Four-day Technology Workshop
on
Hybrid Computing - Coprocessors & Accelerators -
Power-aware Computing & Performance of Application Kernels
(Initiatives on Measurement of Power Consumption & Performance)
(hyPACK – 2013)

Dates: October 15, 2013 (Tuesday) – October 18, 2013 (Friday)

Venue: Centre for Modelling Simulation and Design (CMSD)
High-Performance Computing (HPC) Facility
University of Hyderabad, Hyderabad

Module-4
Documentation for CUDA enabled NVIDIA Laboratory
Part-I

1. Example 1: deviceDetails.cu

Objective: To find out the number of CUDA enabled NVIDIA devices and their properties that are present on the current system

Description:
This is a sample program that queries using the cuda API calls about the number of cuda enabled devices that are present on the system and the various properties of the devices like, the device model, max number of threads per block, compute capability, warp size, available Global,shared,and constant memories etc.

CUDA API Used: **cudaGetDeviceCount(),cudaGetDeviceProperties()**
Input: none
Output: The various properties of all the devices that are present on the current system

2. Example 2 : globalMemoryAccessPatterns.cu

Objective: To demonstrate different access patterns of global memory and compare bandwidths achieved for each of the access pattern.

Description:
This is sample program to demonstrate the different access patterns of the global memory. It measures corresponding bandwidth for each of the access patterns.

Coalescing: Simultaneous memory access by threads in a half warp (16 threads) can be combined into single memory transaction of 32,64 or 128 bytes.
It implements copy kernel by using following different access patterns:

Coalesced float memory access :
It is the access pattern where successive threads are accessing the successive memory locations in the Global memory and the array is aligned. It results in a single memory transaction.

Coalesced float memory access (divergent warp):
It is access pattern where successive threads are accessing the successive memory locations in the Global memory and array is aligned but some of the threads are not accessing any memory location. It results in a single memory transaction.

Non-sequential float memory access: (Non-Coalesced)
It is access pattern where successive threads are not accessing the successive memory locations in the Global memory but the starting address is aligned.

Access with a misaligned starting address: (Non-Coalesced)
It is access pattern where successive threads are accessing the successive memory locations in the Global memory and the starting address is misaligned.

Non-contiguous float memory access: (Non-Coalesced)
It is access pattern where the successive threads are not accessing the successive memory locations and the starting address is aligned. (ex. one address left un-accessed)

Non-coalesced float3 memory access:
This access pattern involves accessing contiguous float3 element's first component by the contiguous threads and the starting address is aligned.

The access patterns that are considered in the program are meant for the devices with compute capability <= 1.1

CUDA API Used: **cudaMalloc(),cudaEventCreate(),cudaEventRecord(), cudaEventSynchronize(),cudaEventElapsedTime(), cudaEventDestroy(), cudaFree()**

Input: none
Output: The different bandwidths (in GB/sec) that are achievable by the six patterns and by the cudaMemcpy routine.

3. Example 3: coalescedFloat3Access.cu

Objective: to demonstrate the usage of shared memory to get coalesced data access pattern for float3 array and the corresponding advantages in terms of the bandwidth that is achievable

Description:
This a sample program to demonstrate that for seemingly non-coalesced accesses can be

made coalesced by using shared memory. The two patterns that are demonstrated:

Non-coalesced float3 memory access :
In which, each thread accesses an element from float3 array and copies component wise to another element in the output float3 array

Coalesced float3 memory access using shared memory :
In which, all threads in a block coordinate to load the corresponding elements into shared memory then store the values in the corresponding element in output array such that both loads and stores can be "coalesced"

CUDA API Used:

__syncthreads(),cudaMalloc(),cudaEventCreate(), cudaThreadSynchronize(),cudaEventRecord(),cudaEventSynchronize(), cudaEventElapsedTime(),cudaEventDestroy(),cudaFree()

Input: none

Output: The different bandwidths (in GB/sec) that are achieved by the non-coalesced access and the corresponding coalesced access using the shared memory

4. Example 4: sharedMemoryReadingSameWord.cu

Objective: To demonstrate the achievable shared memory bandwidth while reading the same word

Description: This is an example code to demonstrate that while reading the same word by all the threads, there will not be any serialization even though all threads are accessing from the same bank.

The 32-bit word gets **broadcasted** to all the threads - hence bandwidth can be comparable to the value got when there were no bank conflicts

Note: The code gives warning while compiling that the array is used before its value is set. This warning can be ignored, as we are not doing any nontrivial computation using the data. The main aim of the code is to generate the access patterns rather than doing any computation.

CUDA API Used:

cudaEventCreate(),cudaEventRecord(),cudaEventSynchronize(), cudaEventElapsedTime(),cudaEventDestroy()

Input: none

Output: The different bandwidths that are achieved while all the threads read the same 32-bit word and when all threads read words from different banks with no bank conflicts

5. Example 5: sharedMemoryRestructuringDataTypes.cu

Objective: To demonstrate the achievable shared memory bandwidth while accessing arrays of different inbuilt data types

Description: Example code to demonstrate the different shared memory bandwidths achieved when

1) accessing a 3d array of floats
2) accessing a float3 array
3) accessing a 4d array of floats
4) accessing a float4 array

In all the kernels, the arrays are just being initialized.

CUDA API Used: **cudaEventCreate(),cudaEventRecord(),cudaEventSynchronize(), cudaEventElapsedTime(),cudaEventDestroy()**

Input: none

Output: The different bandwidths of the shared memory that are achieved in the above-mentioned accesses

6. Example 6: sharedMemoryStridedAccessPatterns.cu

Objective: To demonstrate bank conflicts that can occur while accessing the shared memory

Description:
This is a sample code to demonstrate the different strided access patterns in the shared memory and the corresponding bandwidths of the shared memory that are achievable.

A series of I/O requests are considered to be a *simple-**strided*** access pattern if each request is for the same number of bytes, and if the file pointer is incremented by the same amount between each request. The following strided accesses are demonstrated:

stride of one 32-bit word : causes no bank conflicts
two 32-bit words : causes 2-way bank conflicts
three 32-bit words : causes no bank conflicts
eight 32-bit words : causes 8-way bank conflicts
sixteen 32-bit words : causes 16-way bank conflicts

CUDA API Used: **cudaEventCreate(),cudaEventRecord(),cudaEventSynchronize(), cudaEventElapsedTime(),cudaEventDestroy()**

Input: none

Output: The different bandwidths of the shared memory that are achieved in the above strided accesses

7. Example 7: SOAvsAOS.cu

Objective: To demonstrate the advantage of having Structure of arrays rather than array of structures in the application while representing data and the corresponding advantages in terms of the bandwidth of the global memory that is achievable

Description:

This example takes "Triangle" as structure with three arrays of three floating points each representing the three vertices of a triangle the same information is also is stored using a structure "Triangles" which has arrays for each field of each vertex.

Both the representations are initialized generating typical access patterns that will be present while accessing those structures.

CUDA API Used: **cudaMalloc(),cudaMemcpy(),cudaEventCreate(),cudaEventRecord(), cudaEventSynchronize(),cudaEventElapsedTime(),cudaEventDestroy(), cudaFree()**

Input: none

Output: The different bandwidths of the global memory that are achieved by having different data representations.

Remarks:

1. In all the above programs the block size and the array size are *#defined* constants which can be changed by changing the definitions explicitly in the corresponding source code of the program. But by changing the constants we may need to check weather they are with in the boundaries of the maximum available blocksize, shared memory etc.

2. The validation and the error checking is very minimal as the aim of the codes is to demonstrate the different issues related to the memory accesses and the relative study of different strategies and access patterns that give different bandwidths with respect to each other.

3. The code - "sharedMemoryReadingSameWord.cu" -- gives warning while compiling that - the array is used before its value is set. This warning can be ignored, as we are not doing any nontrivial computation using the data. The main aim of the code is to generate the

access patterns rather than doing any computation.

4. For more detailed information, please refer to the README file attached with each code.

Module-4
Documentation for CUDA enabled NVIDIA Laboratory
Part-II

1. Example 1: Stream Benchmark in CUDA for GPU

Objective: The program measures the sustainable memory bandwidth (Global memory of the NIVIDA GPU device).

Description:
This is a Stream Benchmark for GPU. It finds the bandwidth of global memory of GPU card by timing the four operations - Copy, Scale, Add and Triad.
The different operations were timed using CUDA events provided by runtime library.

Input: None

Output: Captures the bandwidth in GB/s with time (i.e average, minimum & maximum) taken for each of the four operations (i.e COPY, SCALE, ADD and TRIAD)

2. Example 2: blockPartitioning.cu

Objective: To demonstrate the global memory bandwidth differences for varying block sizes.

Description:

- This Program measures the bandwidth of global memory for the different block sizes and fixed length array in a copy operation.
- In CUDA choosing the appropriate block size for your application is left to the programmer. But the block size affects the global memory bandwidth and performance.
- This code tests the same for different block sizes in a simple copy operation. Array size is fixed to maximum limit possible.
- Best size for block depends on the application. In this case, when the block size is 128 the occupancy is close to 100% and hence we get maximum bandwidth whereas occupancy is just 33% when block size is equal to 32.
- Minimum scheduling unit in a SMP is a warp. Each instruction on SMP could take maximum of 24 cycles.

Input: None

Output: Global Memory Bandwidth in GB/s for different block sizes.

3. Example 3: vectorModel.cu

Objective: Bandwidth improved when thread handles more than one element by making use of GPU as a 32-way SIMD processor

Description:

- This Program measures the bandwidth of global memory for simple initialization kernel operation [a(i) = value].
- GPU can be considered as 32-way SIMD processor.
- This program demonstrates that if each thread handles more than one data element (here 4) then we can achieve better performance.
- Array size & block size are fixed as this program is for demonstration only.
- Factor corresponds to the no. of data elements handled by each thread. It is fixed to 4 but could be varied.

Input: None

Output: Global memory bandwidth achieved (GB/s) and timing (average) for two initialization kernels – normal and with vector model.

4. Example 4: partitionCamping.cu

Objective: To demonstrate the difference in bandwidth achieved when blocks access global memory with and without partition camping.

Description:

- This Program measures the bandwidth of global memory for the initialization operation [a(i) = value] using NVIDIA GPU for 2 kernels that access global memory with and without partition camping
- This code is written for 8x and 9x series of NVIDIA GPUs. The global memory of these gGPUs has 6 partitions.
- There code was tested on 9600 GT accelerator card which has 8 multiprocessors. Since there are 6 partitions we cannot write a kernel free of partition camping. At least 2 partitions will experience collisions from atleast 2 blocks.
- Two kernels, which initialize a fixed length array, are written such that one minimizes partition camping (i.e. initializationWithoutPartitionCamping) by accessing global memory more uniformly.
- Block size is fixed to 64 floats (256 bytes) as width of each partition is 256 bytes (i.e. consecutive chunks of 256 bytes will be stored in different partitions). If active blocks request data in different partitions then there is no partition camping.
- Array size is fixed to 2195264 so that total no. of blocks is equal to 34301 which is relatively prime to 6. Moreover, 34302 is multiple of 6. Hence in the second kernel consecutive blocks (in terms of ID) access chunks of 64 floats in stride of 6 (blocks). This will lead to heavy partition camping.
- Array size and block size should not be changed. They are fixed for the purpose of demonstration only.

Input: None

Output: Global memory bandwidth achieved (GB/s) and timing (average) for two initialization kernels – with and without partition camping.

5. Example 5: warpDivergence.cu

Objective: To demonstrate the difference in bandwidth achieved when threads within a warp follow different execution paths.

Description:

- This Program measures the sustain memory bandwidth of global memory for the initialization operation [a(i) = value] using NVIDIA GPU which has a SIMT architecture.
- GPU employs a SIMT (single instruction multiple thread) architecture in which the threads of a block are executed in groups of 32 called warp. A warp executes a single instruction at a time across all its threads, and it makes substantial difference in performance if threads within a warp follow different execution paths.
- In this program there are 4 kernels with varying no. of branch instructions. The kernel with more branches has more execution paths within a warp as a result some threads have to stall leading to performance degradation.
- Array size and block size could be changed.

Input : None

Output : Global memory bandwidth achieved (GB/s) and timing (average) for initialization kernels with varying no. of execution paths.

Remark: For more detailed information, please refer to the README file attached with each code.

Source & References: As given in PEMG-2010 CUDA enabled NVIDIA GPU references