Programming of shared memory GPUs shared memory systems

Jean-Philippe Bergeron
School of Information Technology and Engineering
University of Ottawa
Ottawa, Ontario
Email: jberg081@uottawa.ca

***Abstract*—The concept of shared memory is known to engineers for a long time: a large block of memory shared between processors to allow for synchronization. But what if there where thousands of threads with hundreds of processors accessing a single memory block. This challenge is discussed in this paper. There are APIs used to program the graphics processing unit (GPU) which takes advantage of the high number of processing elements. The memory architecture is specifically designed in order to obtain the maximum bandwidth. A great speedup is achieved after optimizations.**

I. Introduction

In a multiprocessor system relying on shared memory, the bottleneck is the memory modul[illegible]. The solution is usually to use distributed memory to solve the scaling problem, but this non-uniform memory access (NUMA) architecture has other implications such as very different access time depending on the memory location. The novel approach used by NVIDIA[2] and ATI [3] uses a single memory module, and an array of single instruction, multiple data (SIMD) processors. The novel approach forces the programs to be built in parallel, the algorithms have to be rewritten. This is made possible by using fine-grained APIs which schedule thousands of threads with a minimal overhead. The memory scaling problem is solved by using a thread scheduler. It orders the memory access and allows the memory to send multiple words to multiple threads in one operation.

II. Prior Art

A. What is a shared memory system?

In a shared memory system, any processor can directly reference any memory location and there is a single memory space accessible to the programmer. There might be an interconnection network between a processor and the memory, but the processor can only connect to the memory. There cannot be direct communication between processors in this architecture. The only medium of communication is the shared memory. To communicate to another processor, one processor has to write to the memory and the following processor will read from the same address. This access to the memory is possible since one or multiple blocks of memory appear as one large block to the programmer. The memory modules are organized continuously so that the physical location is abstracted from the programmer; All he knows is the address of the memory he wants.

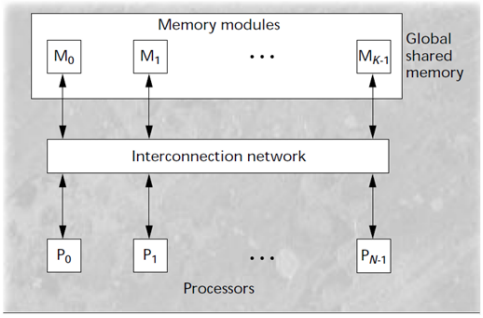


Fig. 1. Shared memory with multiple processor and memory

There might be multiple memory modules. As said by [4], in the case where there is only one module, every processor will access that module and there might be memory contention since multiple requests may be sent at the same time by multiple processors. On the other hand, by following the approach illustrated in Figure 1, where the memory contention may be smaller by using multiple memory modules since the requests from the processors will be distributed. The circuitry will be a little more complex since the address needs to be looked at to route the requests to the right memory.

The usage of shared memory for multiples processors adds more requirements to the logic on the board. With the arrival of multiple processors, the memory can change behind the CPU cache by another processor. It is important for all of the processors to have the most up-to-date value even if they are utilizing a cache. This requirement forces the implementation of a software or a hardware cache coherency. This will ensure that the cache gets invalidated or updated when another process writes to the memory.

Another important factor while programming with multiple processors, is the possibility of having exclusivity to certain variables. This is usually done using mutexes. To support a mutex on a shared memory system, there is the necessity to have atomic operations. An atomic operation is an operation which cannot be divided or interrupted by another processor.

B. Examples of shared memory system

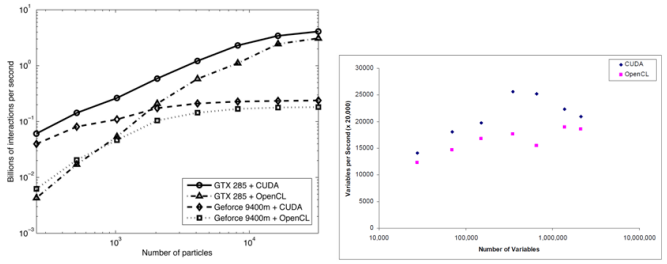
Shared memory systems are popular because of their ease of use. The central processing unit (CPU) shares the main memory among many processors to provide a way for communication between threads. On the graphics processing unit (GPU), it typically runs hundreds to thousands of threads in parallel, organized in SIMD blocks as written in [2].

III. Frameworks

Due to the massive number of threads, applications have to be rewritten entirely and numerous frameworks have been implemented to make this task easier. In the languages, kernels of code are compiled to run in parallel on a selected platform. In the kernel, there are variables indicating the processor id and the number of processors. It is usually possible for the developer to select how many processors it wants to use.

Apple created the open computing language (OpenCL) and it was submitted to the non-profit Khronos Group in 2008 described in [5]. One of the strength of that language is the heterogeneous computing, where the same code run on both the CPUs, the GPUs and other processors.

NVIDIA created the compute unified device architecture (CUDA) to allow kernels of code to run on their GPU. They developed a large community and a large number of samples and APIs. They have a library to perform generic math operation named CUBLAS and a library to perform fast Fourier transform efficiently named CUFFT.



(a) Performance comparison of OpenCL and CUDA QMC kernels [7] [6]

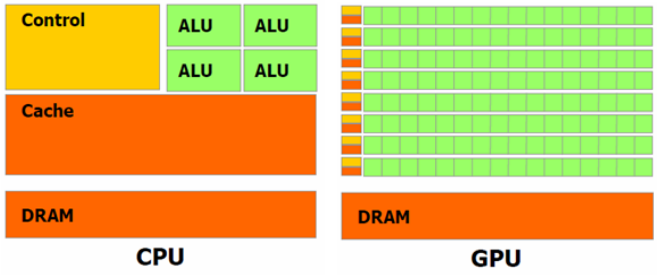
(b) Processing speed for different problem sizes [7]

Fig. 2. Performance of CUDA and OpenCL

TABLE I
CUDA vs OpenCL

Feature	OpenCL	CUDA
C Language Support	Yes	Yes
CPU Support	Yes	No
License	Royalty Free	Proprietary
Community size	Medium	Large
Speed	Fast	Very fast
CUBLAS (math API)	No	Yes
CUFFT (FFT API)	Yo	Yes

OpenCL and CUDA frameworks are really similar, and there is a way to run OpenCL on top of CUDA, that is good to prove that OpenCL is very portable. In Table I, one can see the difference between OpenCL and CUDA. Both are programmed in C, OpenCL can run on the CPU where CUDA cannot. The community and quantity of examples are bigger in favor of CUDA. From [6], they did a comparison of the speed between CUDA and OpenCL on the same software code. The results can been seen in Figure 2(a) and one can see that CUDA is faster. [7] did the same experiment and they got the same conclusion shown in Figure 2(b).



(a) CPU architecture

(b) GPU architecture

Fig. 3. Comparison of CPU and GPU [2]

IV. GPU architecture

The CPU and the GPU are very different, in Figure 3(a) one can see that a large portion of the chip is used for the cache and the control logic. The control logic include the branch prediction, instruction fetch, data path, register etc. There is a small portion of the chip dedicated to the arithmetic logic unit (ALU). On the other hand, the GPU has the emphasis on the ALUs. As seen on Figure 3(b), most of the chip is dedicated to the computation and the rest to the cache and control logic.

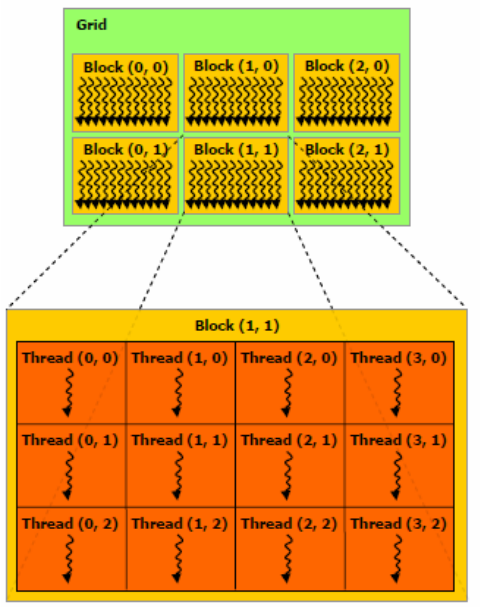


Fig. 4. GPU thread architecture [2]

Inside the GPU, the ALUs are split into blocks. There are

multiple microprocessors which are called blocks in the Figure 5. Each microprocessor has multiple processors or threads and each thread runs on its ALU. On a microprocessor, each thread runs the same instruction on its data thus the classification of SIMD. The different microprocessor can run different code.

V. CUDA MEMORY TYPES

There are four major types of memory on an NVIDIA GPU as seen on Figure 5. The first large memory is called the device memory, it is a DDR memory shared amongst every processors. That large memory is off-chip and it is very slow since there is no cache on the majority of models. It takes hundreds of clock cycles to access one memory address as said by [8]. The device memory contains the texture memory, but there is a special texture cache inside the chip. The texture memory is of the same speed as the device memory, but the cache is really fast and it is local to a multiprocessor. This read only cache is special in the sense that an interpolation is free over a texture, it takes the same time to access the pixel (0,1) than the pixel (0,1.5) where that pixel is an interpolation between pixel (0,1) and (0,2). A third cache is the constant cache, it is always 8KB and it serves the purpose of caching the constant memory on the device memory. The last important memory type is the shared memory, there is one shared memory per multiprocessor and it takes only one clock cycle to access. It was of a size of 16KB in every chip and this size has been increased on newest models to 64KB. It can serve as a cache for the global memory.

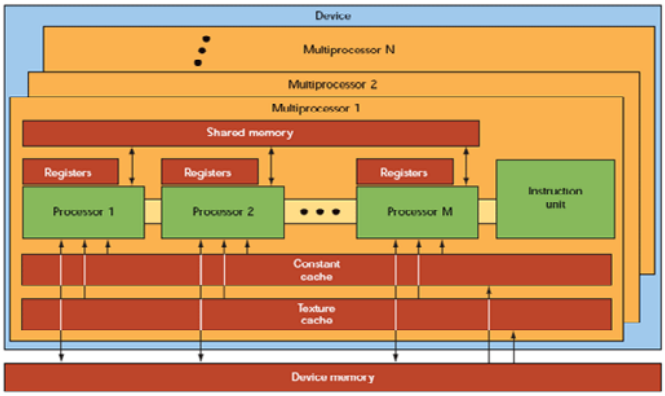


Fig. 5. GPU thread architecture [9]

There are two different types of shared memory in a GPU hardware. The device memory is shared among the multiprocessor, and it supports atomic operations.On the older device of a compute capability of less than 2, there is no cache and it is cache coherent. On the state of the art device with a compute capability of two or more, there is a L2 cache shared between the multiprocessors which is cache coherent. There is also a L1 cache private to the multiprocessors within its shared memory and this cache is not cache coherent, but there is the possibility to disable it.

The shared memory within a multiprocessor could be classified as a shared memory, but it not a truly shared memory, because it is local to a multiprocessor. It does support atomic operations. It often serves as a manual cache for the global memory, the programmer explicitly write to the shared memory the data he knows it will be used in a near future.

VI. MEMORY BANDWIDTH

The memory is often the restricting factor in a system, [4] wrote that using a single bus and a single memory module restricts the system scalability, because only one processor could access the memory at one time and it leads to serialization. The solution on the GPU is to use a very large memory bus width such as 384-bit in [10]. This can lead to 48 byte read in parallel in one operation. Those bytes have to be contiguous in the memory, the hardware scheduler sends the right byte to the right thread. In this case, it is important that the threads request data in this window of 48byte. If this requirement fail, multiple read requests will be made. The software is responsible to request data in this way.

For theoretical scenario with a GeForce 480 GTX card, the Memory Bandwidth can be as high as 177.4 GB/sec and the performance can be as high as 1.35 Tflops. This card can read 44 billion floats per second and can process 1350 billion floats per second which gives a ratio of around 30. This means that a minimum of 30 computation is required per memory access to overcome the cost of the access. Applications are CPU or memory bound depending on if they pass the threshold of 30 on that card. It is important to note that such speed is possible only between the device memory and the GPU. The communication between the CPU and the GPU is limited by the PCI Express bus and it is important to minimize that communication.

In practice, it is almost impossible to achieve the maximum throughput because of the memory alignment requirements. It is required that the threads request data from the memory continuously and starting at an aligned address multiple of 16 byte. The solution is the shared memory on the multiprocessor. Since it acts as a programmable cache, it is possible to order the reads from the global memory and do the computation on random data inside the faster shared memory.

VII. EXAMPLE

To validate the speedup achieved by using aligned over non-aligned addresses, two programs were made and ran on the GPU. In the Figure 6, the function main declares 2 array, and allocate memory on the GPU. It calls the kernel example with one thousand threads. In the kernel, the variable i is set to the thread number. The variable x is used because threads are disposed in a multidimensional array. The computation done by the kernel it is to shift an array one position to the right and duplicate the first element.

In the Figure 7, the main function is identical to the one in the Figure 6. The kernel function is different and this one declares a shared array of a thousand floats. The global array ”A” is transferred to the shared array ”S” where each thread reads a different address next to the first thread and the first thread read an aligned address. There is a call to syncthreads

```
__global__ void example(float* A, float* B)
{
    int i = threadIdx.x;
    if (i > 0)
        B[i] = S[i - 1];
}
int main()
{
    float* A, * B;
    cudaMalloc(&A, 1000);
    cudaMalloc(&B, 1000);
    example<<<1, 1000>>>(A, B);
}
```

Fig. 6. CUDA code without shared memory

```
__global__ void example(float* A, float* B)
{
    __shared__ float S[1000];
    int i = threadIdx.x;
    S[i] = A[i];
    __syncthreads();
    if (i > 0)
        B[i] = S[i - 1];
}
int main()
{
    float* A, * B;
    cudaMalloc(&A, 1000);
    cudaMalloc(&B, 1000);
    example<<<1, 1000>>>(A, B);
}
```

Fig. 7. CUDA code with shared memory

which act as a barrier where all treads needs to be at that point before continuing.

The program is executed on a GeForce 8800 GTS board with a compute capability of 1.0. In Table II, the comparison between Figure 6, Figure 7 is made and the program with shared memory is about 10 times faster than the one without. This fact can be explained by the fact that the Figure 6 did 298,424 memory read because it had to read each value with a different request since the values were not aligned. In the Figure 7, the reads are aligned since thread 0 reads the array at address A[0]. This changed the memory request to only doing 9,376 large coalesced reads. This explanation can be seen in the Figure 8 where it takes two 64B reads to read 256 values. In the case of an un-aligned access, it takes 32 32B read for the same number of values read.

VIII. Conclusion

The differences between CUDA and OpenCL are small, OpenCL is perfect for portable code which can be run on multiple platforms. CUDA is better for applications which need higher speed or need CUBLAS or CUFFT libraries as

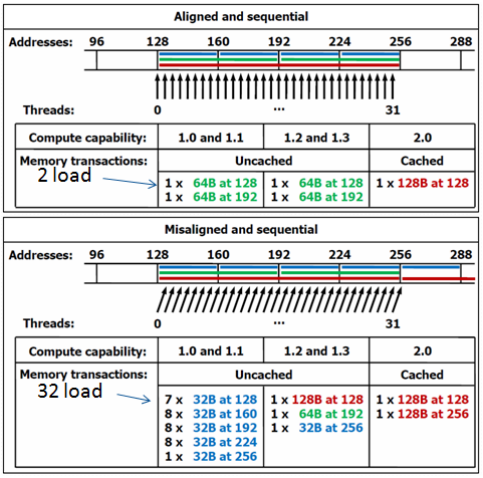


Fig. 8. Aligned and non-aligned memory read [2]

TABLE II
Comparison of shared memory and device memory in term of speed

	Time (ms)	Load coalesced	Load uncoalesced
Shared memory	325.6	9,376	0
Global memory	2965.98	0	298,424

seen in Table I. The memory limitations of massive shared memory systems can be overcome by using a larger bus and feeding multiple processors at the same time. The usage of caching in the newest GPU cards also helps to obtain higher speeds.

References

[1] K. Harzallah and K. Sevcik, "Hot spot analysis in large scale shared memory multiprocessors," in *Supercomputing '93. Proceedings*, 1993, pp. 895 – 905.

[2] N. Corporation, "NVIDIA CUDA C Programming Guide," pp. 1–173, 2010, http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf.

[3] A. Corporation, "GPU Computing: Past, Present and Future," pp. 1–42, 2010, http://developer.amd.com/gpu_assets/GPU%20Computing%20-%20Past%20Present%20and%20Future%20with%20ATI%20Stream%20Technology.pdf.

[4] S. Dandamudi, "Reducing run queue contention in shared memory multiprocessors," *Computer*, vol. 30, no. 3, pp. 82 –89, Mar. 1997.

[5] K. Group, "Introduction and Overview," pp. 1–20, 2010, http://www.khronos.org/developers/library/overview/opencl_overview.pdf.

[6] R. Weber, A. Gothandaraman, R. Hinde, and G. Peterson, "Comparing hardware accelerators in scientific applications: A case study," *Parallel and Distributed Systems, IEEE Transactions on*, 2010.

[7] F. H. Kamran Karimi, Neil G. Dickson, "A Performance Comparison of CUDA and OpenCL," pp. 1–10, 2010, http://arxiv.org/abs/1005.2581v2.

[8] Y. Hung, "CUDA Advanced Memory Usage and Op5miza5on," pp. 66–70, 2010, http://www.math.ntu.edu.tw/~wwang/mtxcomp2010/download/cuda_04_ykhung.pdf.

[9] B. Oster, "Programming The CUDA Architecture: A Look At GPU Computing," p. 1, 2009, http://electronicdesign.com/article/embedded/programming-the-cuda-architecture-a-look-at-gpu-co.aspx.

[10] NVIDIA, "GeForce GTX 480," p. 1, 2010, http://www.nvidia.com/object/product_geforce_gtx_480_us.html.