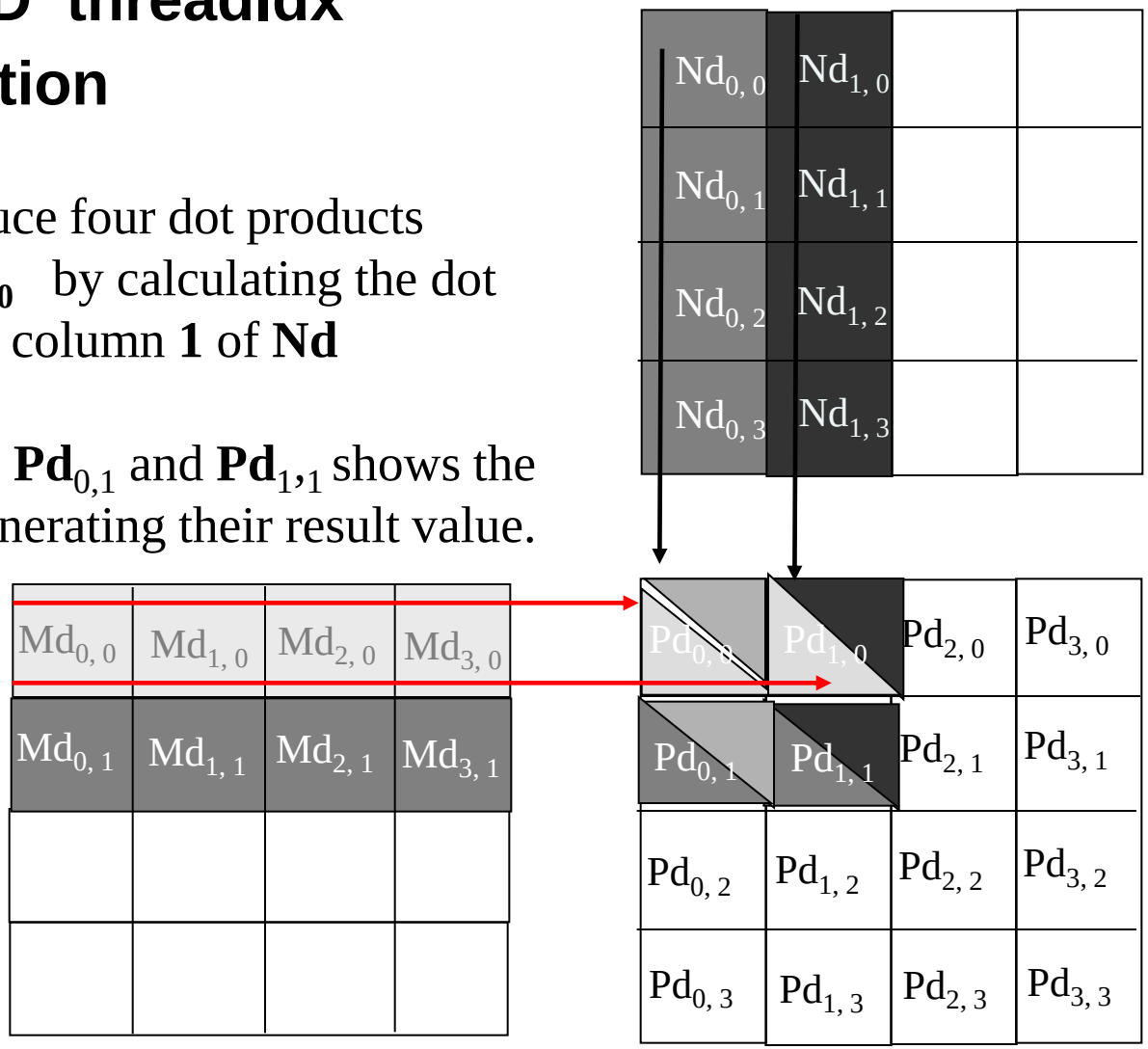
NVIDIA :CUDA Thread Organisation

USING blockIdx AND threadIdx

Ex : Matrix Multiplication

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

- The arrows of $\mathbf{Pd_{0,0}}$, $\mathbf{Pd_{1,0}}$, $\mathbf{Pd_{0,1}}$ and $\mathbf{Pd_{1,1}}$ shows the row and column used for generating their result value.



Matrix multiplication actions of one thread block

NVIDIA :CUDA Thread Organisation

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);
    …
}
```

NVIDIA :CUDA Thread Organisation

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

NVIDIA :CUDA Thread Organisation

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.

NVIDIA : CUDA Thread Organisation

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);
```

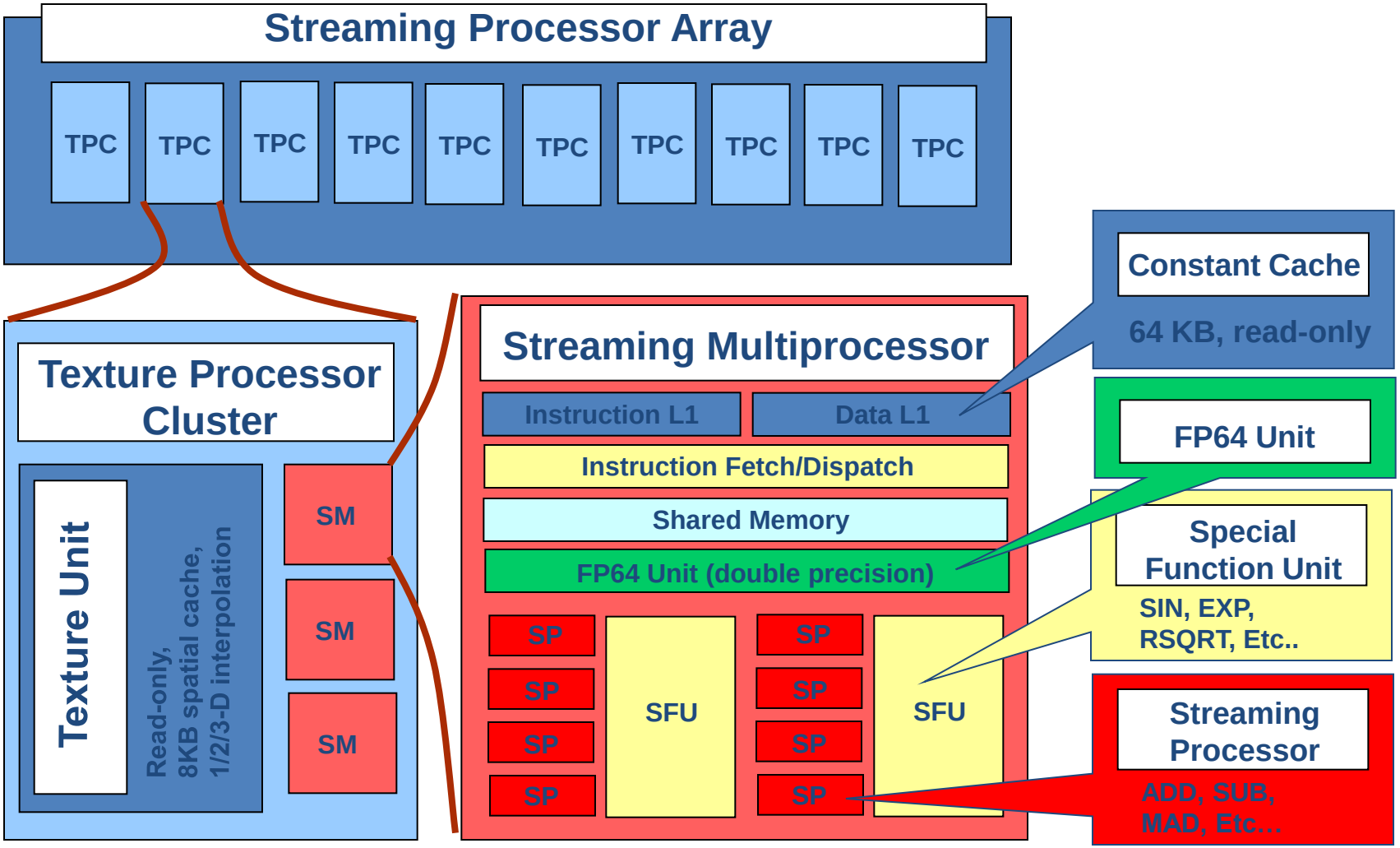
Note : dimGrid receives the value of **Width/TILE_WIDTH** for both the **x** dimension and **y** dimension.

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

NVIDIA : CUDA Thread Scheduling & Latency Tolerance

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**

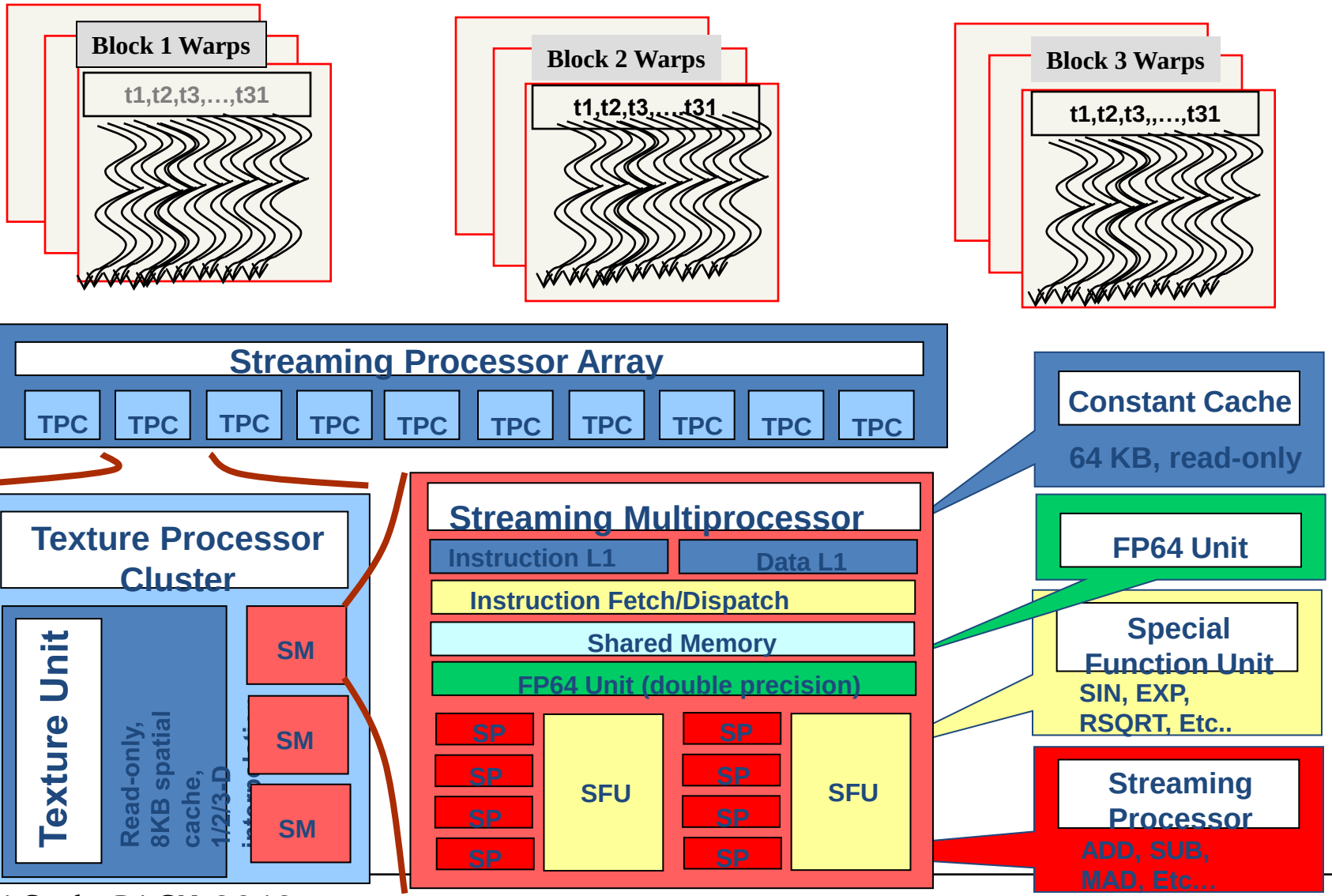
Pros & Cons of choice of "different thread blocks" for the GT200

- **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

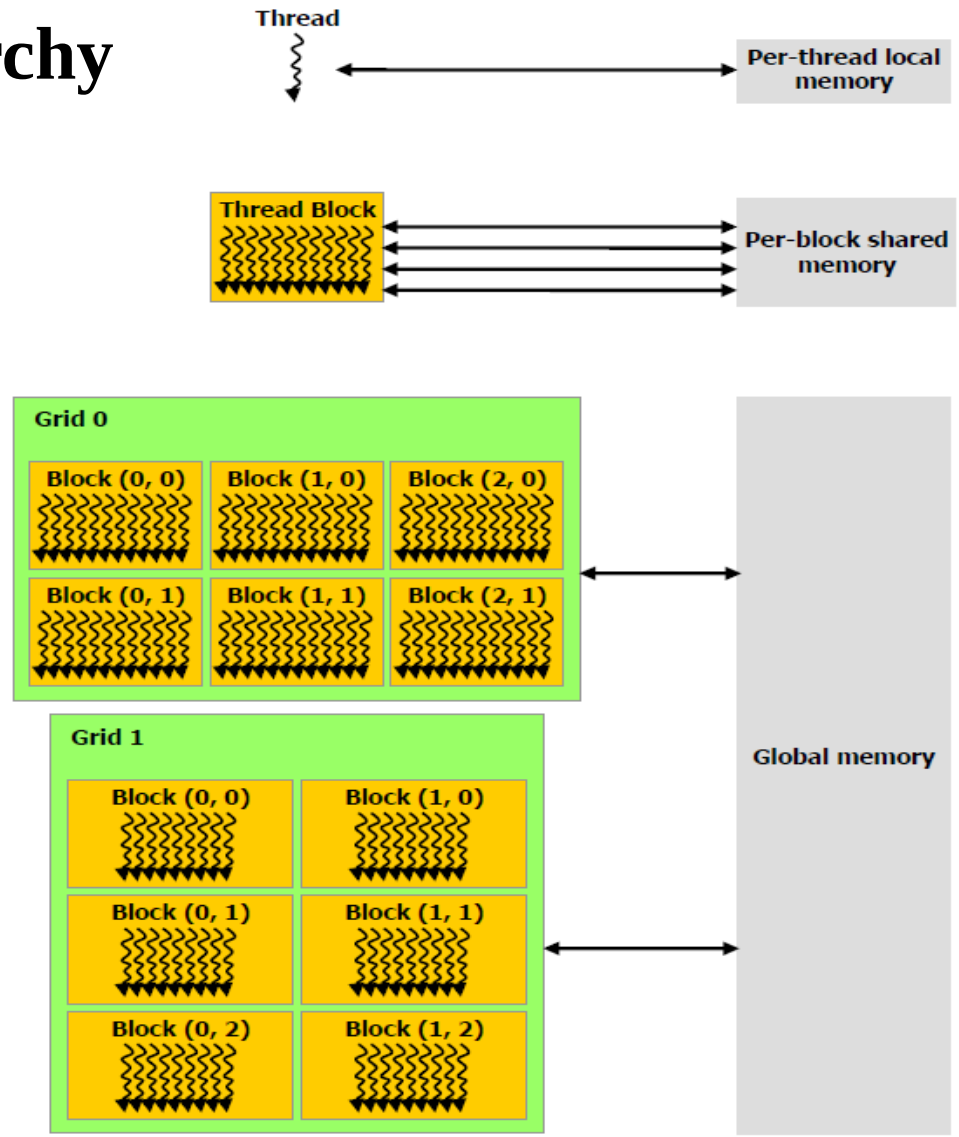
Part-4

CUDA Memories

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

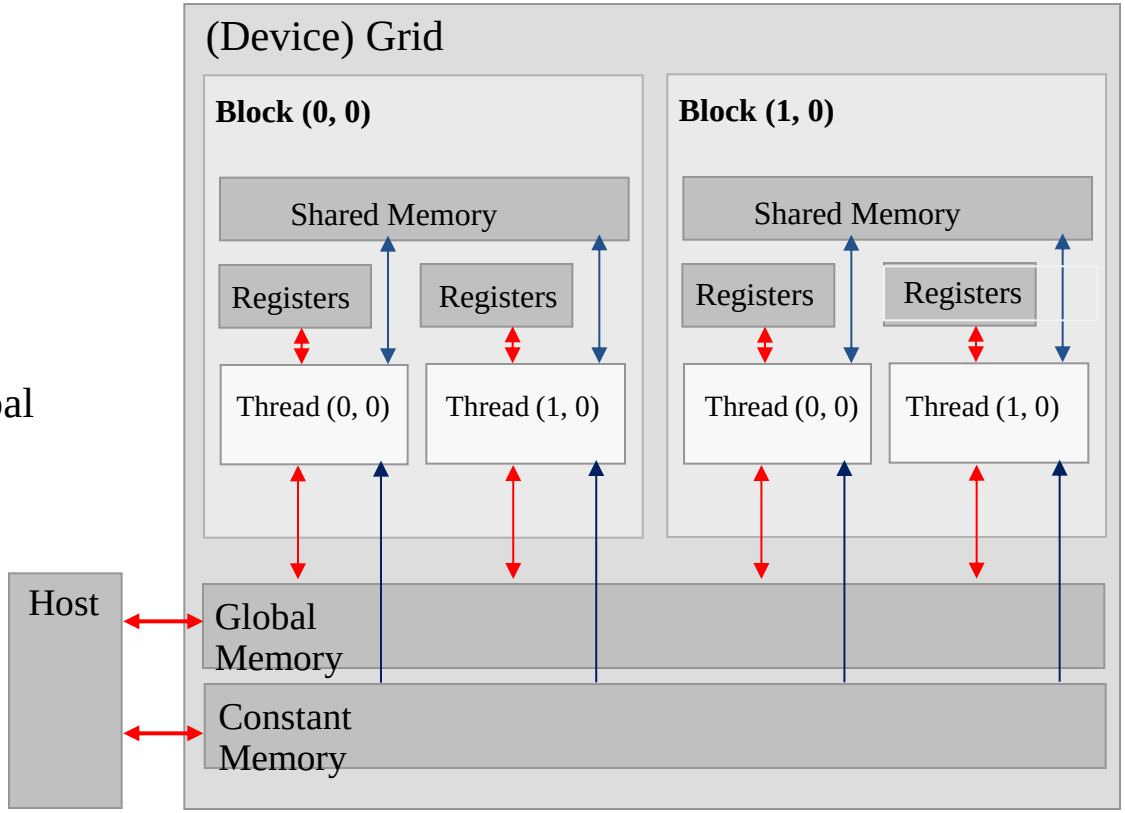
NVIDA :CUDA DEVICE MEMORIES & DATA TRANSFER

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 : Importance of Memory Access Efficiency

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 : Importance of Memory Access Efficiency

CUDA Device Memory Types - Shared Memory

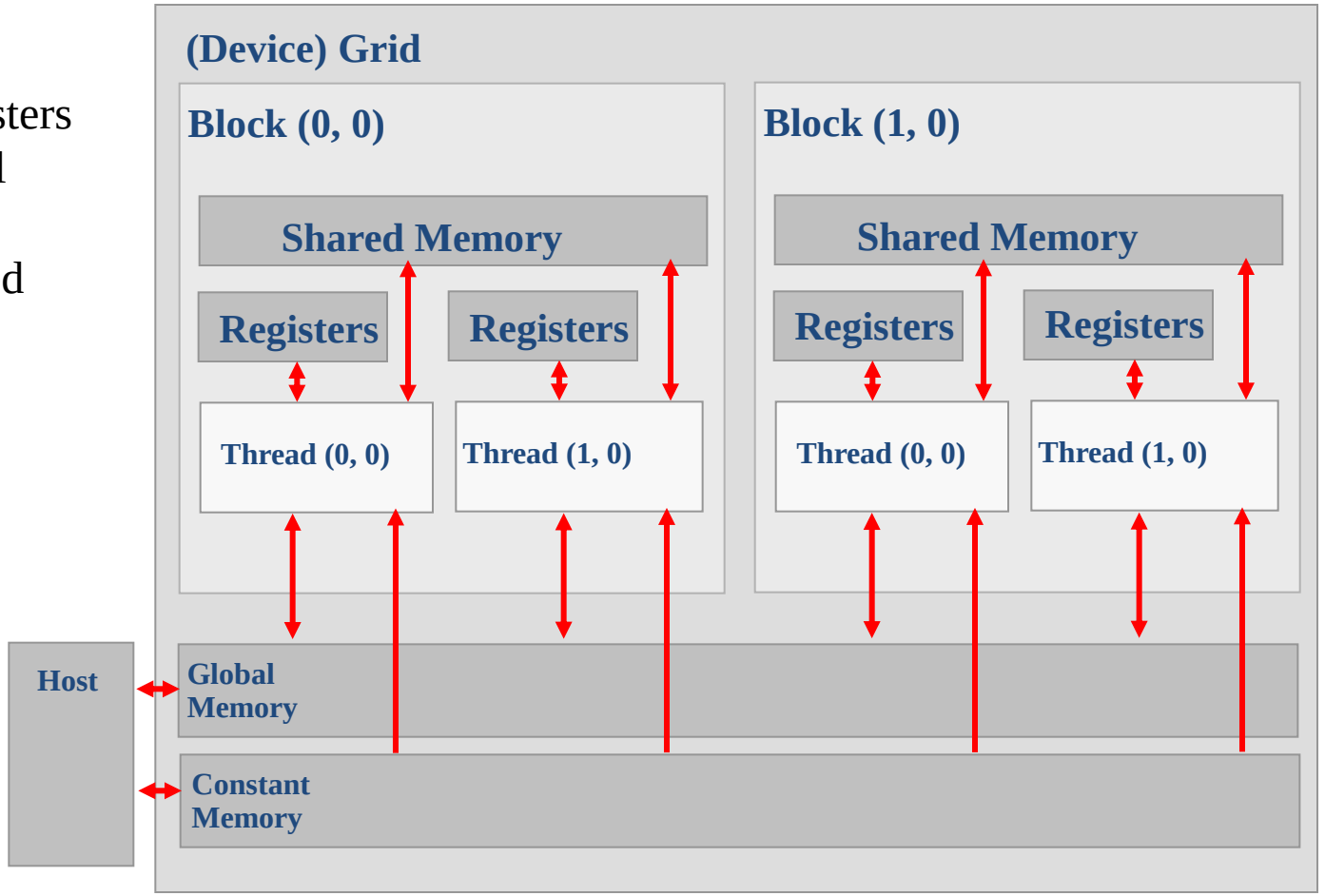
SCOPE :

- 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

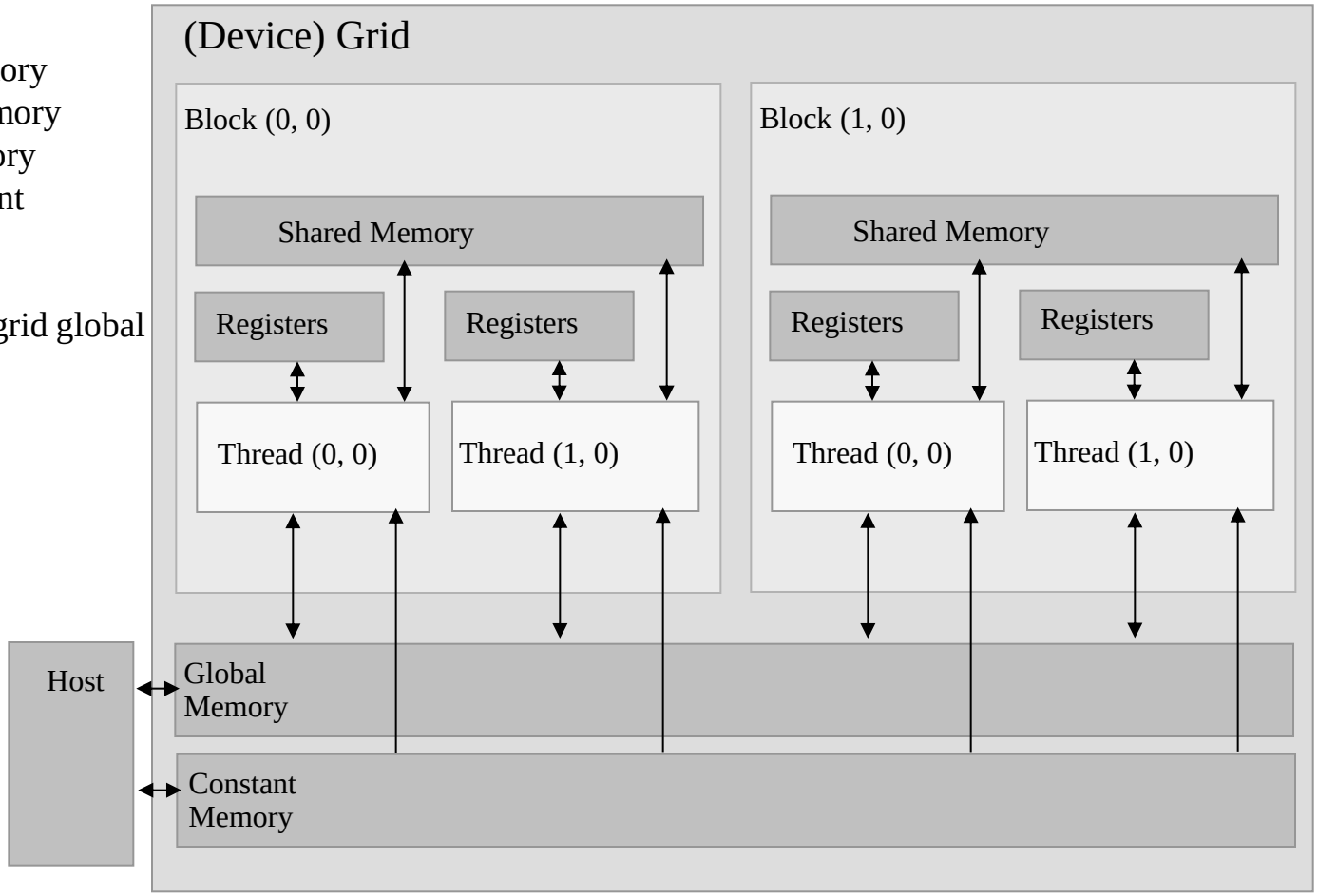


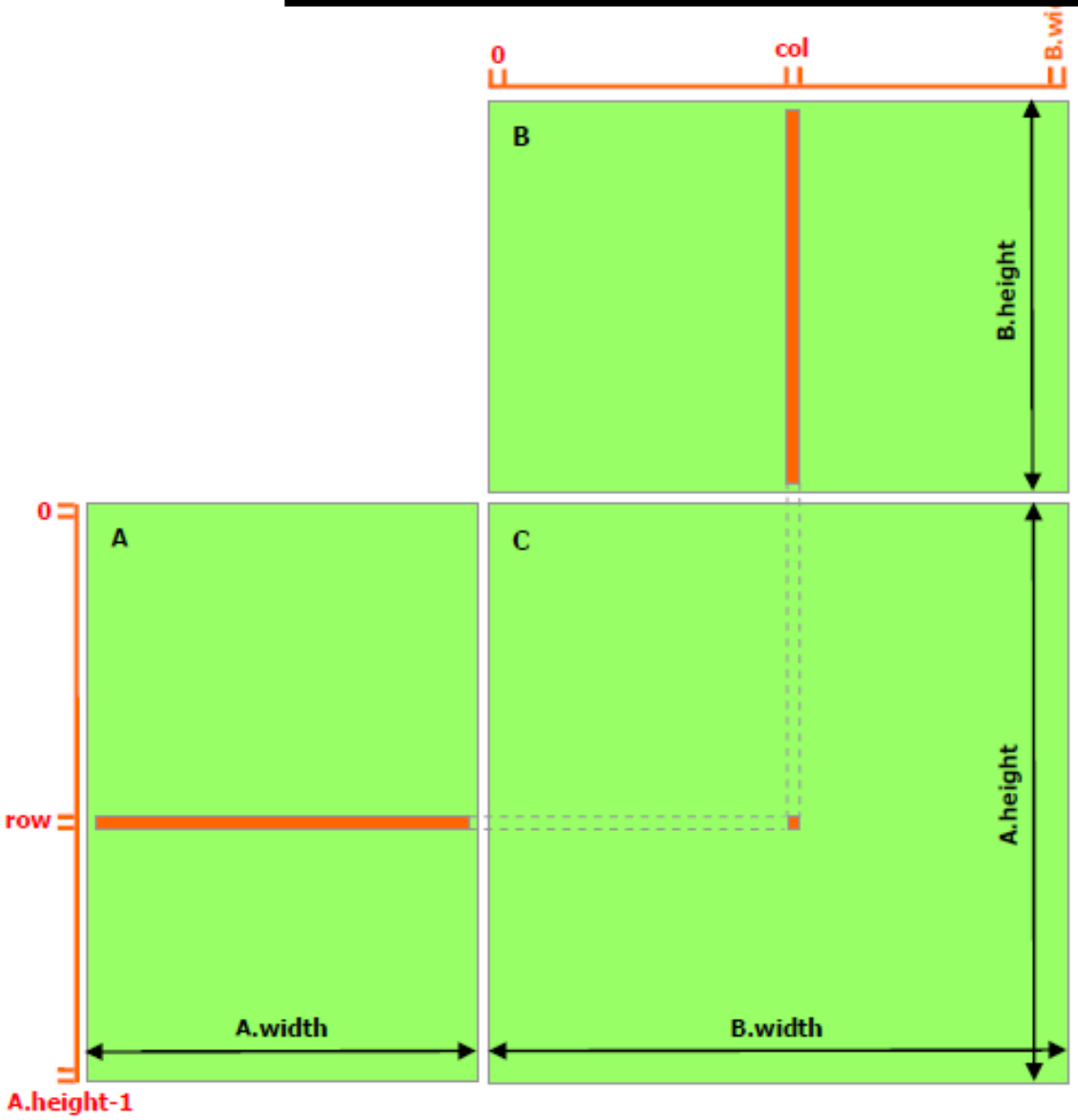
Figure 3.7 Overview of the CUDA device memory model .

NVIDIA :CUDA Thread Organisation

Revised matrix multiplication kernel using multiple blocks

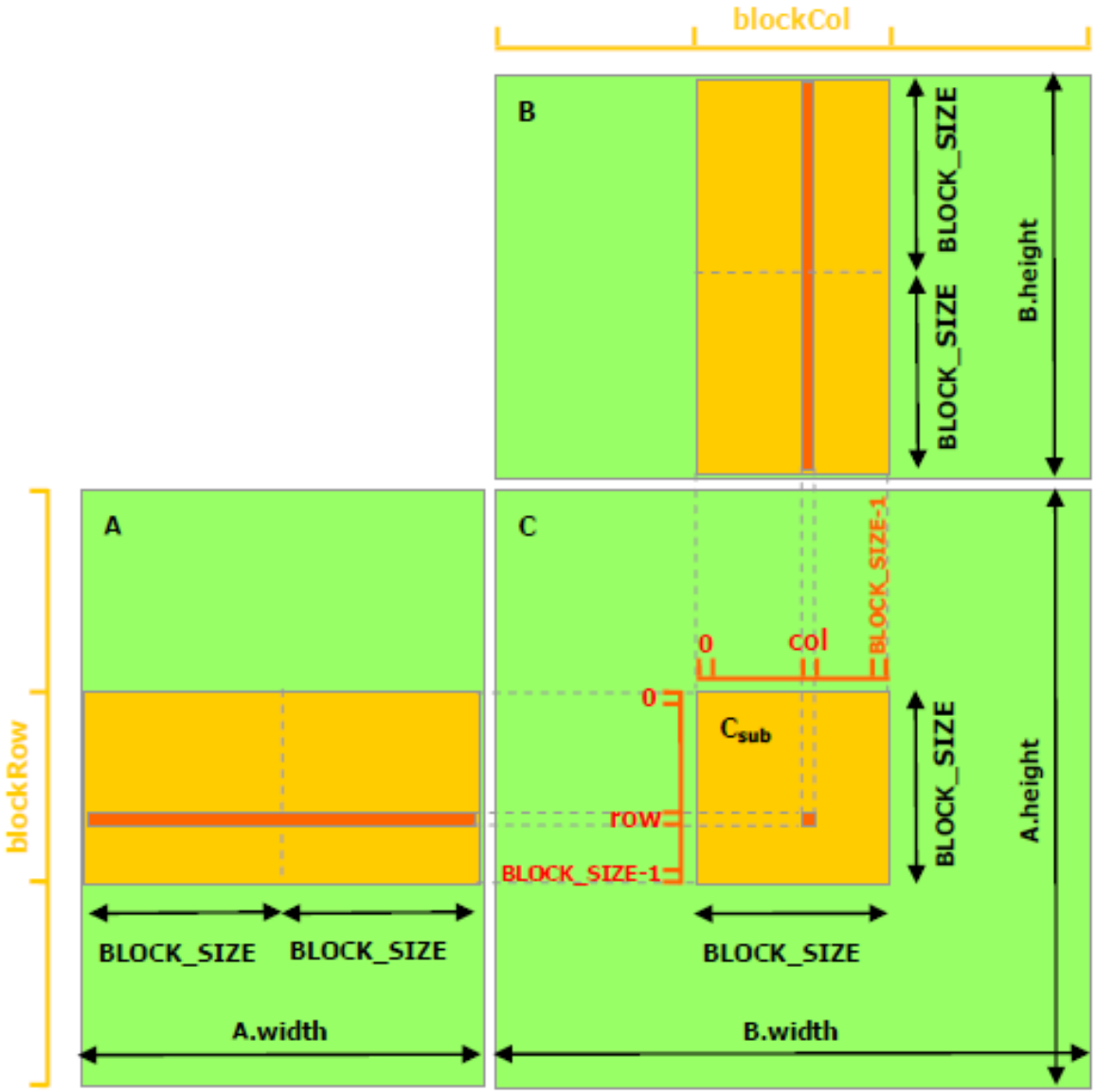
```
__global__ void MatrixMulKernel(float* Md, float* Nd,float* Pd,
int Width)
{
    // Calculate the row index of the Pd element and M
    int Row  = blockIdx.y *TILE_WIDTH + threadIdx.y;

    // 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

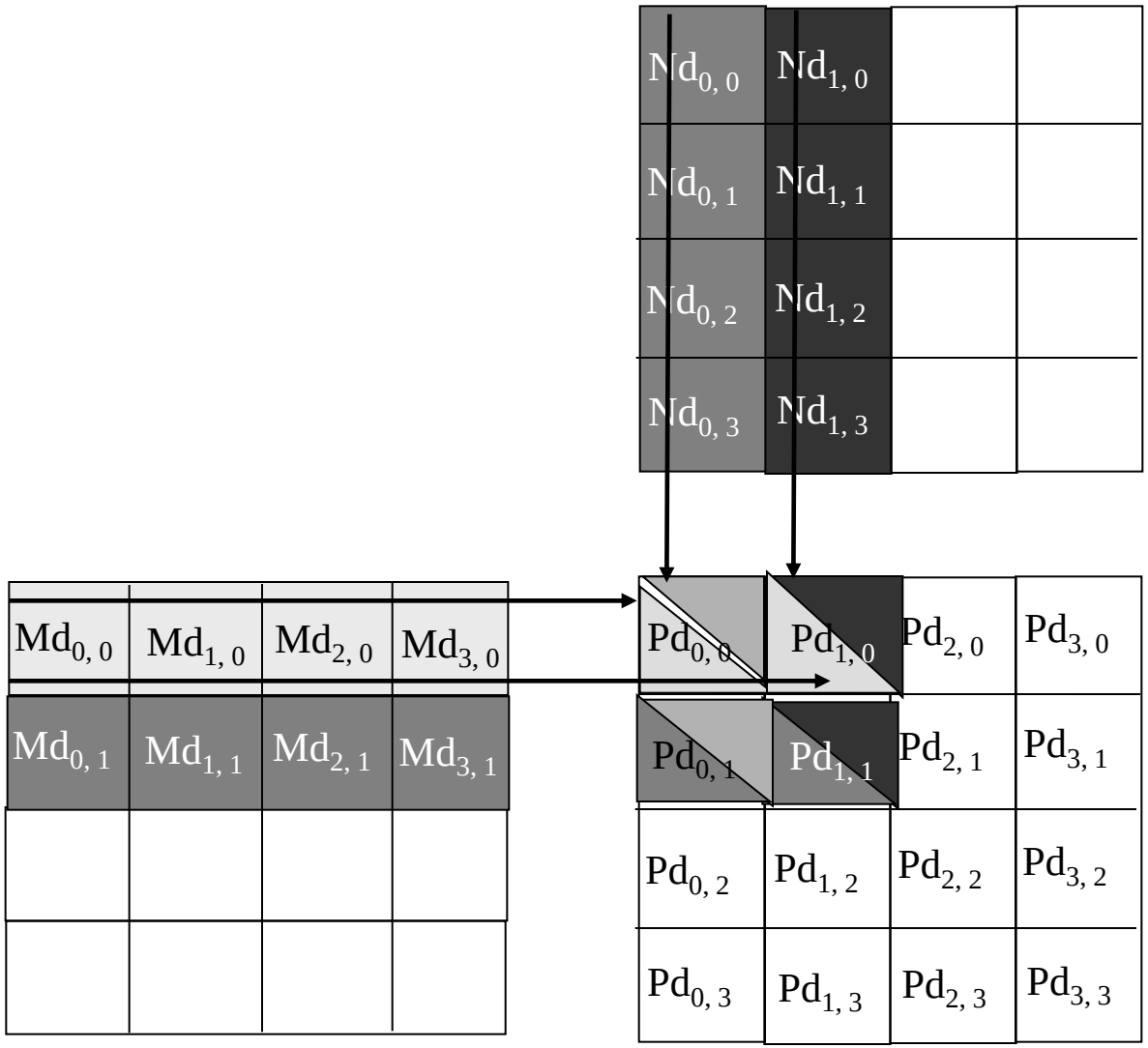


Figure Matrix multiplication actions of one thread block.

Part-5

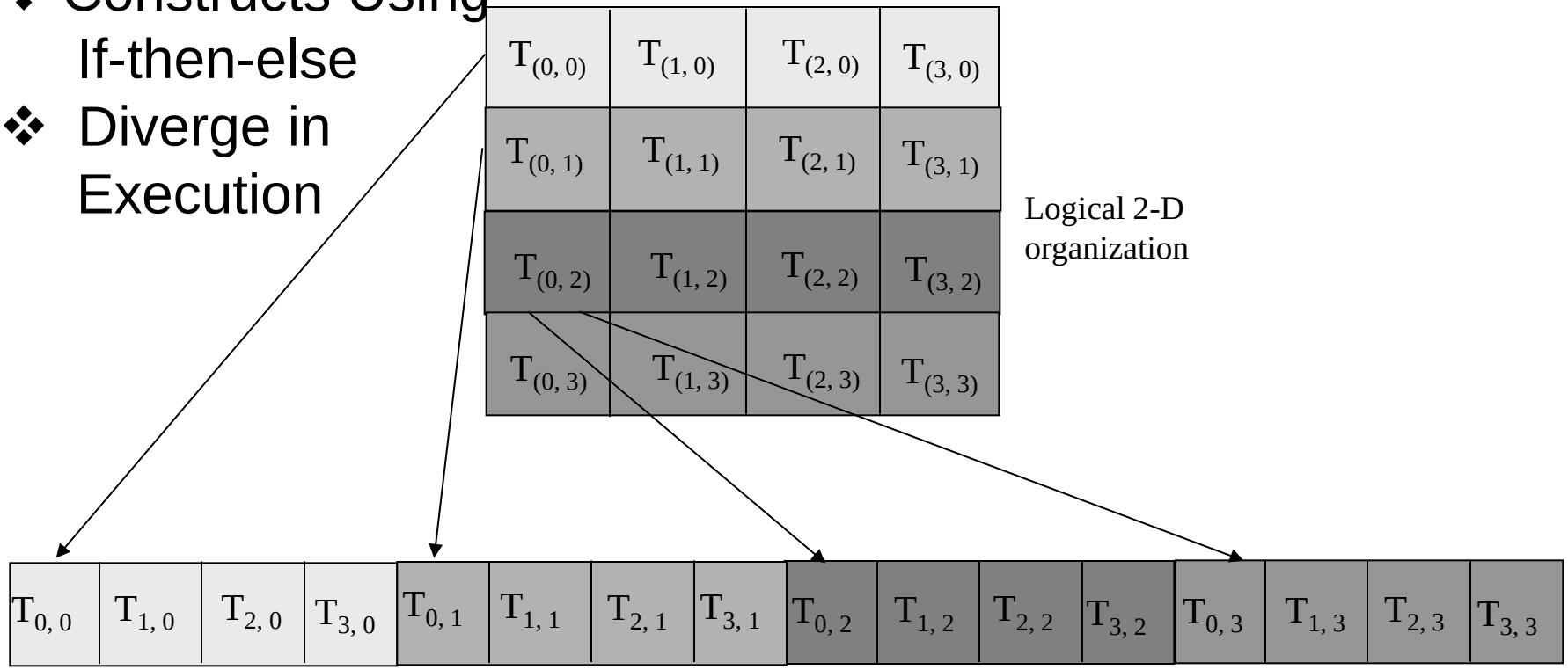
CUDA Execution

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

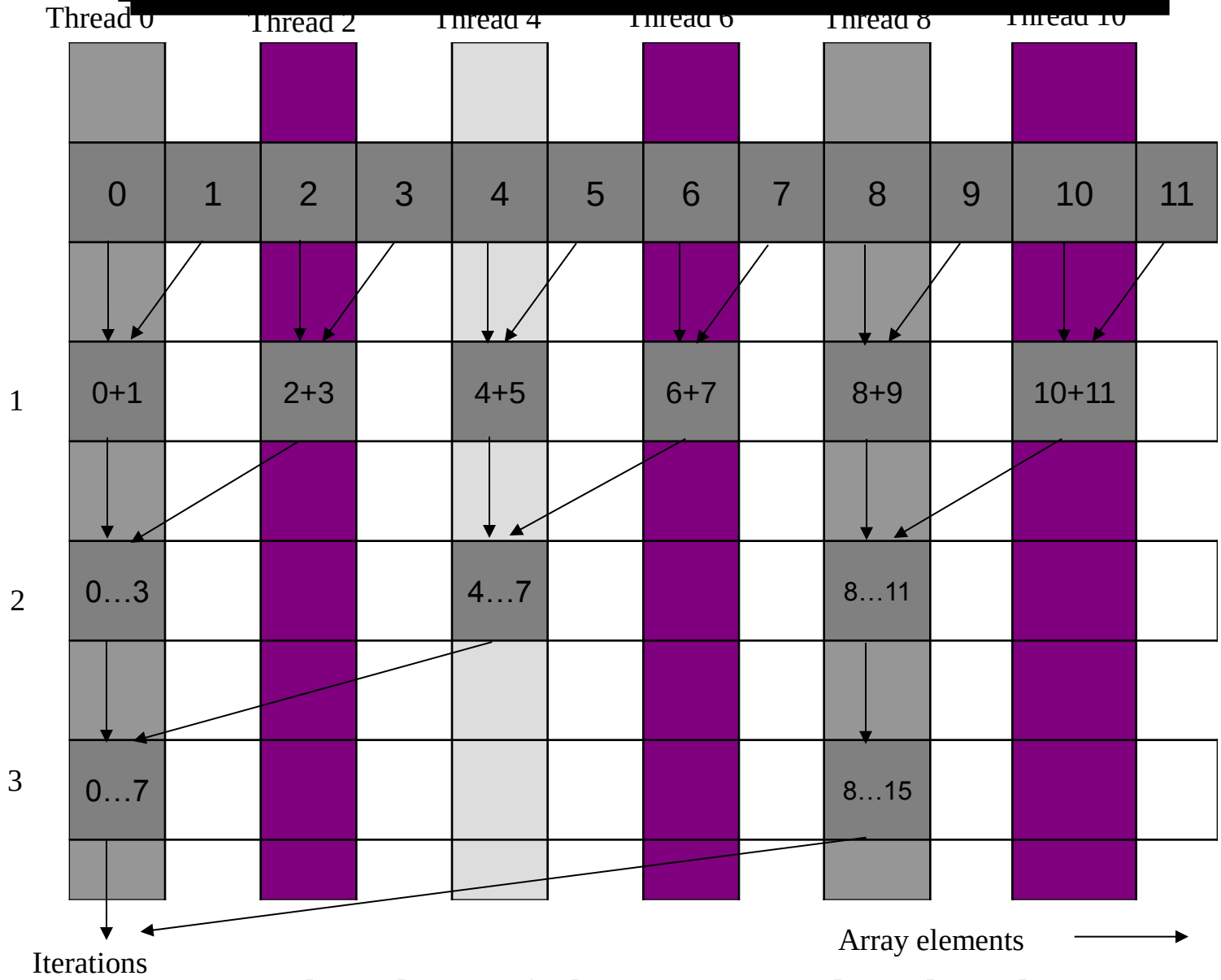
CUDA Thread Execution - Performance

```
1. _shared_float partialSum[ ]

2. Unsingned int t = threadsIdx.x;

3. for (unsigned int stride = 1;
4.          stride < blockDim.X; stride *=2)
5.  {
6.    __syncthreads ( );
7.    If (t % (2*stride) == 0)
8.   partialSum[t] + = partialSum[ t +stride];
9. }
```

A simple sum reduction kernel.

Source & Acknowledgements : NVIDIA, References

CUDA Thread Execution - Performance



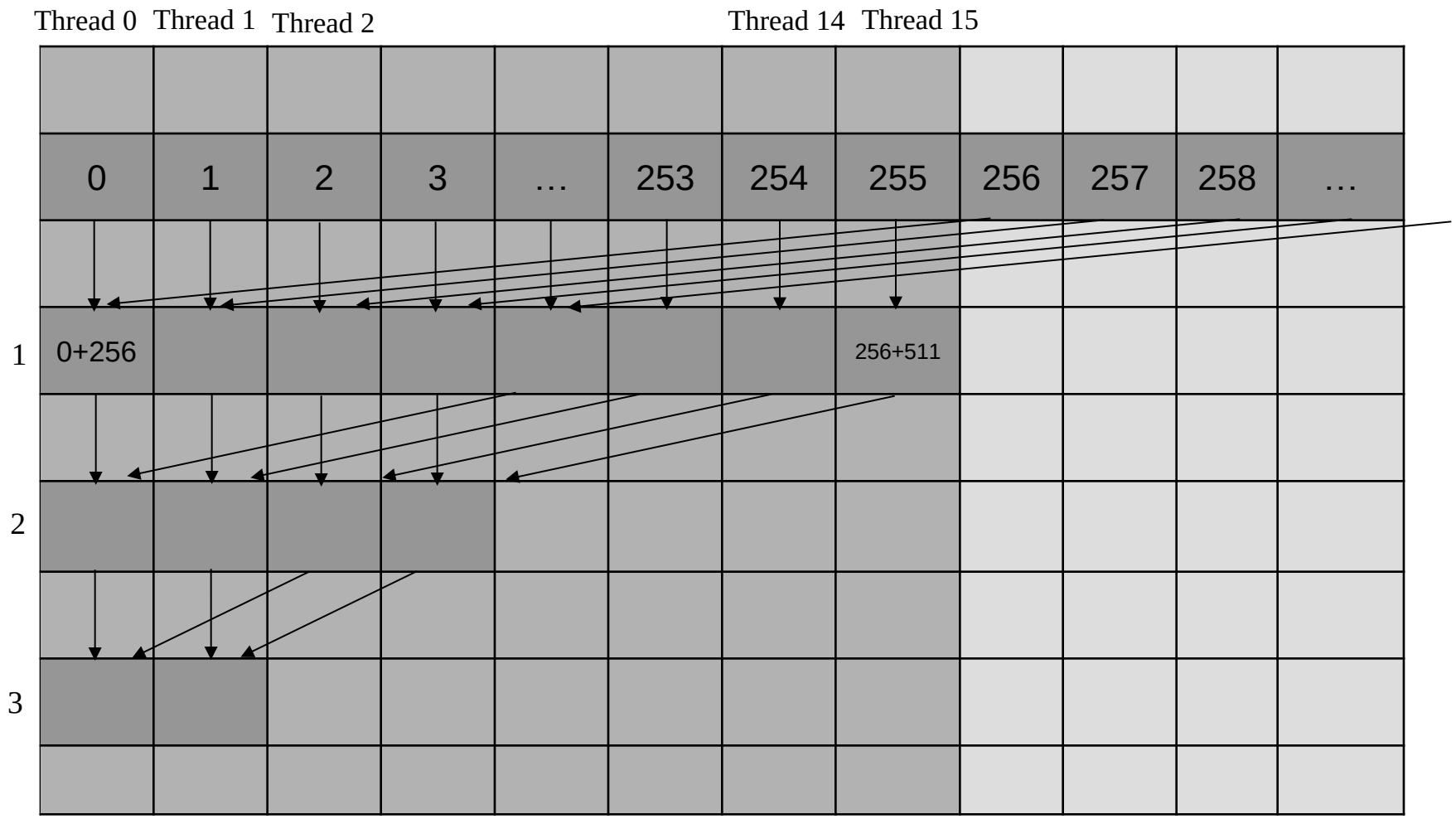
A Deduction of the sum reduction kernel.

CUDA Thread Execution - Performance

```
1. _shared_float partialSum[ ]

2. Unsingned int t = threadsIdx.x;

3. for (unsigned int stride = 1;
4.         stride < blockDim.X; stride *=2)
5. {
6.     __syncthreads ( );
7.     If (t  < stride)
8.     partialSum[t] + = partialSum[ t +stride];
9. }
```

A kernel with les thread divergence.

CUDA Thread Execution - Performance



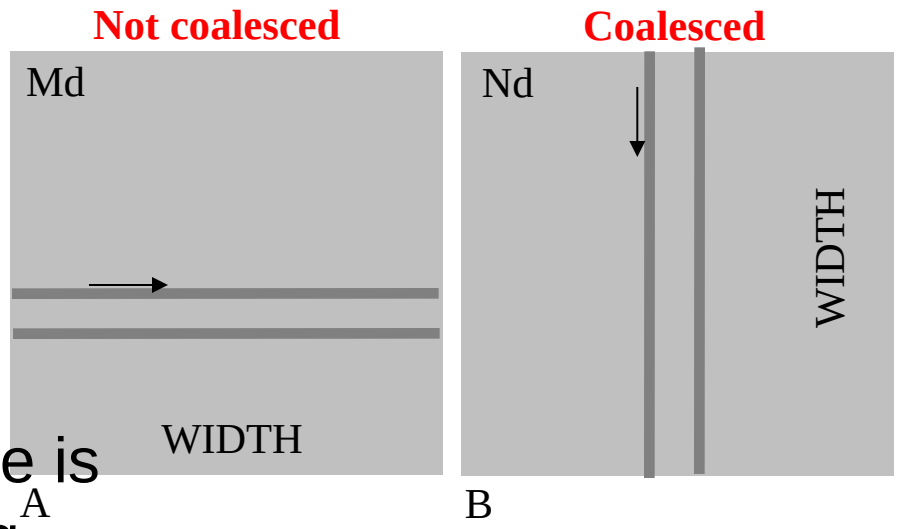
Execution of the revised algorithm.

CUDA Thread Execution - Performance

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.

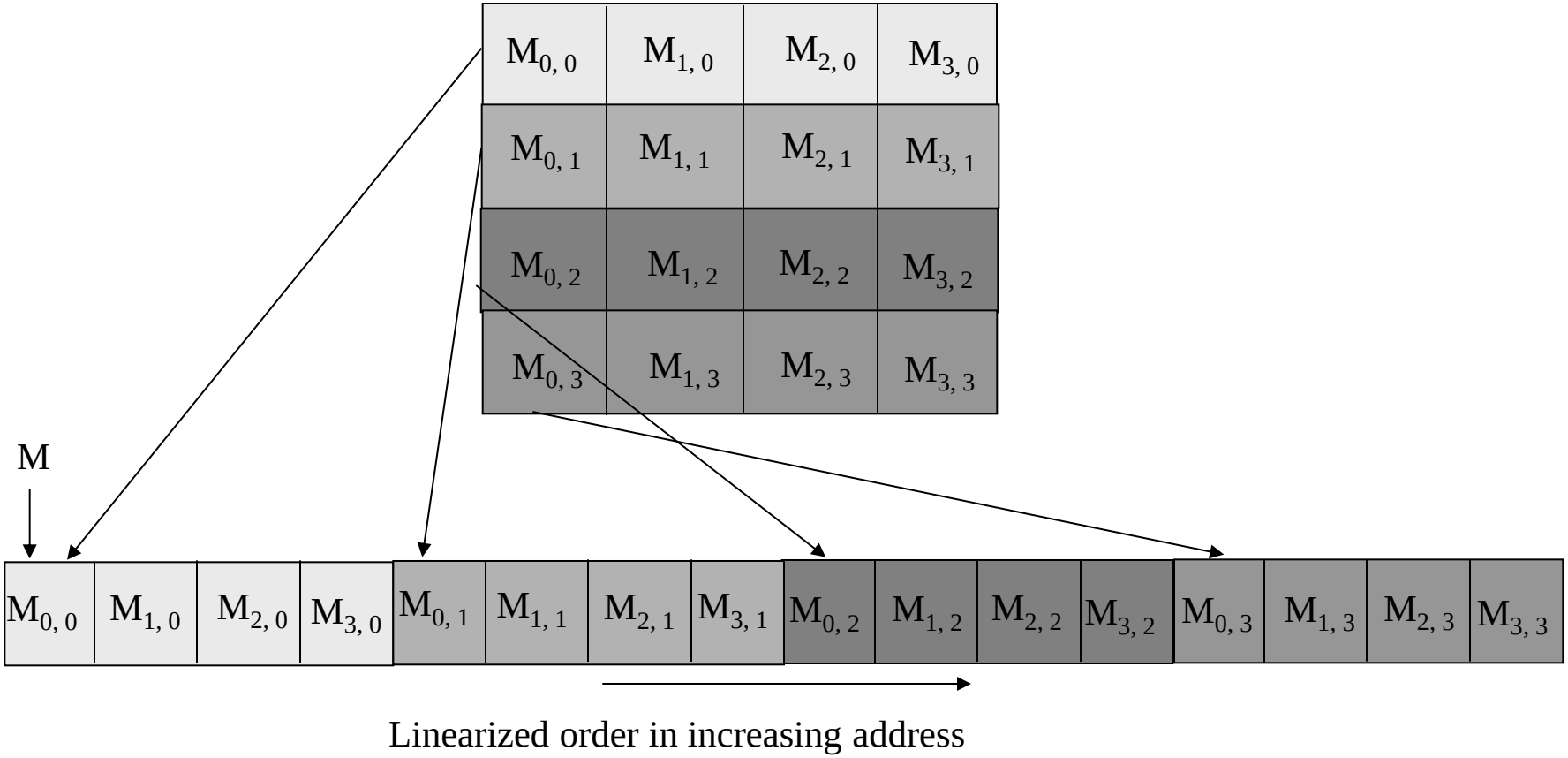


- Memory Coalescing technique is used in conjunction with tiling technique

Memory access pattern for coalescing.

CUDA Thread Execution - Performance

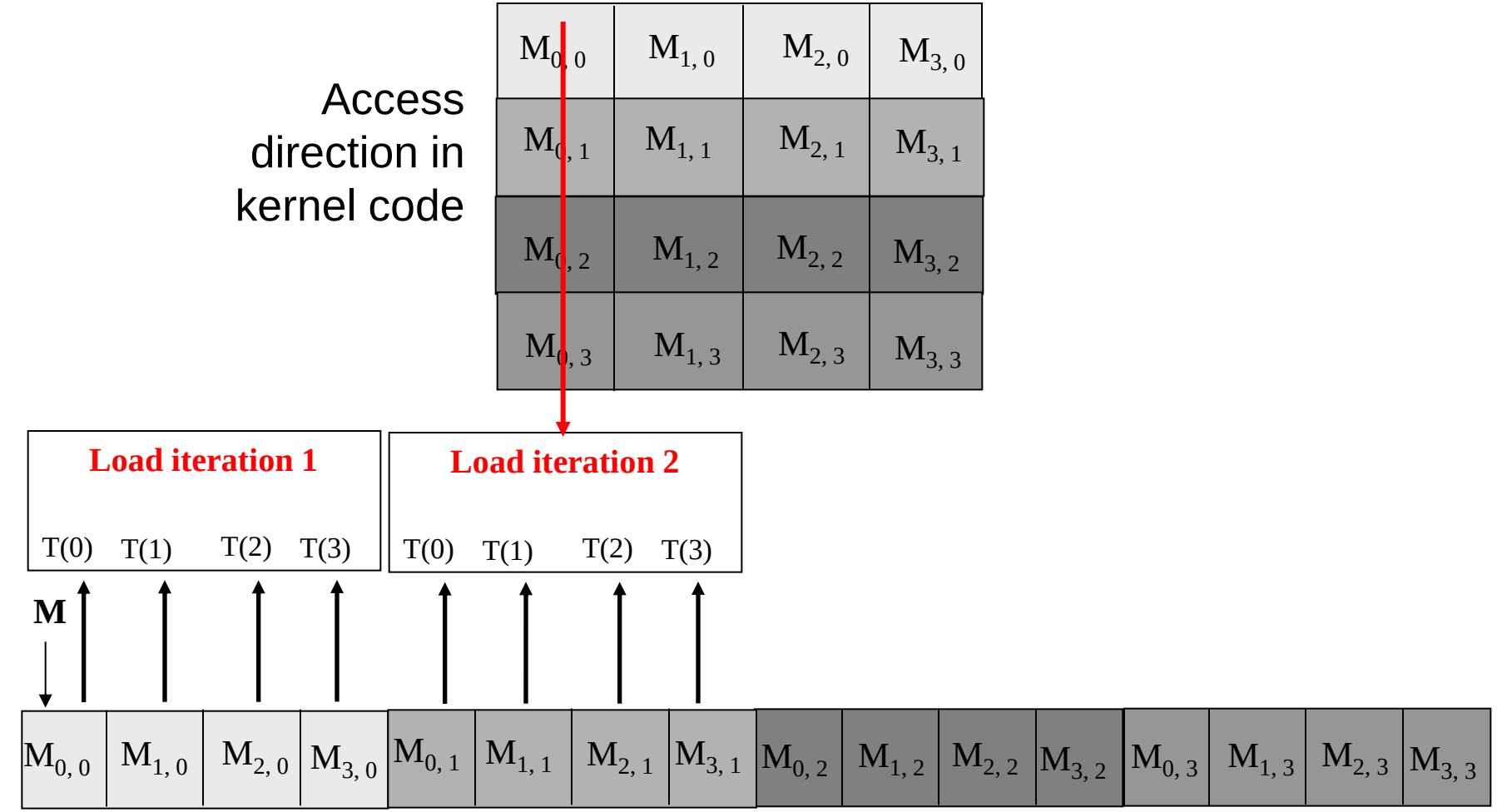
Global Memory Bandwidth



Placing matrix elements order into linear order.

CUDA Thread Execution - Performance

Global Memory Bandwidth

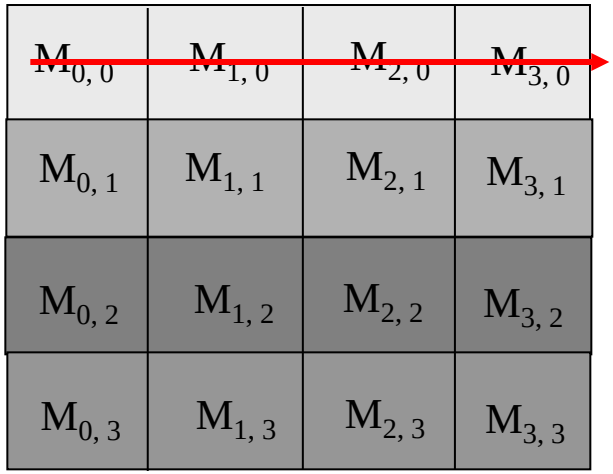


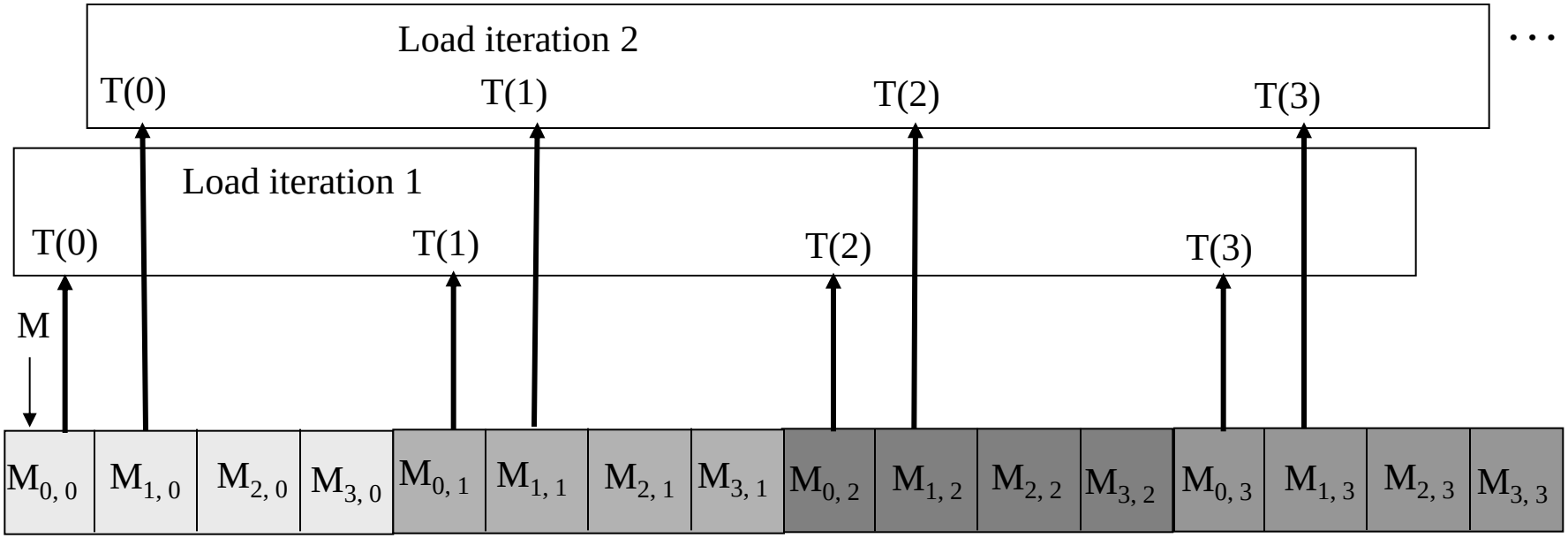
A coalesced access pattern.

CUDA Thread Execution - Performance

Global Memory Bandwidth

Access direction in kernel code

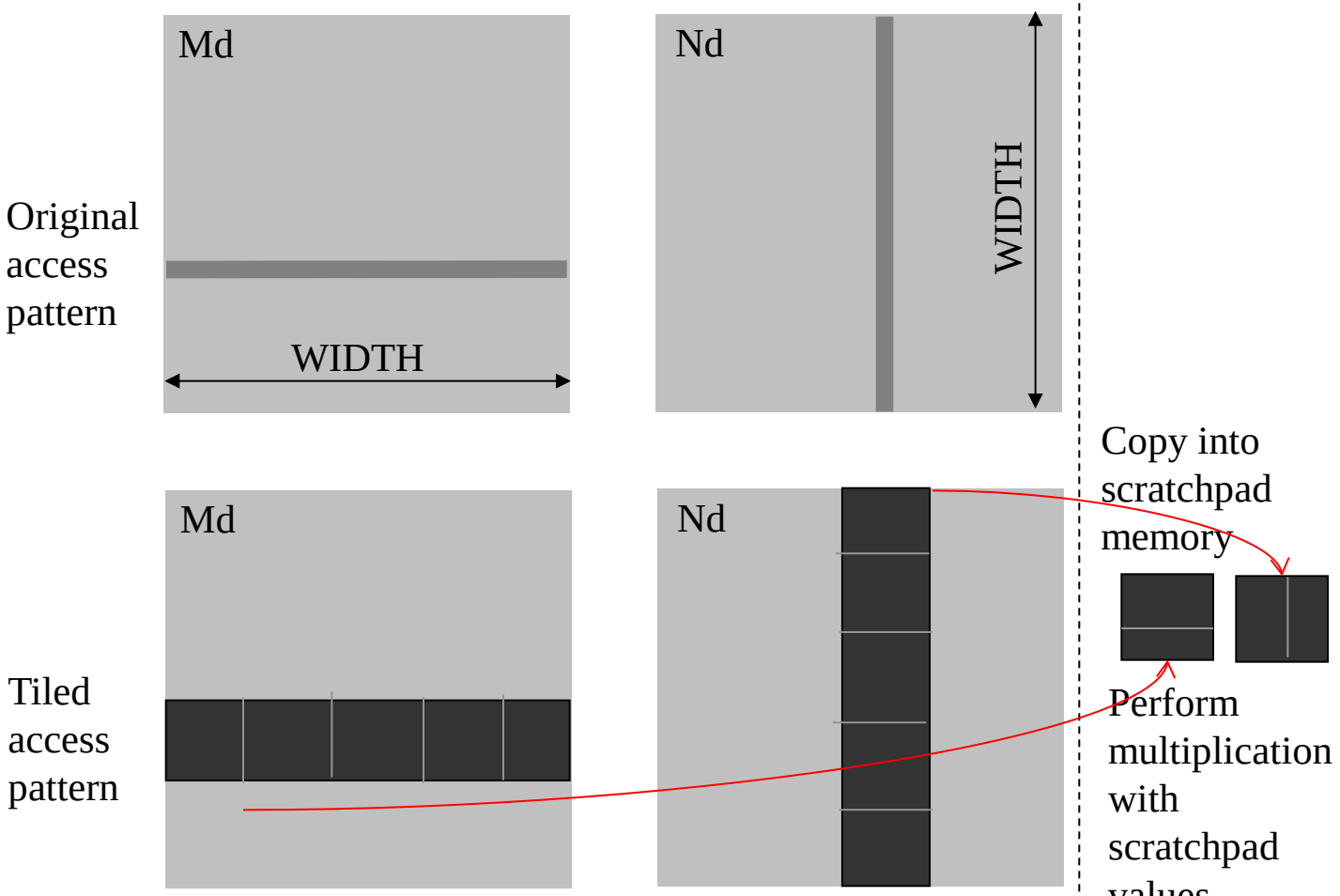




A uncoalesced access pattern.

CUDA Thread Execution - Performance

Global Memory Bandwidth



Using shared memory to enable coalescing.

CUDA Thread Execution - Performance

```
_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];
```

Md Nd

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

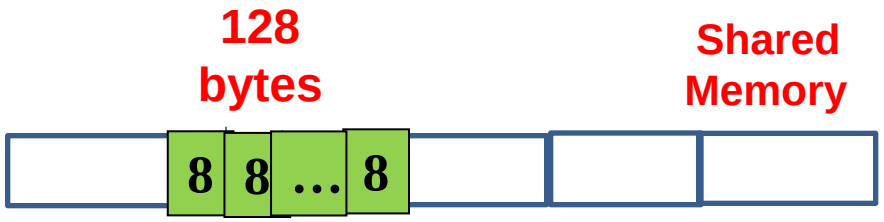
Global Memory Bandwidth

```
// 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;
     }
}
```

The matrix multiplication kernel using shared memories.

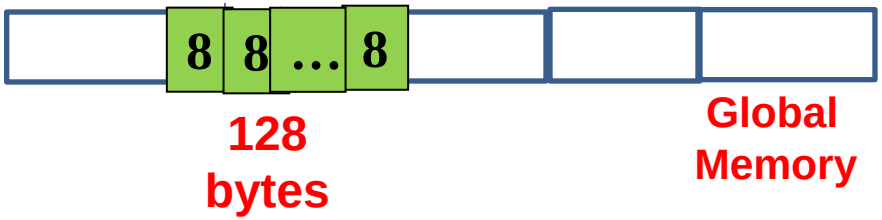
GPU performance : Memory Coalescing

- Request >16-bytes serviced iteratively



Reading 16-
bytes at a time

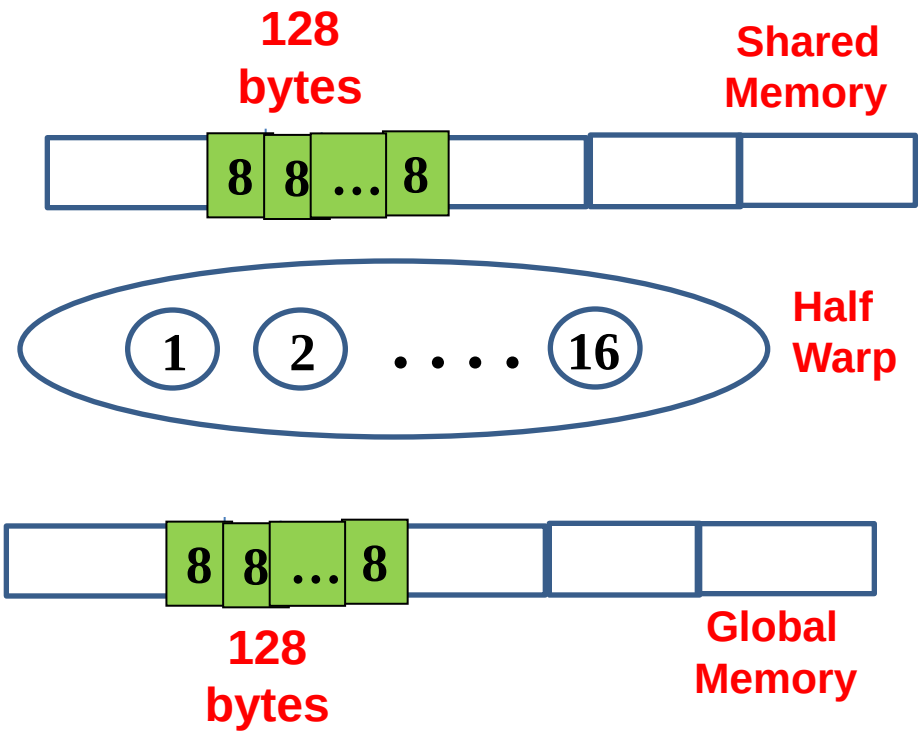
Thread



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

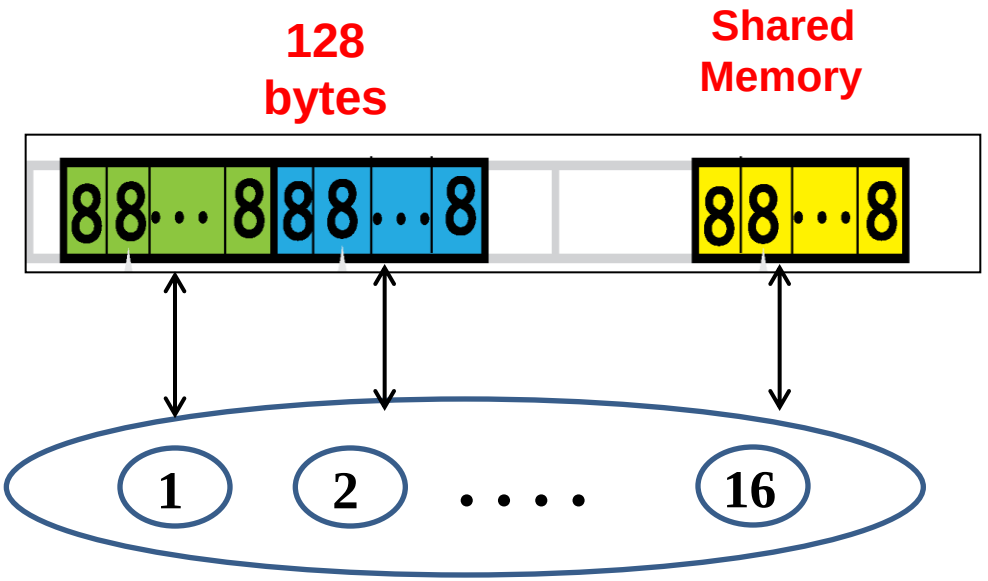


Source & Acknowledgements : NVIDIA, References

GPU performance : Memory Coalescing

Modify operation:

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



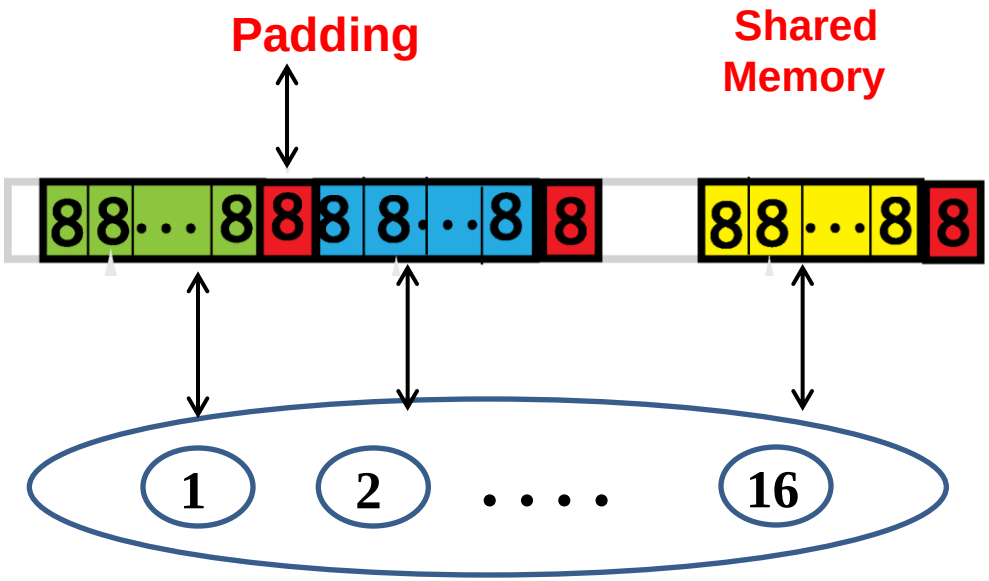
Threads of a Half-warp

Source & Acknowledgements : NVIDIA, References

GPU performance : Memory Coalescing

Modify operation:

❖Pad offset of 8 bytes,
Thereby reduce bank conflicts



Threads of a Half-warp

Source & Acknowledgements : NVIDIA, References

CUDA Thread Execution - Performance

Global Memory Bandwidth : Dynamic Partitioning of SM resources

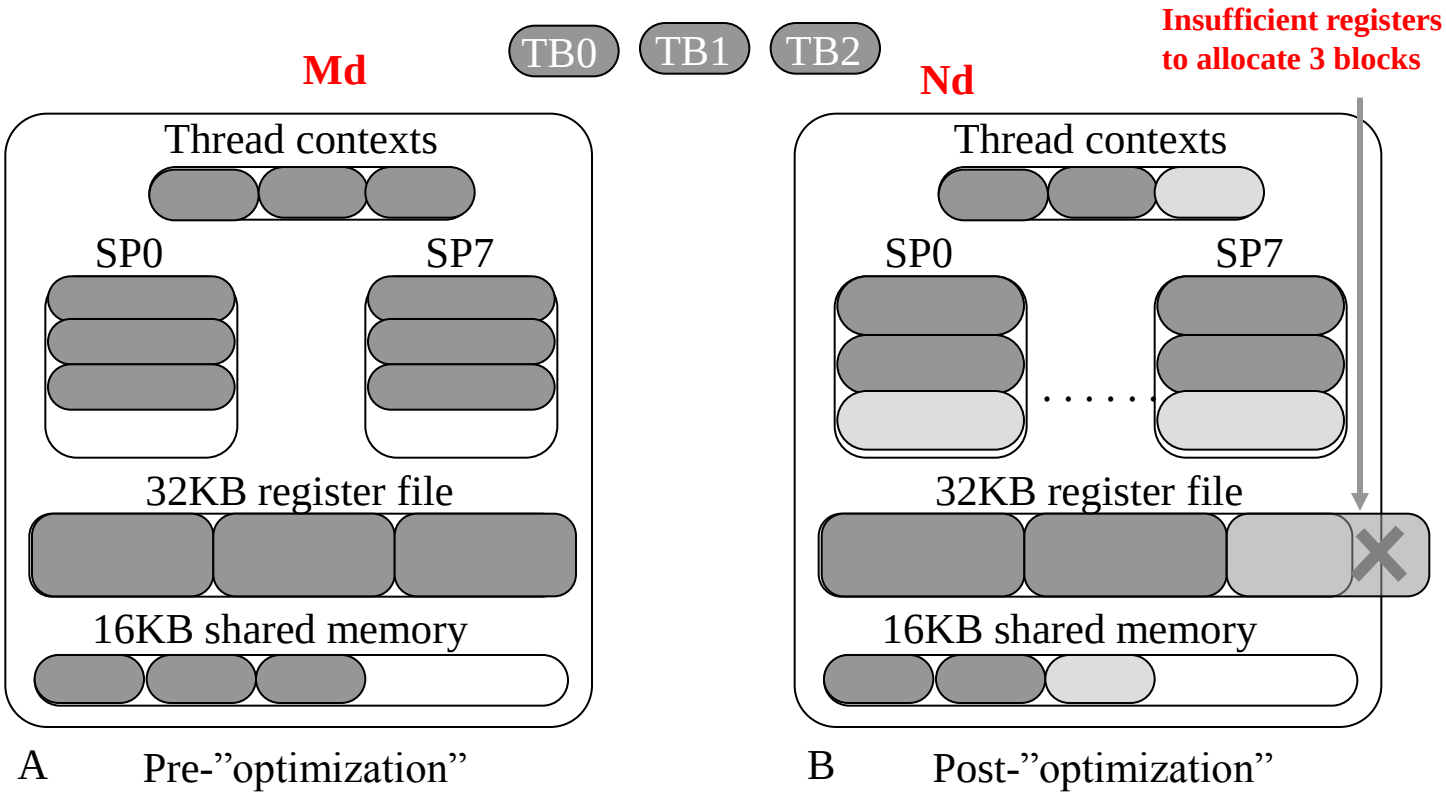


Figure. Interaction of resource limitations.

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 ( )
}
```

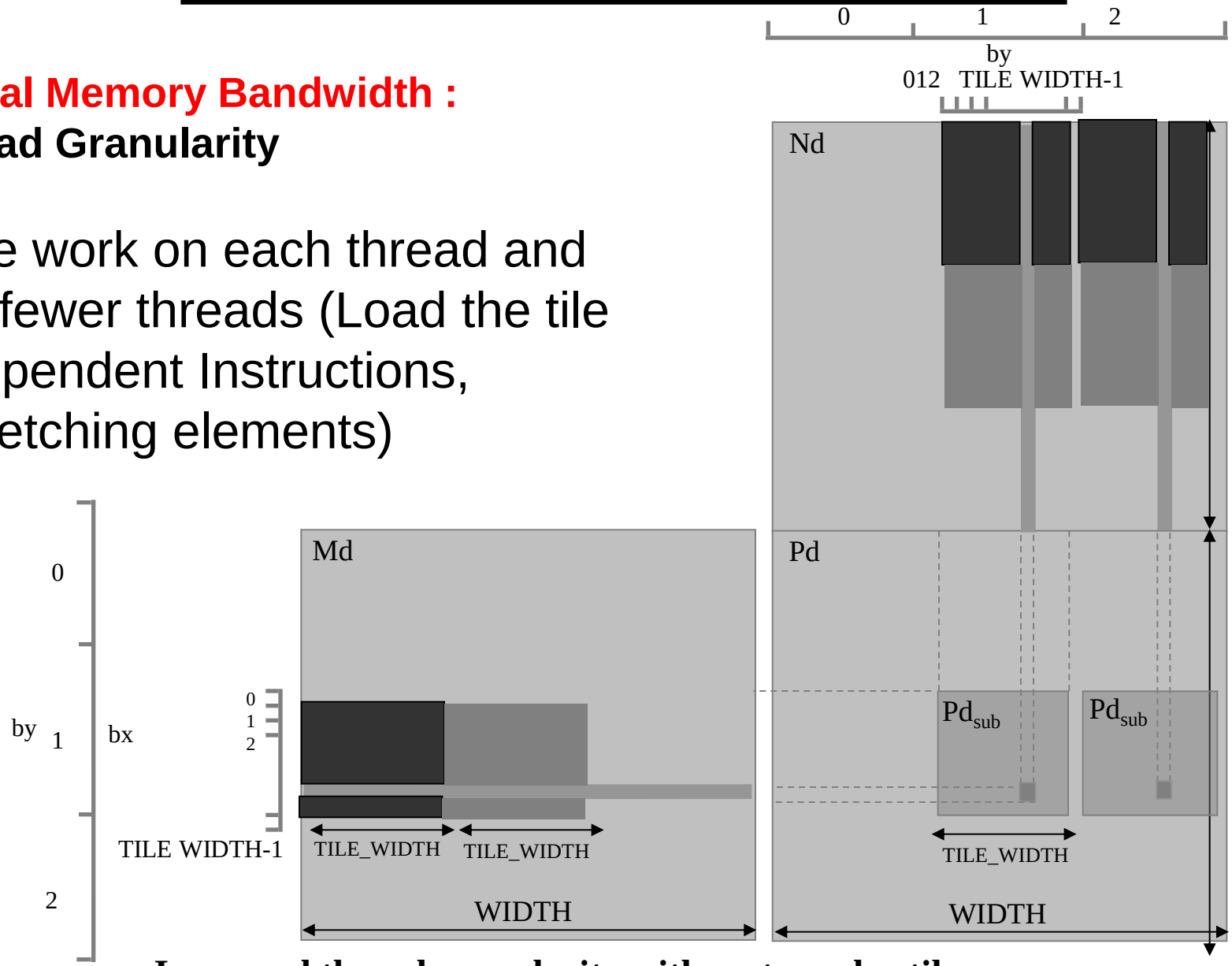
B With prefetching

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.

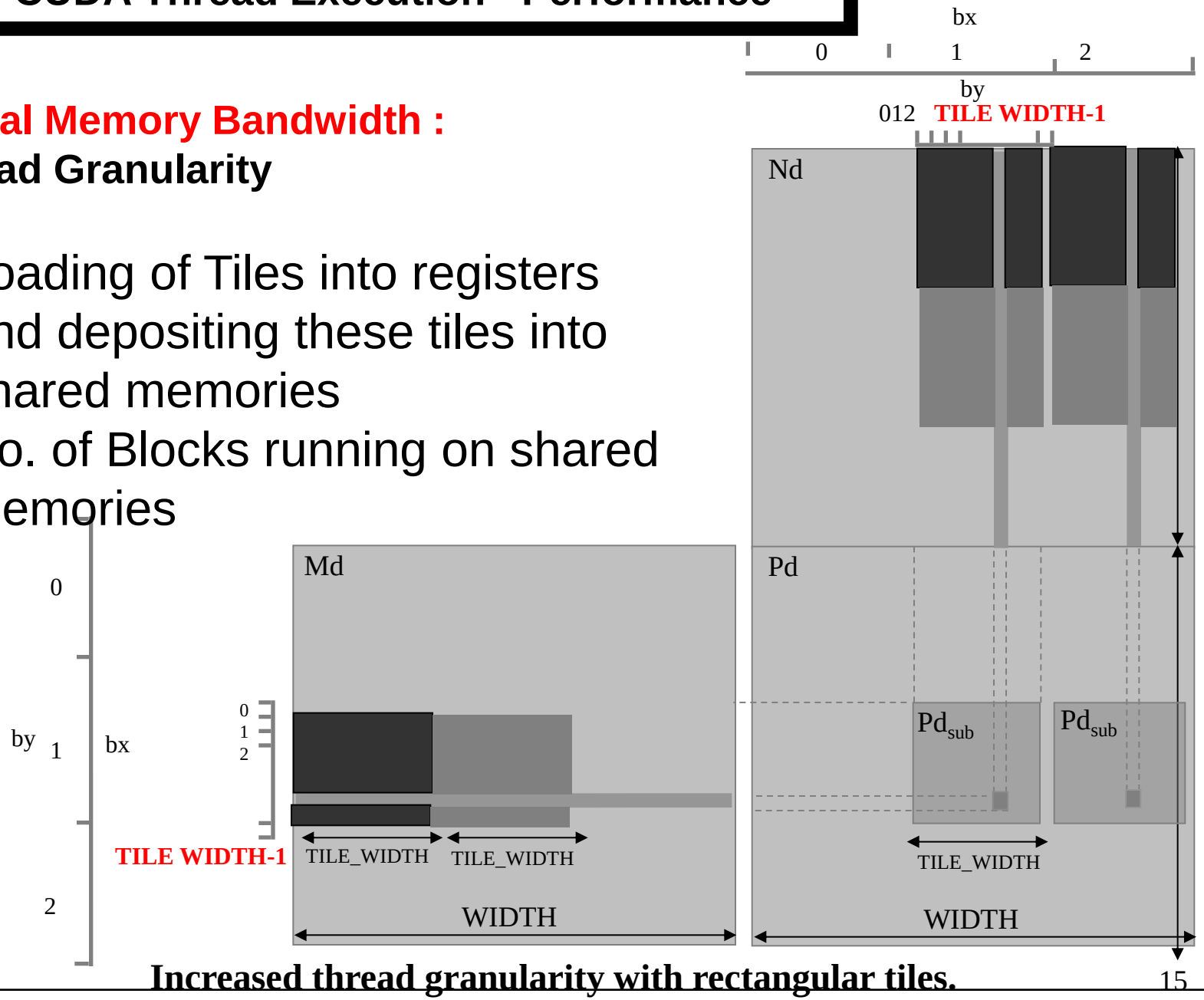
15

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.

15

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 6

CUDA 5.0 / NVIDIA Kepler GK110

(*) = 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**

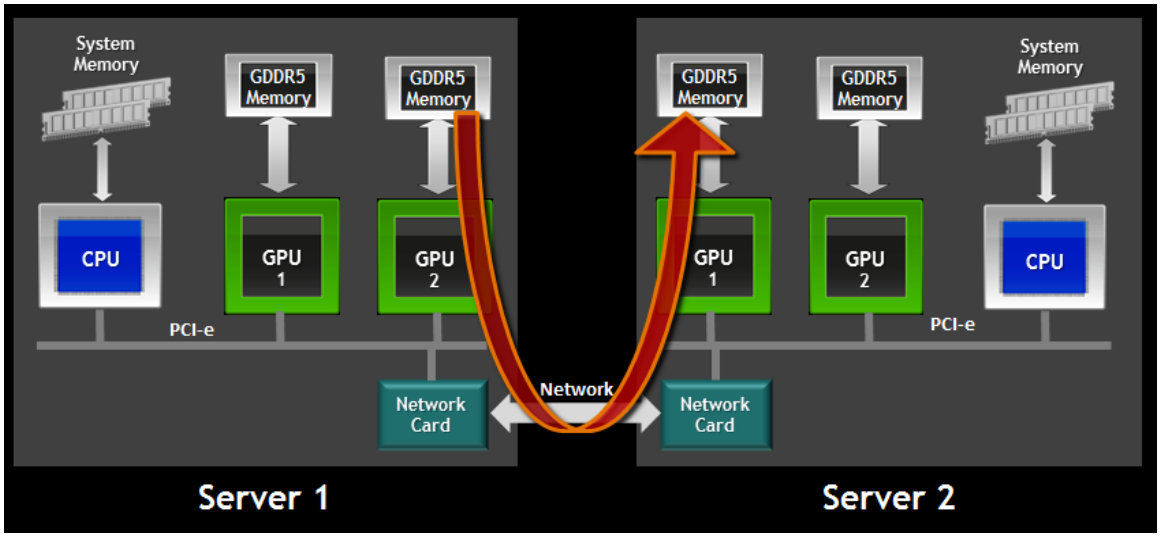
- **Peer-to-Peer memory access :** Optimize communication between GPUs using NUMA-style access to memory on other GPUs from within CUDA kernels
- **RDMA :** Eliminate CPU bandwidth and latency bottlenecks using direct memory access (DMA) between GPUs and other PCIe devices, resulting in significantly 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

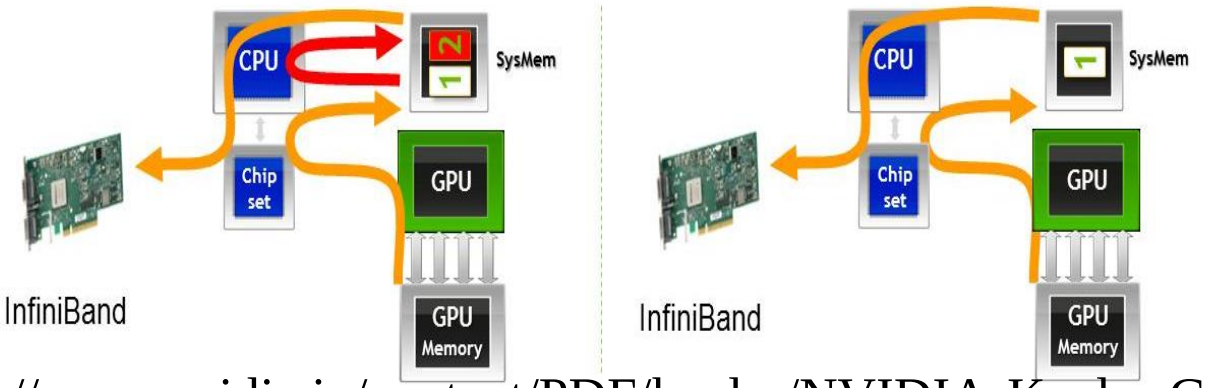
Same data copied three times

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



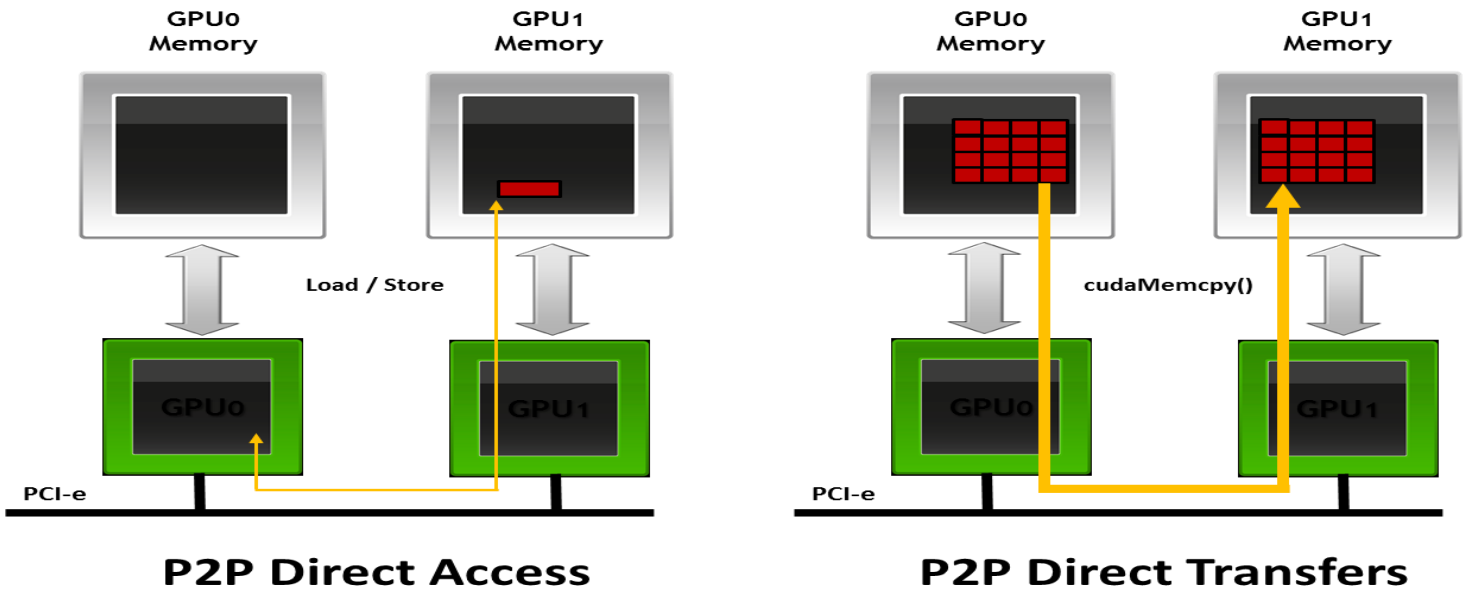
Source : http://www.nvidia.in/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

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:**

- **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

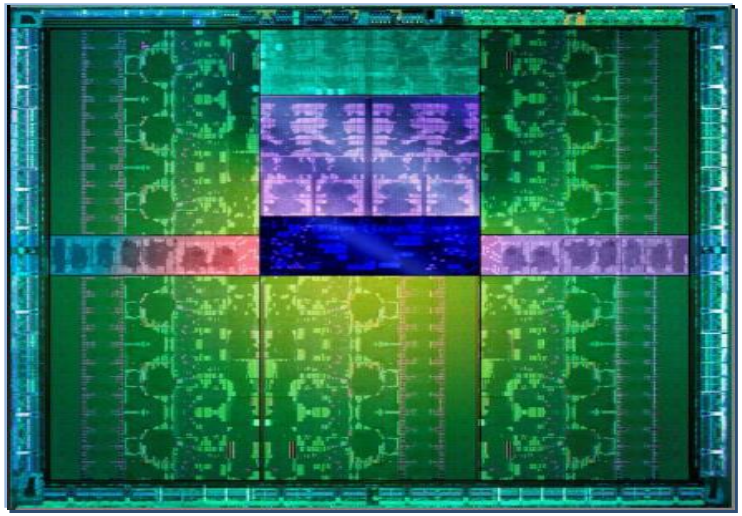
Kepler GK110-the new CUDA Compute Capability 5.0

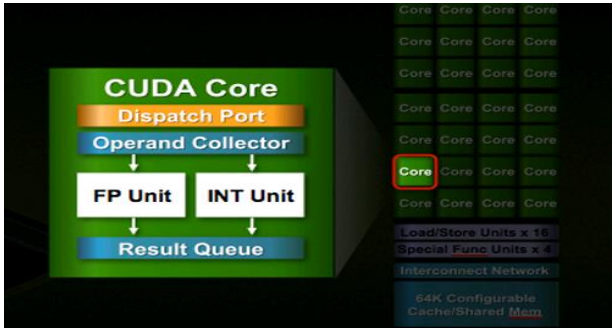
	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Threads Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessors	32768	32768	65536	65536
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

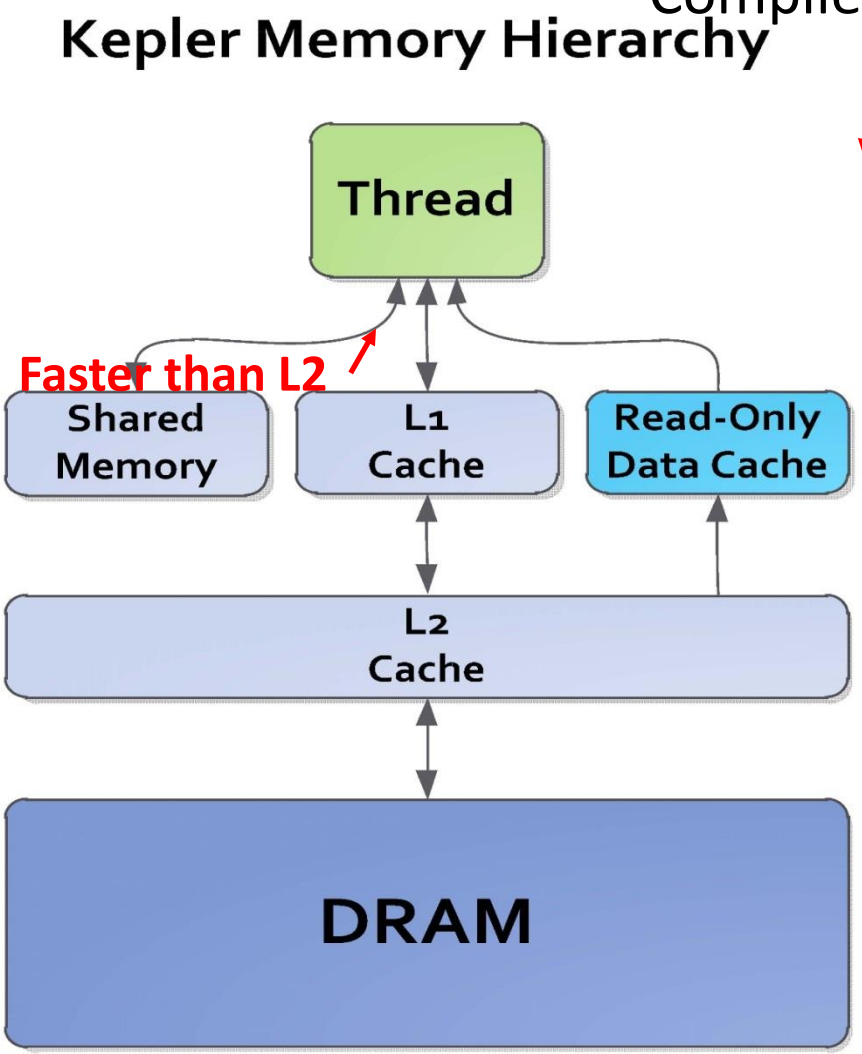
- A full k110 implementation includes 15 SMX units and six 64-bit memory controllers. Different products will use different configurations of K110.

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

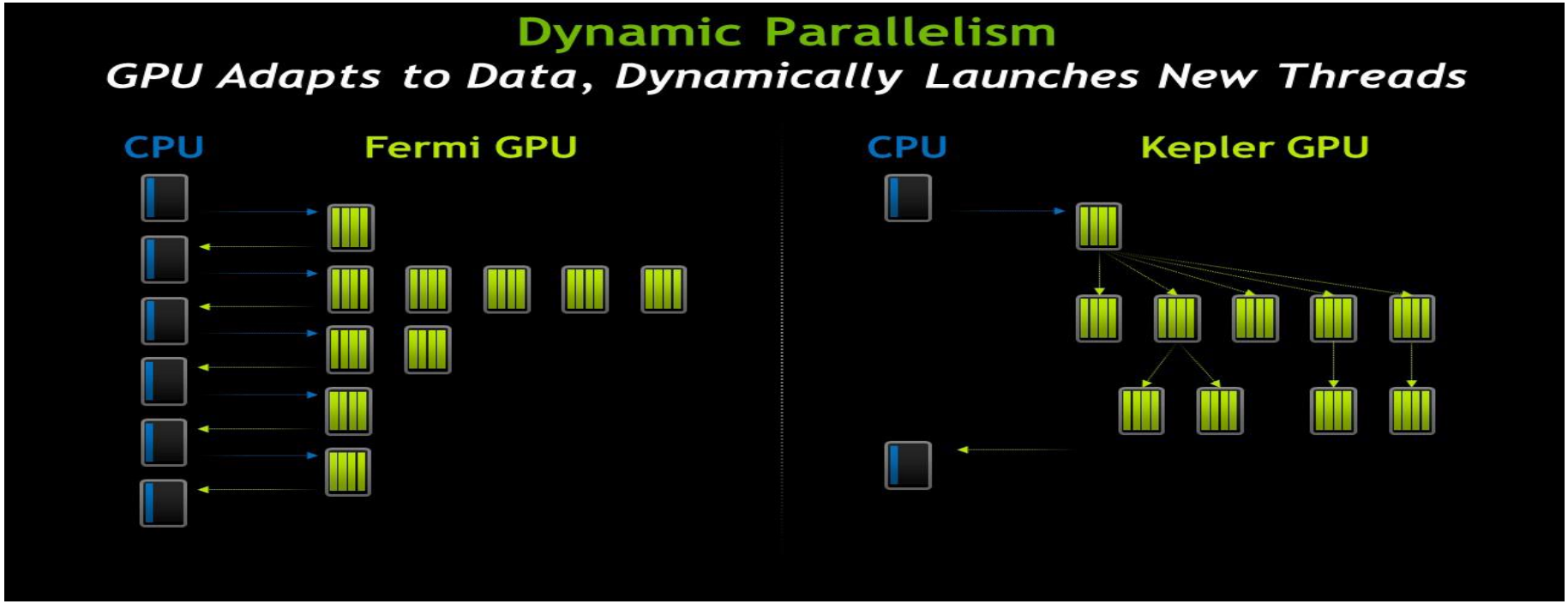
- 16 KB of shared memory with 48 KB of L1 cache

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)."



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 FERMI

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

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

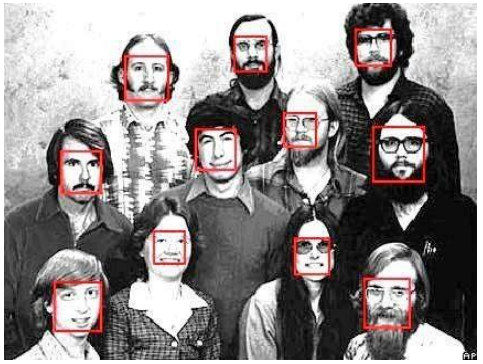
(*) = Speedup results were gathered using untuned and unoptimized versions of benchmarks & NVIDIA CUDA Prog. Env - This is C-DAC In-house HPC GPU Cluster project work

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.

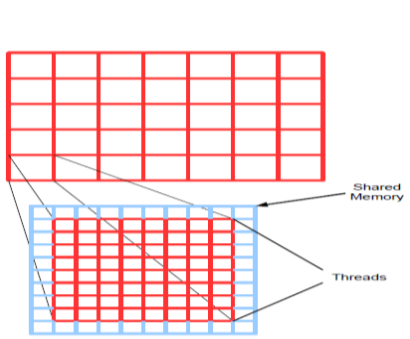
Courtesy : C-DAC Projects & Viola and Jones Alg.

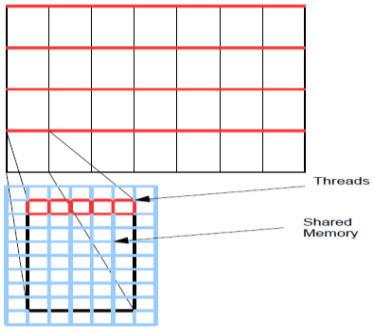
(*) = Speedup results were gathered using untuned and unoptimized versions of benchmarks & NVIDIA CUDA Prog. Env - This is C-DAC In-house HPC GPU Cluster project work

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





Edge Detection : Canny Edge Detection (*)

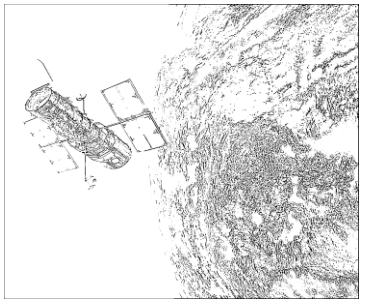
Pixels	OpenCV (Time in *ms*)	CUDA - GPU optimized Block Size of 8 x 8 (*Time in ms*)
512 x 512	8.40	0.62
1024 x 1024	28.01	2.30
2048 x 2048	108.52	9.34
4096 x 4096	398.14	38.17

Courtesy : Viola and Jones

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



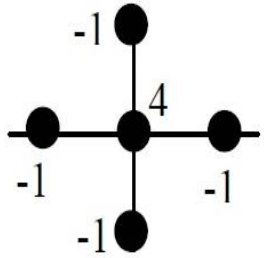
Courtesy : C-DAC Projects & Wikipedia 512*512 1024*1024

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

Poisson & Parabolic Eq. Solver

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

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

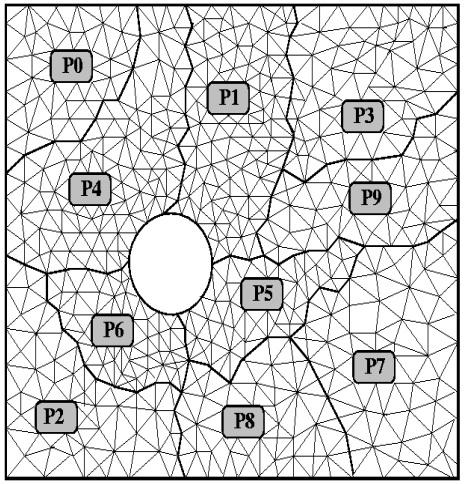


Data Re-arrangement Kernels & Jacobi / CG Methods

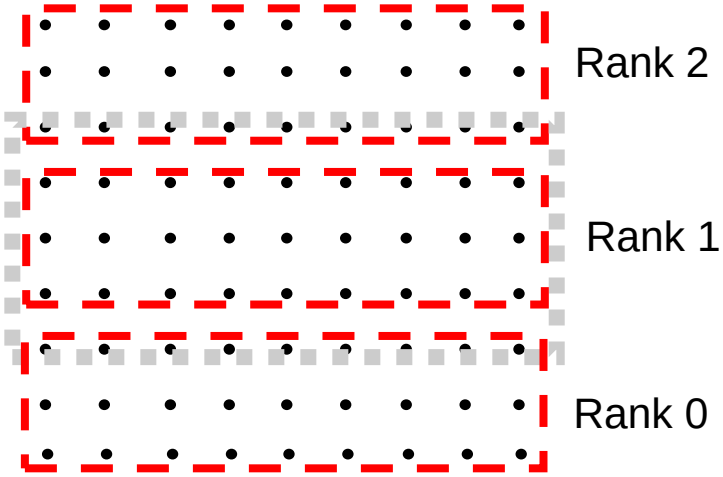
FEM

Graph Partition Software **METIS**

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

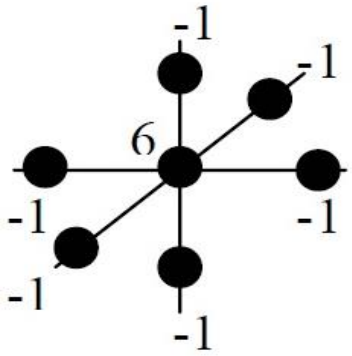


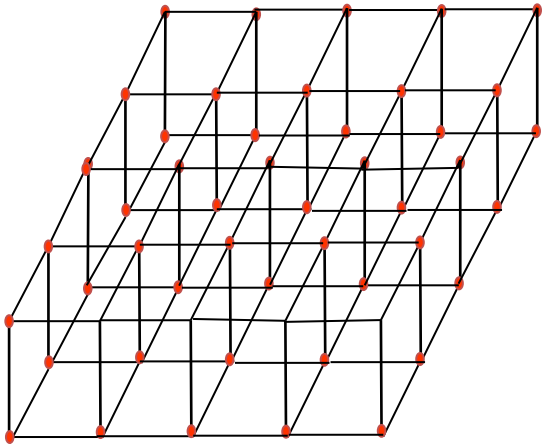
FDM

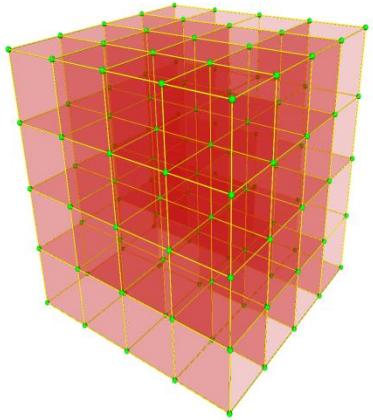


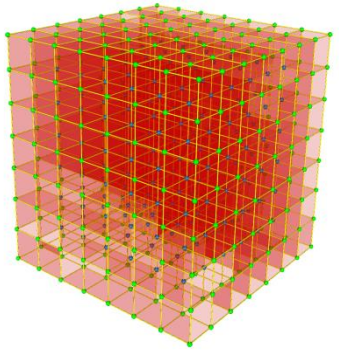
Courtesy : C-DAC HPC-FTE Student Projects

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









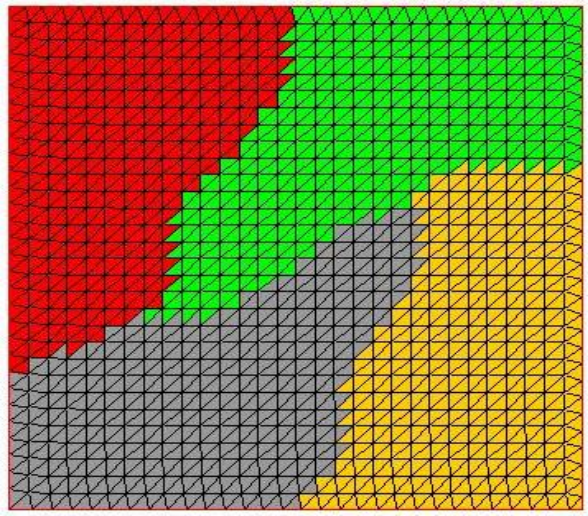
Stencil for Poisson Eq. in 3D

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

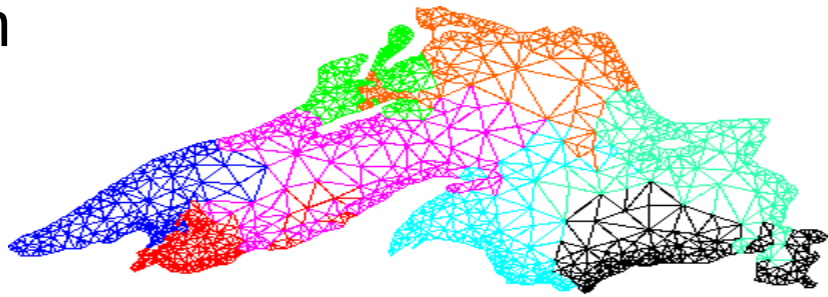
Type of Domain	Nodes/ (Partitions/ MPI Process)	Elapsed time (in seconds) MPI GPU Cluster		
		MPI	CUDA	OpenCL
2D-Structured grid -FDM (64X64)	4096 (1/1)	4.28		
	4096 (2/2)	3.12	0.82	1.28
2D-Structured grid -FDM (128X128)	16384 (1/1)	11.22		
	16384 (4/4)	3.74	0.98	1.42
3D-Structured grid -FDM (64X64X64)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

Domain decomposition, with blocks of size - 32x32

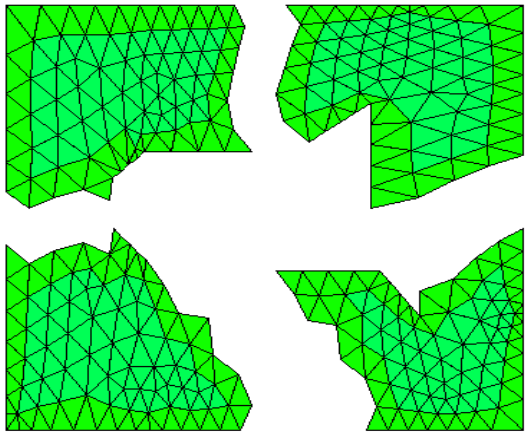
(*) = Speedup results were gathered using untuned and unoptimized versions of benchmarks & NVIDIA CUDA Prog. Env - This is C-DAC In-house HPC GPU Cluster project work

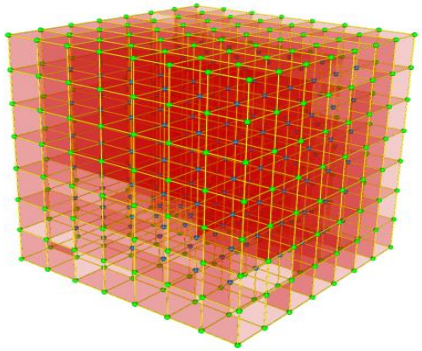
HPC GPU Cluster : Parallel Finite Element Method Comps. (Unstructured Grids)

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

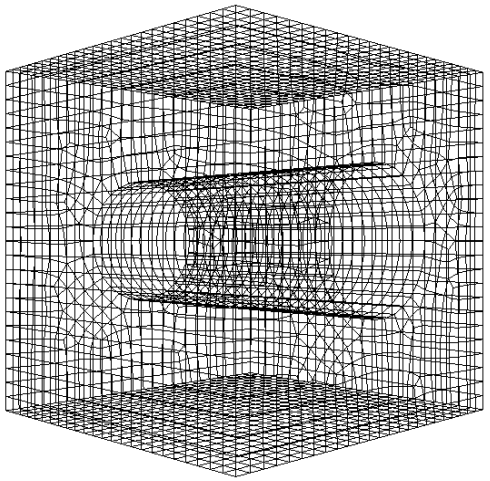
Domain decomposition :Graph Partitioning

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

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

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

NVML (NVIDIA Management Library)

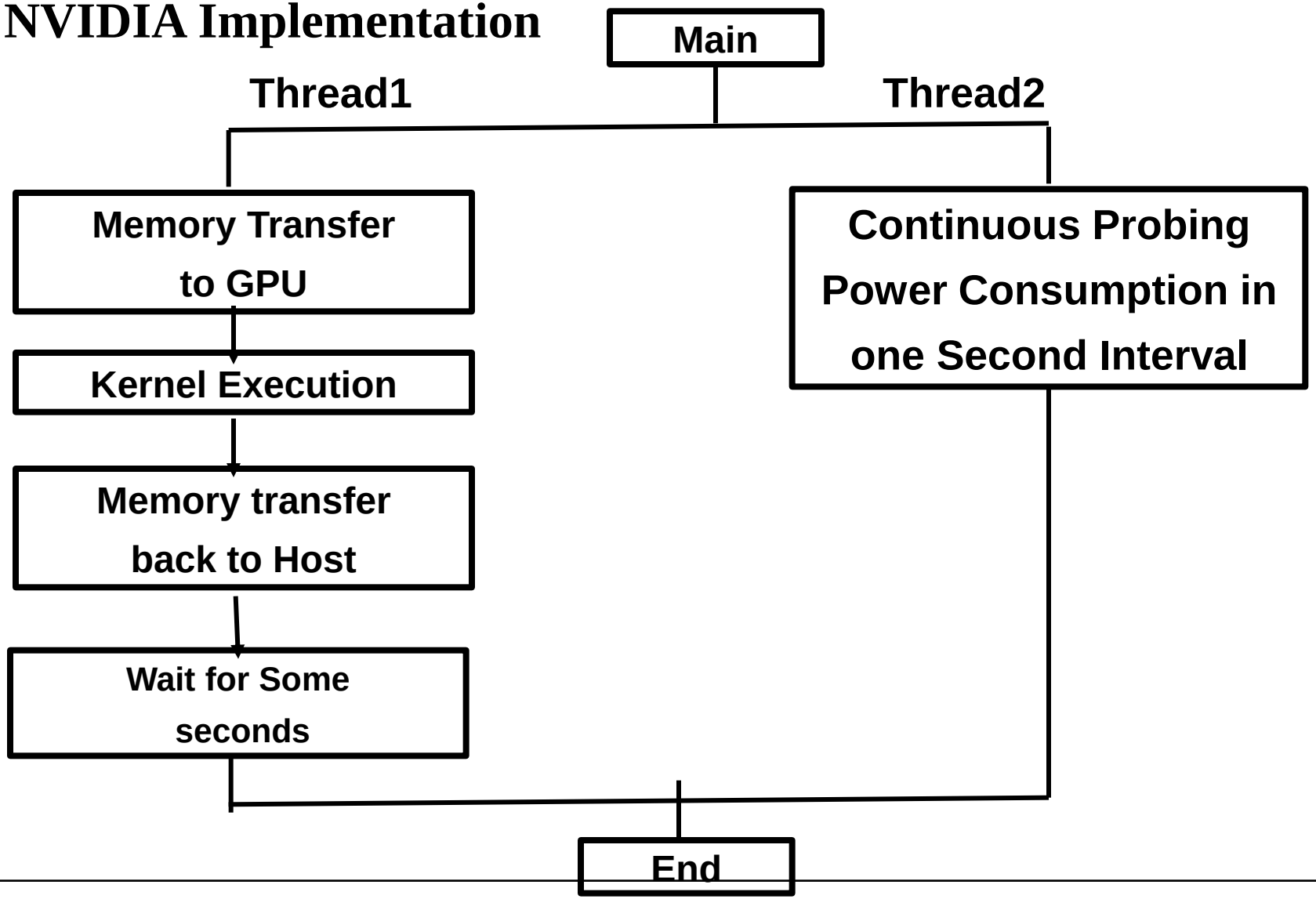
- **NVIDIA NVML : Power Measurement**

- 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

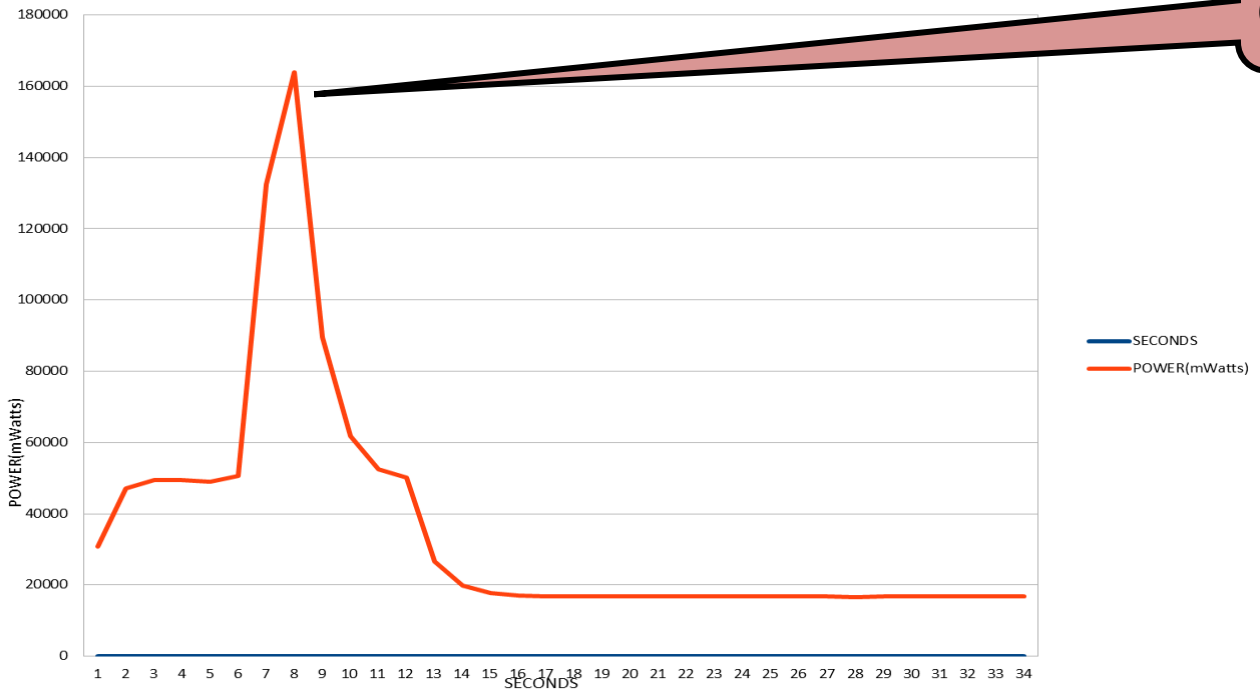
CPU + GPU Time (Sec): 2.575

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

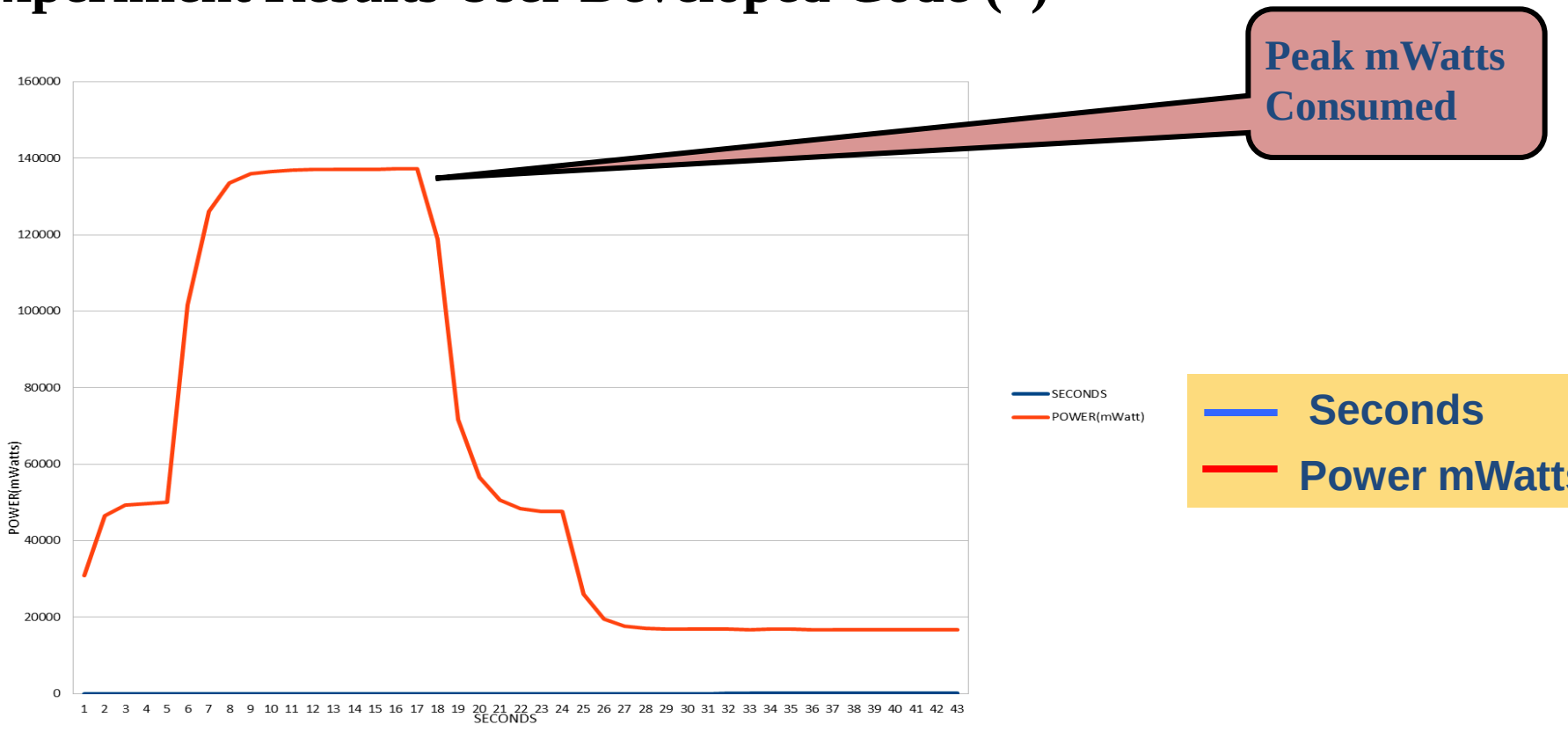
Time (Sec.)	Power in milliwatt
6	101504
7	**133627**
8	**135000**
10	**136574**
12	**137145**
16	**137330**
17	118776
18	71695
19	56504

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 (*)

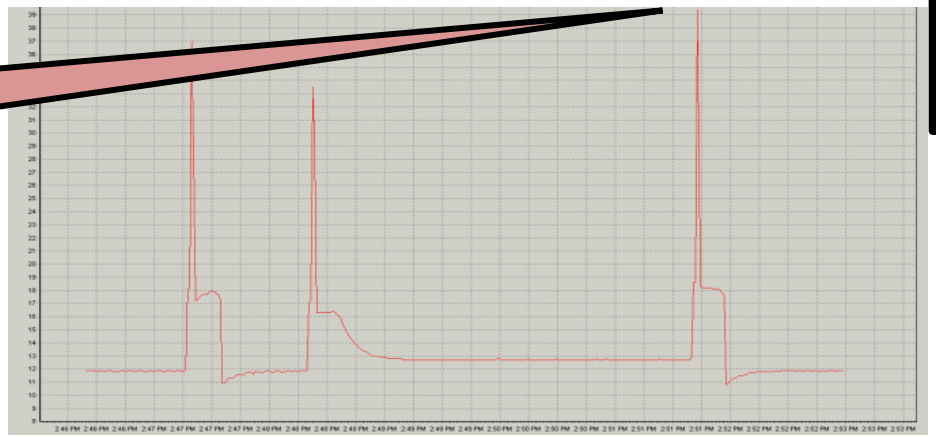
Matrix-Matrix Multiplication			
CUBLAS (Vendor)		User Code (IJK loop)	
GFLOPS	Time (Sec)	(GFLOPS)	Time (Sec)
125.7783	0.00834	28.9092	0.03627
125.7004	0.00834	28.9070	0.03627
125.7426	0.00834	28.9085	0.03627

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

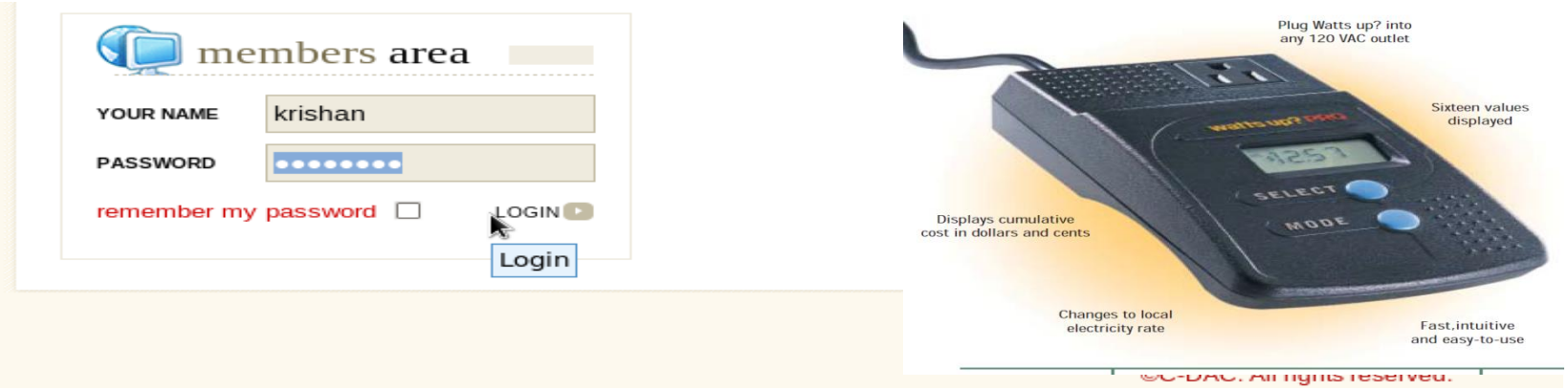
Seconds
Power Watts

(*) = 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

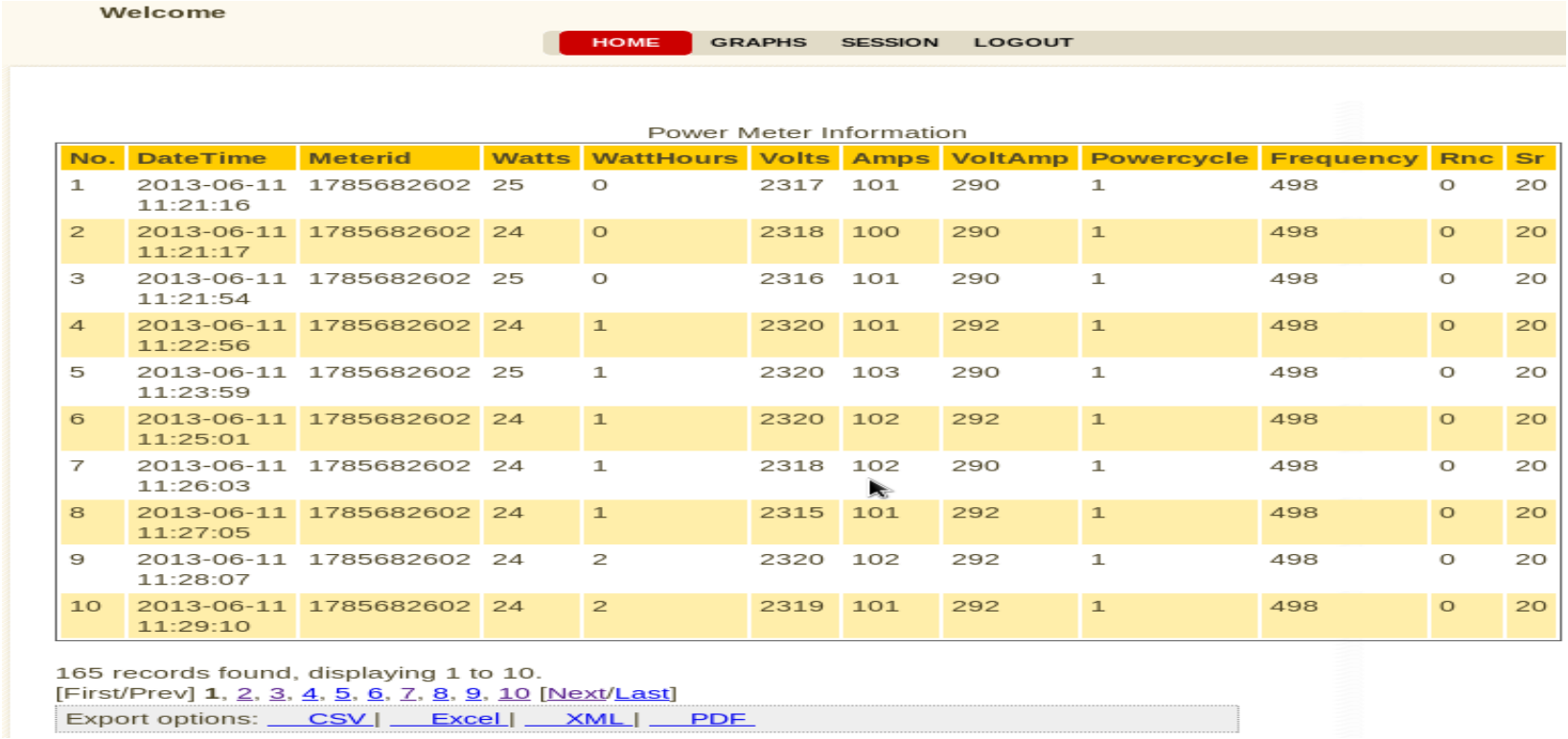
- **Login to portal**



- **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

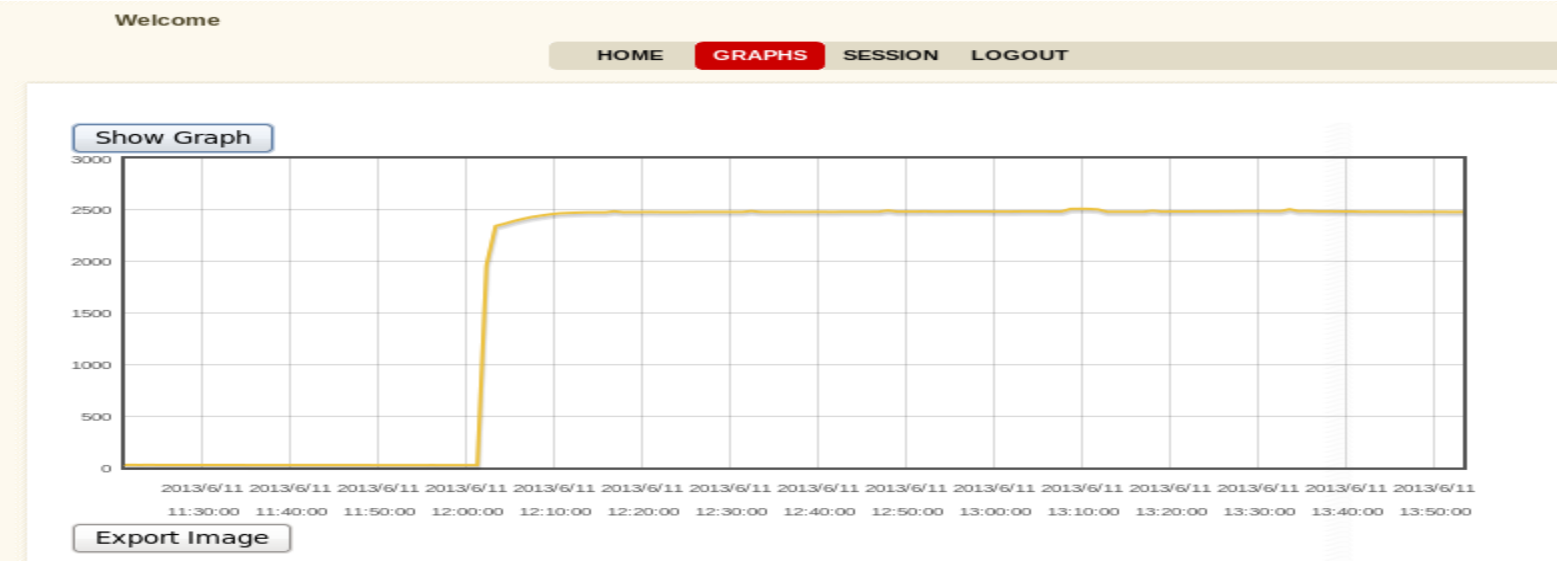
165 records found, displaying 1 to 10.
[First/Prev] 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 [Next/Last]
Export options: CSV | Excel | XML | PDF

(*) = 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

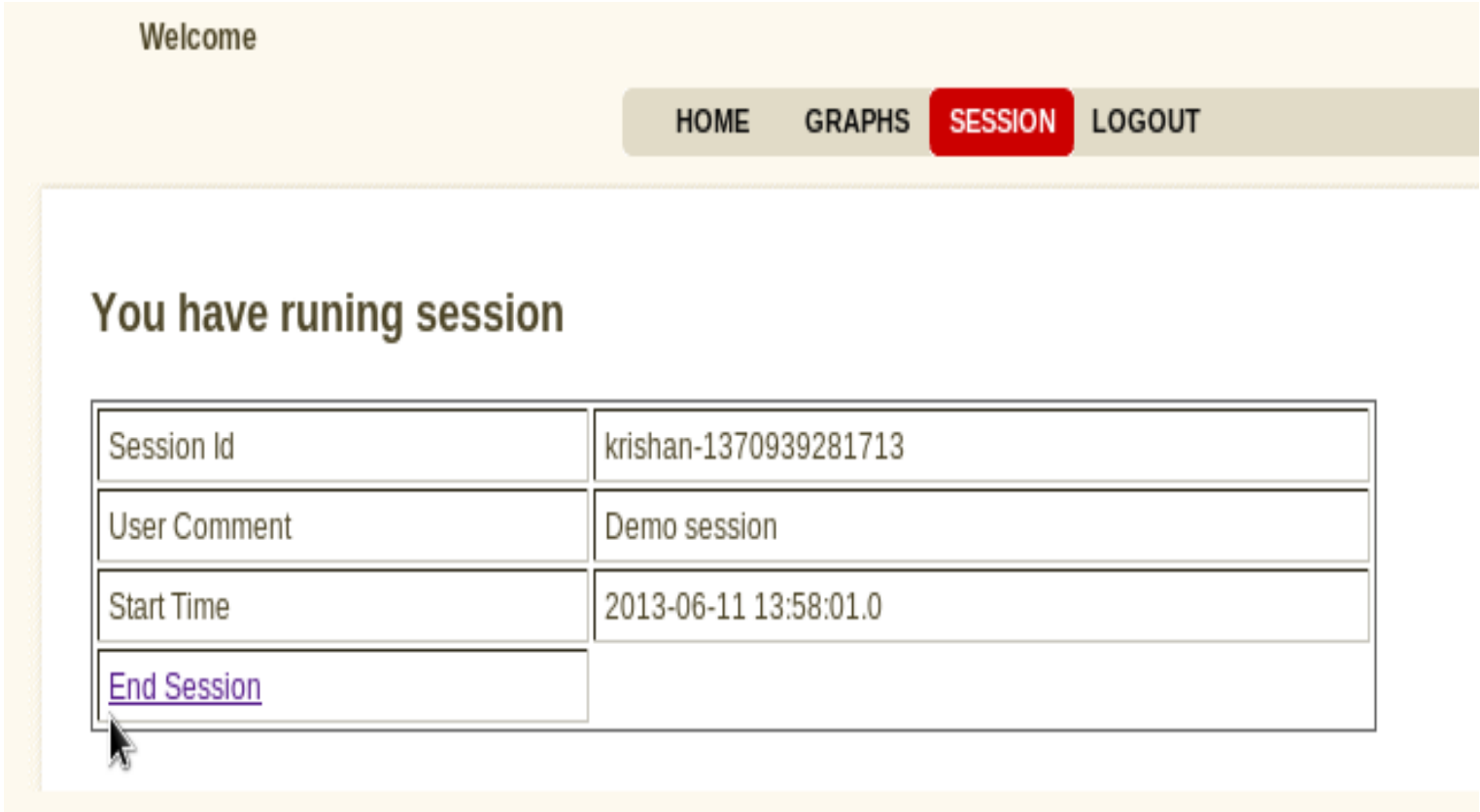
Experiment Results User Developed Code (*)

- 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



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: 4.0
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: 4.0
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

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

Thank You

Any questions ?