CHAPTER «ENGINEERING SCIENCES»

CUDA ARCHITECTURE ANALYSIS AS THE DRIVING FORCE OF PARALLEL CALCULATION ORGANIZATION

Andriy Dudnik[1]
Tetiana Domkiv[2]

DOI: https://doi.org/10.30525/978-9934-588-38-9-59

Abstract. Advancements in the Internet and cloud computing have led to a wealth of multimedia data, and processing of these data has become more complex and computationally intensive. With the advent of scalable low-cost GPUs with very high computing power, processing such big data has become less expensive and efficient. Rapid developments are also taking place in the field of programming languages and various programming and debugging tools, which simplify GPU programming. However, efficient and complete use of GPU resources remains a challenge. The purpose of this article is to provide a brief overview of the NVIDIA CUDA architecture and to consider the various programming and optimization strategies adopted by researchers to accelerate GPU computing. The purpose of this study is to provide researchers with knowledge about the various programming methods and optimizations in GPU programming and to motivate them to create highly efficient parallel algorithms by removing the maximum available graphics processor capabilities. Graphics Processing Unit (GPU) has entered the General Purpose Computing Domain (GPGPU) for over a decade now. The growth of frequencies of universal processors is stopped by physical limitations and high power consumption, and their performance is increasing more and more often due to the placement of several cores in one chip. Each core works separately from the others, following

[1] Doctor of Technical Sciences, Associate Professor,
Associate Professor, Department of Network and Internet Technologies,
Taras Shevchenko National University of Kyiv, Ukraine
[2] Graduate Student in the Department of Software Engineering,
National Aviation University, Ukraine

different instructions for different processes. Specialized vector capabilities (SSE2 and SSE3) for four-component (single precision floating-point calculations) and two-component (double precision) vectors appeared in universal processors due to the increased demand for graphic applications in the first place. That is why the use of GPU is more profitable for certain tasks, because they have been made for this. For example, in Nvidia video chips, the main unit is a multiprocessor with eight to ten cores and hundreds of ALUs in general, several thousand registers and a small amount of shared memory. In addition, the video card contains fast global memory with access to all multiprocessors, local memory in each multiprocessor, as well as special memory for constants. CPU cores are designed to execute a single stream of consecutive instructions with maximum performance, and GPUs are designed to quickly execute a large number of parallel threads of instructions. Universal processors are optimized to achieve high performance of a single instruction stream, processing both integers and floating point numbers. At the same time, access to memory is random.

1. Introduction

GPU differs from CPU also in terms of access to memory. In GPU, it is connected and easily predictable: if a texel of texture is read from memory, then after a while the time will come for neighboring texels. And the same thing when recording: the pixel is written to the framebuffer, and after a few ticks the one located next to it will be recorded. Therefore, the organization of memory is different from that used in CPU. And the video chip, unlike universal processors, just does not need a large cache, and for textures you need only a few (up to 128-256 in the current GPU) kilobytes.

And the work with memory in GPU and CPU itself is somewhat different. Not all central processors have built-in memory controllers, but in all GPUs there are usually several controllers. In addition, faster open memory is used, and as a result of the video chip, memory bandwidth is several times more available, which is also very important for parallel calculations that operate with huge data streams [1, p. 56; 5, p. 30].

About the differences in caching. Generic CPUs use cache memory to increase performance by reducing memory access latency, while GPUs use cache or shared memory to increase throughput. CPUs reduce memory access delays with large cache sizes, as well as code branch prediction.

These hardware parts occupy most of the chip area and consume a lot of energy. Video chips circumvent the problem of memory access delays by simultaneously executing thousands of threads: while one of the threads is waiting for data from memory, the video chip can perform calculations of another thread without waiting and delays [2, p. 146; 6, p. 315].

There are many differences in supporting multithreading. The CPU performs 1-2 calculation threads per processor core, and video chips can support up to 1024 threads per multiprocessor, of which there are several pieces in the chip. And if switching from one stream to another for a CPU costs hundreds of clock cycles, then the GPU switches several threads in one clock cycle [3, p. 76; 7, p. 480].

As a result of all the differences described above, the theoretical performance of video chips is significantly superior to the performance of CPUs.

2. GPU Architecture

The Fermi architecture is considered to be the first complete NVIDIA GPU (Patterson, 2009; Wittenbrink et al., 2011) because it provides virtually all the features required for the most demanding high-performance computing applications. This was the most significant step forward since the G80 architecture.

Figure 1a shows a high-level block diagram of a first Fermi chip. As shown in the figure, the Fermi architecture consists of 512 accelerator cores called CUDA cores. Each core contains a fully pipelined integer arithmetic and floating-point unit that perform one integer or floating-point operation for impact. Each CUDA core is organized into 16 streaming multiprocessors (SM), each with 32 CUDA cores.

The second layer, 768 KB, is used by all 16 Multiprocessors and 384-bit GDDR5 DRAM. The host interface shown in Figure 1a is used to connect the GPU to the processor via the PCI-Express bus.

The global Giga Thread scheduler then distributes the flow blocks across the multiprocessor flow schedulers. Figure 1b shows one SM consisting of 32 nuclei, each of which can execute a single floating-point command or integer instruction per clock cycle. Each SM also has 16 storage operations for memory operations, four special function modules, a 4K word register file, and 64K local SRAM split between cache and local memory [4, p. 185; 9, p. 646].

Special Function Blocks (SFUs) are used to perform instructions such as sine, cosine, square root, and interpolation. Threads are divided into groups of 32 parallel threads, which are called oblique. There are two strain planners and two command sending blocks, as shown in Figure 1b.

This allows you to perform two deformations and display them simultaneously. In addition, there are 64 KB of internal memory that can be configured as 48 KB of shared memory and 16 KB of L1 cache or vice versa (Patterson, 2009; Wittenbrink et al., 2011; Brodtkorb et al., 2013) (Figure 1) [8, p. 22; 10, p. 175].

3. GPU Programming Model

Programs executed by the GPU are called kernels. The execution of these cores is assigned by the programmer to one or more computing units (for example, in the figure there are two blocks: (0, 0) and (1, 0)).

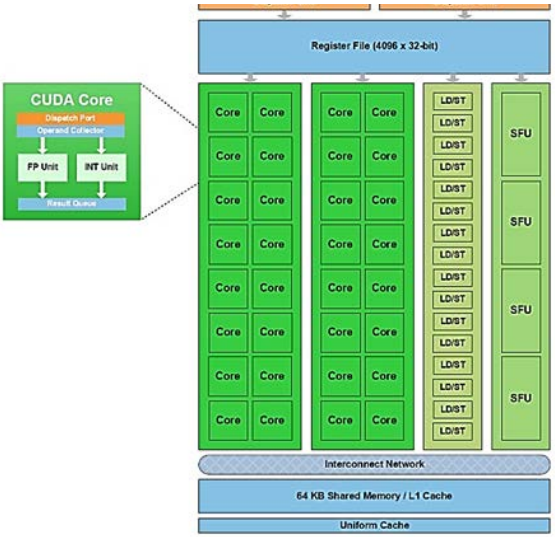
Blocks are distributed by hardware among the available SMs, and depending on the amount of resources required, each SM may be able to execute multiple blocks simultaneously due to multithreading.

Each block consists of a certain number of threads, for example, in the figure there are only two threads per block: stream (0, 0) and stream (1, 0). Threads are executed by hardware in small groups called strains (starting from the first GPUs with CUDA support, each strain consists of 32 threads). All flows within a given deformation share the same program counter; therefore, in the case of a conditional code or a disagreement between cycles, operations are serialized [11, p. 5].

Each thread is associated with a certain number of private registers and local memory. Local memory is also private for each thread and is used to store dynamically addressed arrays of registers or data that do not fit into registers. Despite its name, it is physically located in global memory, so it is relatively slow. Shared memory is common to each block and can be used to exchange data streams.

Internally, collective memory is divided into banks, and when several threads from the same deformation try to simultaneously access different cells of shared memory stored in the same bank, a conflict arises between banks and access is serialized. At the last level of the hierarchy, threads can only read from constants and texture memory, but they have read / write access to global memory. It is also possible to exchange stream data in

(a)



(b)

Figure 1. (a) Fermi Architecture consisting of 16 Streaming Multiprocessors (SMs); (b) Single SM (Patterson, 2009; Wittenbrink et al., 2011)

global memory, but only with a fraction of the speed compared to memory, especially if atomic instructions are used [12, p. 169].

Execution can be synchronized in each block through barriers, however, the only way to perform global synchronization between all blocks is to complete kernel execution and start a new one. To increase efficiency, global access to memory is done in small segments, not at the word level. To achieve optimal performance, threads must follow a series of merge rules (which depend on the hardware capabilities) to avoid generating multiple memory requests, thereby reducing the maximum achievable throughput. CUDA applications are written in «C for CUDA», which is a subset of C with extensions to perform functions in parallel. The programs are compiled using NVCC, the NVIDIA CUDA compiler. The CUDA program calls parallel kernels that run in parallel in a set of parallel threads. A programmer or compiler organizes these threads into thread blocks and grid blocks of threads. The kernel program is created on the GPU as a grid of parallel streaming blocks. Each thread in a thread block executes a kernel instance and has a thread ID in its thread block, program counter, registers, private memory for each thread, inputs, and output.

A flow unit is a collection of simultaneously running threads that can interact with each other through barrier synchronization and shared memory. Each stream block has a unique block ID in its grid.

A grid is an array of flow blocks that execute the same kernel, read input from global memory, write results to global memory, and synchronize between different kernel calls. For each stream in the CUDA concurrent programming model, there is a separate memory area for each stream that is used to populate registers, function calls, and C variables for the automatic array. Each flow block has a shared memory space for each block used to communicate between streams, share data, and share results in parallel algorithms. Stream block grids share results in global memory after global kernel synchronization. The CUDA grid concept is shown in Figure 2. The figure shows a two-dimensional hierarchy of blocks and flows, commonly used for image processing. The programmer determines the required number of flow blocks, and it is the GPU that decides which flow blocks will be executed on any SM. This abstraction is one of CUDA's greatest strengths because hardware can operate independently and efficiently. CUDA ensures that all flows in a block are executed on the same SM at the same time and that all kernel blocks complete execution before executing the next kernel (Figure 2) [13, p. 18].

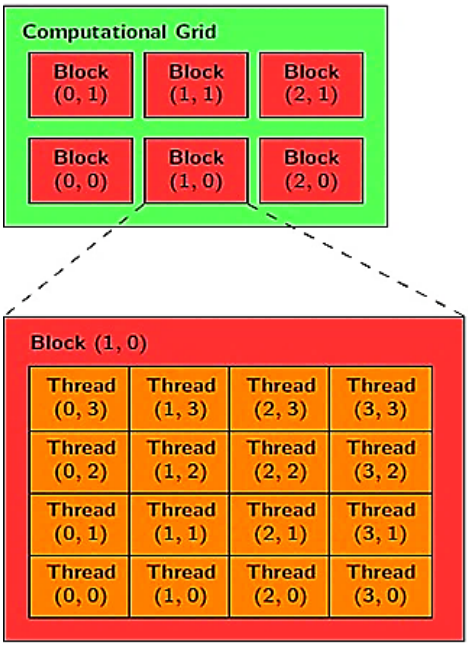


Figure 2. CUDA concept of a grid of blocks

4. CUDA memory model

The memory model in CUDA is distinguished by the possibility of byte-by-address addressing, support for both ga-tera and sca-tera. A fairly large number of registers is available for each stream processor, up to 1024 pieces. Access to them is very fast, you can store 32-bit integers or floating-point numbers in them. Each thread has access to the following types of memory [14, p. 76] (Figure 3):

Global memory is the largest amount of memory available for all multi-processors on a video chip, the size is from 256 megabytes to 1.5 gigabytes on current solutions (and up to 4 GB on Tesla).

It has high bandwidth, more than 100 gigabytes / s for top NVIDIA solutions, but very large delays of several hundred clock cycles. It is NOT cached, it supports generalized instructions LOAD and STORE, and the usual indications of memory. Local memory is a small amount of memory

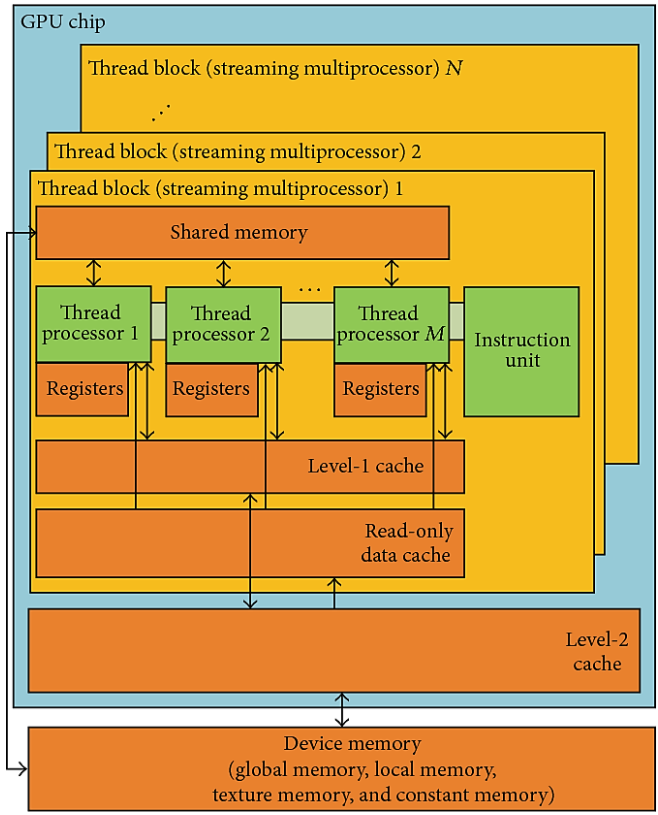


Figure 3. CUDA memory model

that only one stream processor has access to. It is relatively slow: the same as the global one.

Shared memory is a 16-kilobyte shared memory block for all stream processors in a multiprocessor. This memory is fast enough, the same as registers. It provides interaction of flows, is controlled directly by the developer and has low delays. The advantages of shared memory: the use of a first level cache managed by a programmer, reducing delays in accessing executive units (ALUs) to data, reducing the number of accesses to global memory [15, p. 640].

Constant memory is a 64 kilobyte memory area that is read-only by all multiprocessors. It caches 8 kilobytes per multiprocessor. It is slow enough: a delay of several hundred cycles in the absence of the necessary data in the cache.

Texture memory is a block readable by all multiprocessors. Data is sampled using the texture units of the video chip, so linear data interpolation options are provided at no additional cost. 8 kilobytes per multiprocessor are cached. It is slow as global: hundreds of latency cycles when there is no data in the cache.

Naturally, global, local, texture, and constant memory are physically the same memory, known as local video memory of a video card. Their differences in different caching algorithms and access models. The central processor can update and ask only external memory: global, constant and texture [16, p. 25].

From what has been written above, it is clear that CUDA provides for a special approach to development, which is not quite the same as that adopted in programs for CPU. You need to remember about different types of memory, that local and global memory are not cached and the delays in accessing it are much higher than in register memory, since it is physically located in separate microcircuits.

The hardware and software architecture presented by Nvidia for computing on CUDA video chips is well suited for solving a wide range of tasks with high parallelism. CUDA runs on a large number of NVIDIA video chips, and improves the GPU programming model, greatly simplifying it and adding a large number of features, such as resolution memory, the ability to synchronize streams, double-precision calculations, and integer operations [17, p. 230].

CUDA is a technology available to every software developer, it can be used by any programmer who knows the C language, but one of the drawbacks of CUDA is its binding to specific GPU models, because this architecture works only on video chips from this company, and not even at all, but starting from the series GeForce 8 and 9 and the corresponding Quadro and Tesla.

5. Advantages and Limitations

From a programmer's point of view, a graphics pipeline is a set of processing steps. The geometry block generates triangles, and the rasterization block generates pixels displayed on the monitor. The traditional GPGPU programming model is depicted in Figure 4:

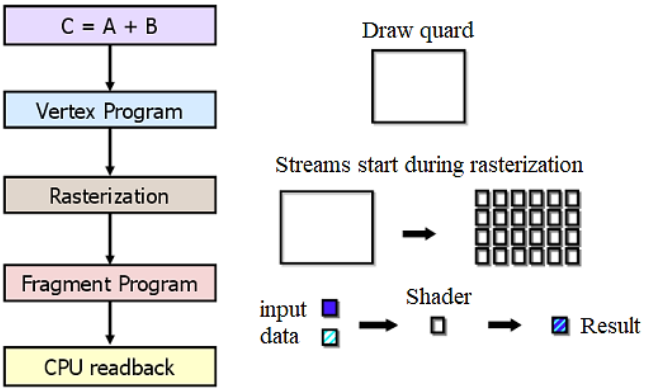


Figure 4. GPGPU programming model

To transfer the calculations to the GPU within the framework of such a model, a special approach is needed. Even the element-wise addition of two vectors will require drawing the figure on the screen or in an off-screen buffer. The figure is rasterized, the color of each pixel is calculated according to a given program (pixel shader). The program reads the input data from the textures for each pixel, adds them and writes them to the output buffer. All these numerous operations in a common programming language are written by a single operator. Therefore, the use of GPGPU for general-purpose computing has a limitation in the form of too much complexity for training developers. A very specific model of memory and execution is of particular note [18, p. 298].

The hardware-software architecture for computing on GPUs from Nvidia differs from previous GPGPU models in that it allows you to write programs for GPUs in the real C language with standard syntax, pointers, and the need for a minimum of extensions to access the computing resources of video chips. CUDA is independent of the graphics APIs and has some features designed specifically for general-purpose computing.

Advantages of CUDA over the traditional approach to GPGPU computing:

– the CUDA application programming interface is based on the standard C programming language with extensions, which simplifies the process of learning and implementing the CUDA architecture;

– CUDA provides access to 16K shared memory between threads on a multiprocessor, which can be used to organize a cache with a wide bandwidth, compared to texture samples;

– more efficient data transfer between system and video memory no need for graphical APIs with redundancy and overhead;

– linear addressing of memory, and gather and scatter, the ability to write to arbitrary addresses;

– hardware support for integer and bit operations.

The main limitations of CUDA:

– lack of recursion support for the functions performed;

– minimum block width of 32 threads;

– CUDA closed architecture owned by Nvidia.

The weaknesses of programming using the previous GPGPU methods are that these methods do not use vertex shader execution units in previous unified architectures, the data is stored in textures and displayed in an off-screen buffer, and multi-pass algorithms use pixel shader units. The limitations of GPGPU include: insufficient use of hardware capabilities, memory bandwidth limitations, lack of scatter operation (only gather), mandatory use of the graphics API [19, p. 415].

The main advantages of CUDA compared to previous GPGPU methods stem from the fact that this architecture is designed for the efficient use of non-graphical computing on the GPU and uses the C programming language, without requiring the transfer of algorithms to a form convenient for the concept of a graphics pipeline. CUDA offers a new way of computing on the GPU that does not use graphics APIs, offering random access to memory (scatter or gather). Such architecture is free from the disadvantages of GPGPU and uses all the execution units, and also expands the possibilities due to integer mathematics and bit shift operations.

In addition, CUDA opens up some hardware features not available from the graphics APIs, such as shared memory. This is a small memory (16 kilobytes per multiprocessor), to which thread blocks have access. It allows you to cache the most frequently used data and can provide a higher speed compared to using texture samples for this task. This, in turn, reduces the sensitivity to the bandwidth of parallel algorithms in many applications. For example, it is useful for linear algebra, fast Fourier transform, and image processing filters.

Convenient in CUDA and memory access. The program code in the graphic API displays data in the form of 32 single-precision floating-point values (RGBA values simultaneously in eight render targets) in predefined areas, and CUDA supports scatter recording, i.e., an unlimited number of records at any address. Such advantages make it possible to execute certain algorithms on the GPU that cannot be effectively implemented using GPGPU methods based on graphic APIs.

Also, graphic APIs without fail store data in textures, which requires preliminary packing of large arrays into textures, which complicates the algorithm and forces the use of special addressing. And CUDA allows you to read data at any address. Another advantage of CUDA is the optimized data exchange between the CPU and GPU. And for developers who want to access a low level (for example, when writing another programming language), CUDA offers the possibility of low-level assembly language programming.

6. Memory Coalescing

GPUs work best when streams are running in adjacent memory areas, that is, when memory access is combined. Memory access becomes serialized in the case of improper access, sparse memory access, or inconsistent memory access, and this will greatly affect performance. The following code example shows the operations of memory merge (Cornel Virtual Seminar, 2013; Pan-American Institute for Policy Studies, 2011).

*__global__ void program_mem(float *g){ float a=3.14;*
*int i= threadIdx.x; g[i]=a; g [i*2]=a;}.*

The most common method adapted to improper memory access is data reorganization. In (Wu et al., 2013) two algorithms for data reorganization are proposed. In the fill algorithm, the memory segments are supplemented with empty slots, which makes the segment access united.

However, since it reorganizes the flows with the data, this method can affect other links in the kernel. Sharing algorithm works by offsetting all non-merging access from global memory to shared memory, thereby reducing data duplication. In (Fauzia et al., 2015), a dynamic tool for analyzing unrelated memory accesses is proposed, and a structure that redistributes work between strains in deformation to avoid unrelated memory accesses.

7. Streams

It is also possible to improve concurrency in CUDA by running multiple cores in parallel using CUDA threads. By increasing the number of concurrent streams, a higher degree of concurrency can be achieved. Flow is an orderly queue of operations that involves running the kernel and transferring memory that will be executed by the GPU. Figure 4 shows n + 1 independent thread running in parallel.

Flows are useful when performing heterogeneous computations in which the CPU and GPU operate simultaneously. While the GPU is busy running the kernel and transferring memory, the CPU continues to perform its own operations and, when completed, synchronizes with the GPU to obtain results. Thus, along with the data parallelism achieved through streams and blocks, parallelism can also be achieved when programming on a GPU using streams.

8. Review on GPU Optimization Strategies

The first step to optimizing CUDA is to identify performance bottlenecks. Three main optimizations considered by default for a GPU are kernel optimization, memory optimization, and delay optimization (Cornel Virtual Seminar, 2013; Pan American Advanced Study Institute, 2011; Patterson, 2009; Wittenbrink et al ., 2011; CUDA Optimization Techniques 2010).

9. Kernel Optimization

The CUDA visual profiling tool can be used to identify bottlenecks in the CUDA core. The first step is to check with Visual Profiles, whether the kernel is bound to the bandwidth or to the computation. For cores with limited bandwidth, some optimizations should be considered to avoid global memory pooling, to use partitioned memory as a programmer-developed cache whenever possible, considering the use of array structure structures data.

For computed bound kernels that are less likely, some reduction in command strength can be made, for example, replacing a multiplication operation by a shift or addition operation, or by replacing a division operation by a reverse multiplication operation. Reducing the total number of transactions can also help.

Expensive conversion can be reduced by pre-calculating and storing values in temporary variables. In addition, kernels should be made as large as

possible so that the maximum amount of work can be performed with limited kernel calls, and kernel startup overhead can be reduced. Restrictions on shared memory and the use of registers in the kernel can also be attempted, thus increasing kernel occupancy. In (Lee et al., 2012) some effective kernel optimization strategies for neuroimaging algorithms are proposed.

They optimized compute-related kernels by reducing the use of registers and increasing data throughput by increasing workloads of threads. For memory-related cores, the data is reorganized into stand-alone structures, and a multi-pass approach is used for efficient optimization.

The kernel startup configuration can also be optimized by adjusting the number of blocks and the number of threads in each block so that the device is used to its maximum. There should be enough independent threads to hide the delays of instructions and memory. Avoiding thread divergence is also an important technique used to optimize the kernel.

10. Memory Optimization

When optimizing memory, there are two main factors to consider: memory access templates and the number of concurrent memory requests. Unlike processors that are designed in such a way that slightly irregular memory access templates do not affect performance, in GPUs the same access templates can greatly affect performance. Using this special GPU address space, such as memory of constants and textures, which is based on spatial and temporal localization, this limitation can be eliminated.

For very commonly used data, you can also use shared memory, which is a bit limited in size. Memory pooling is another method to consider when using global memory.

However, there are advantages to minimizing the use of global memory and maximizing the use of shared memory without conflicts of banking where possible. Another important consideration when optimizing memory is to reduce the overhead of transferring memory between the host and the device. Programmers should remember that maximum computing is performed on the device so that frequent memory transfers between CPU and GPU can be reduced.

Programmers should also remember that CUDA memory management operations, which are cudaMalloc and cudaFree, are expensive operations compared to their C malloc counters and free.

Figure 5. (a). Left: linear addressing with stride of one 32 bit word (no bank conflict), Middle : linear addressing with stride of two 32 bit word (2 way bank conflict), Right: linear addressing with stride of three 32 bit word (no bank conflict); (b). Left: conflict free access via random permutations, Middle: conflict free access since threads 3,4,6,7 and 9 access the same word within bank 5, Right: conflict free broadcast access (all threads access the same word) (Wen-Mei, 2011)

Therefore, to reduce the use of these operations, it is always advantageous to allocate memory once at the beginning of the operation and to continue to use memory for each kernel call. Threads, which are a sequence of operations performed in the issuing order, can be used to override memory operations, and the kernel is started to provide additional asynchrony, thereby improving performance (Figure 5).

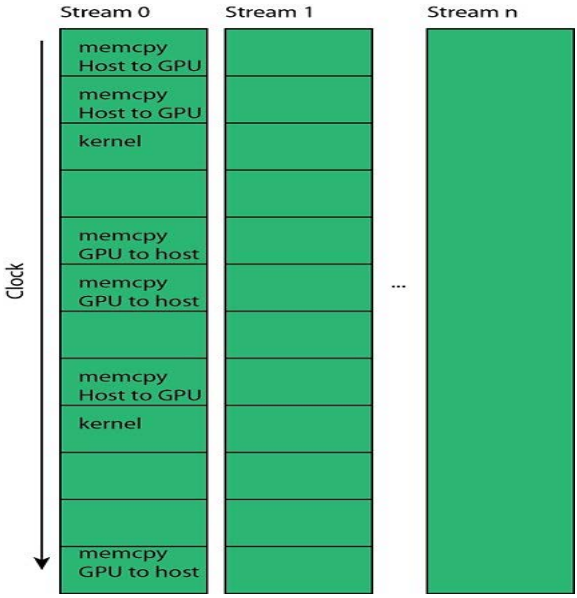


Figure 6. CUDA streams

In (Li et al., 2016), the problem of memory efficiency in deep clotting neural networks (CNNs) in GPUs is studied. The memory access patterns of various memory-related CNN levels are analyzed and effectively optimized to reduce off-chip memory access and core communication. The authors (Siegel et al., 2011) proposed a method of optimizing memory layout in their parallel version of the real Gravit application. They divide large memory structures into smaller substructures and consistently align them in global memory. The performance improvement was 50% compared to the non-optimized application layout.

11. Recent Trends in GPU Computing

The great success of general-purpose GPUs is due to the fact that they are very inexpensive. Two major GPU manufacturers, NVIDIA and AMD, are developing mass GPUs for the entertainment industry, and with the advent of versatile GPUs, they have added additional GPU functionality to meet this new trend. GPUs are now available in almost everything from cell phones to supercomputers. GPU parallelization is currently used in virtually all areas of scientific computing, ranging from image and video processing, remote sensing, machine learning, operations research, data mining, etc. on GPUs spanning different areas of research. Google searches for GPU calculations have resulted in many articles on GPU calculations in various fields of research. Recently, a great deal of work has been done on GPUs in the fields of remote sensing, medicine, molecular biology, and artificial intelligence. In (Ma et al., 2016), a multiple image model based on GPU-based parallel processing for remote sensing applications was proposed.

In (Ke, et al., 2016), a parallel computing environment for filtering and smoothing clouds for remote sensing images was proposed. In addition, a Kepler computing architecture-based graphics processor was used to detect LANDSAT-7 multi-image oil spills (Bhangale et al. 2017). Most of these algorithms have achieved significant acceleration compared to their CPU counterparts. GPU programming is widely used in medicine and molecular biology. Defining a one-part cryo-EM structure is a new trend that is transforming structural biology. GPUs are used to achieve significant acceleration in image classification and high-resolution refinement steps included in the cryo-EM structure determination workflow (Kimanius et al., 2016). Multiple sequence alignment is an extremely intense computational problem in computational molecular biology, where similar DNA sequences are aligned, and a prediction of molecular function is made.

In (Chen et al. 2017), a heterogeneous CPU / GPU platform is used to build such an alignment system. In (Sundfeld et al. 2017), the first GPU solution was proposed to solve the RNA structural alignment problem based on the Sankoff algorithm. In (Dubey et al., 2016) GPUs are used to predict the structure of the protein ab initio, which is a computational prediction of the structure of proteins on its primary amino acid sequence. This is a very expensive computational algorithm, and the authors have made significant gains in computational time when used on GPUs. Research in other areas is also

currently being carried out on GPUs to accelerate intensive computing. One such work in the field of physics is Feynman's integral estimation by a sectoral decomposition approach (FIESTA) (Smirnov, 2016), which is a new algorithm for improving optical performance in integral estimation. It aims to calculate with an increased number of sampling points to reduce uncertainty estimates.

GPU-based (Mantas et al., 2016) implements the realization of several simple numerical examples of equations in private derivatives to give an idea of how effectively such computationally complex mathematical problems can be solved on a GPU platform. (Jung and Bae, 2018) proposes to implement on the GPU a new linear equation solver that can be used to analyze mechanical systems. (Domínguez et al., 2016) proposes to implement on the GPU computationally expensive hydrodynamics of smoothed particles (SPH), a numerical method suitable for describing various complex flows with a free surface with large breaks.

A parallel implementation of the most widely used Elastic Net machine learning algorithms and Lasso algorithms is presented in (Zhou et al., 2015). (Wu et al. 2017) proposes a model called VLogGP to study the behavior of parallel applications communication and memory access models for heterogeneous CPU / GPU systems. In (Doulgerakis et al. 2017), the implementation of the restoration of computational parameters in diffusion optical tomography on GPU is presented. It is believed that this method is very applicable to systems with continuous wave and frequency domains, and it has achieved an almost 10x increase in speed when used on GPUs. Currently, NVIDIA GPUs are also at the forefront of accelerating many deep neural networks and artificial intelligence applications 10-20 times compared to the processor, reducing training time from weeks to days.

Next, we look at the GPU architectures developed by NVIDIA after the Fermi architecture and the new functionality added to each of them to improve the efficiency of GPU computing. Fermi's immediate successor, the Kepler architecture (NVIDIA Kepler GK110, 2013), has undergone major changes to the organization of a streaming multiprocessor (now called SMX) with four multiprocessor processors, each with 192 CUDA cores (1536 CUDA cores per chip). Also, the clock frequency was reduced from 1.5 GHz to 1 GHz. All of this was aimed at improving performance by increasing the number of cores running at a reduced clock speed. Compared to Fermi, L2 cache throughput has also been increased to 512KB for servicing applications that use a large amount of L2 cache.

Each SMX in Kepler is equipped with four deformation planners and eight command scheduling modules, allowing four deformations to be executed and output from 32 parallel streams simultaneously. The number of registers per stream was quadrupled to 255. The memory configuration is like Fermi, with additional partitioning for 64 KB total memory up to 32 KB each between shared memory and L1 cache. Kepler has also introduced a new feature called dynamic concurrency, which allows the GPU to generate new work for itself without the processor. The Maxwell architecture (NVIDIA Maxwell GM204 Architecture, 2016), the successor to Kepler, has provided a great leap in energy efficiency and productivity compared to previous generations. It also provides twice the performance per watt compared to Kepler products. The Maxwell architecture consists of 16SM (now called SMM) with 128 CUDA cores (2048 CUDA cores per chip) and 128 texture blocks. The memory bandwidth was increased from 192 GB/s Kepler to 224 GB/s, and the L2 cache size was increased to 2048 KB.

Each Maxwell SMM contains four strain planners capable of sending two stroke instructions. Compared to Kepler, the memory hierarchy has also changed due to the use of a dedicated memory of 96k, while the L1 cache is combined with the texture cache function. Each Maxwell CUDA core with more dedicated shared memory and large cache can provide about 1.4 times greater performance per kernel compared to Kepler. In addition to increasing power and computing performance, Maxwell also provides some other features, such as NVIDIA Voxel Global Illumination (VXGI), multi-frame sampling smoothing (MFAA), dynamic superresolution, conservative rasterization, multicast, and sparse text.

The Tesla P100 (NVIDIA Tesla P100 Technical Document, 2016), Maxwell's successor to the Pascal architecture, is the world's fastest GPU with 15.3 billion transistors. The most important feature of this GPU is the new high-speed NVLink interface, which provides data transfer between the GPU at up to 160 GB / s. It provides a single, single VAP for CPU and GPU memory, greatly simplifying GPU programming. Subtraction computing is an important feature in Pascal architecture, which allows commands to be subtracted with command-level granularity rather than granularity of flow blocks, as in earlier architectures. The GP100 SM ISA also provides new arithmetic operations that can perform FP16 operations on a single-core CUDA very quickly, and allows two FP16 values to be stored in 32-bit GP100 registers. This allows

you to quickly and efficiently train and deploy large neural networks with deep learning. The GP100 also provides improved atomic operations. A comparison between the three NVIDIA GPU architectures is shown in Table 1. The next-generation NVIDIA GPU should be launched in early 2018.

Table 1

Comparison between three Tesla GPU architectures (NVIDIA Parallel Forall, 2014)

GPU	Kepler GK110	Maxwell GM200	Pascal GP100
Compute capability	3.5	5.2	6.0
Threads per warp	32	32	32
Max warp per multiprocessor	64	64	64
Max threads per multiprocessor	2048	2048	2048
Max thread blocks/ multiprocessor	16	32	32
Max 32 bit registers per SM	65536	65536	65536
Max registers per block	65536	32768	65536
Max registers per thread	255	255	255
Max thread block size	1024	1024	1024
CUDA cores per SM	192	128	64
Number of SMs	8	16	60
Total CUDA cores	1536	2048	3840
Shared memory size/ SM configurations	16K/32K/48K	96K (dedicated)	64KB (dedicated)
L1 cache/SM	64KB	64KB (split)	24KB (dedicated)
L2 cache	512KB	2048KB	4096KB

12. Conclusion

Key to performance improvement in CUDA applications is to reduce the global memory latency by providing massive multithreading so that the cores have enough amount of work to perform.

Though porting any algorithms to the GPU is fairly easy fine tuning the programs to exploit the maximum capacity of the GPU is a major challenge.

The GPU code has to be largely optimized to attain the maximum efficiency of the GPU being used. In this paper we have reviewed with examples some common programming strategies adopted by the GPU programmers to take maximum advantage of the GPU programming.

We have also reviewed some common optimization techniques like kernel optimization, memory optimization, register optimization adopted by the GPU programmers to fine tune the GPU applications.

Our future work includes parallelization of some powerful image retrieval algorithms using the latest CUDA architecture and fine tune the application using the different programming and optimization strategies discussed. However, achieving peak achievable performance for a particular architecture rather than getting stuck in the local maximum of performance is a big challenge.

Performing calculations on GPU shows excellent results in algorithms that use parallel data processing. That is, when the same sequence of mathematical operations is applied to a large amount of data.

A flow unit is a collection of simultaneously running threads that can interact with each other through barrier synchronization and shared memory. Each stream block has a unique block ID in its grid.

Moreover, the best results are achieved if the ratio of the number of arithmetic instructions to the number of memory accesses is large enough.

These places less demands on execution control (flowcontrol), and the high density of mathematics and the large amount of data eliminate the need for large caches, like on CPU.

References:

1. Kvasnikov, V. P., Dudnik, A. S., Pysarchuk, O. O., & Domkiv, T. S. (2020). Using Cuda and Blockchain Technologies to Recover an Encrypted Pdf File Password. *Metrology and Instruments*, (6), 54–60. doi: https://doi.org/10.33955/2307-2180(6)2019.54-60

2. Rokochinskiy, A., Volk, P., Kuzmych, L., Turcheniuk, V., Volk, L.,& Dudnik, A. (2019, December). Mathematical model of meteorological software for systematic flood control in the carpathian region. In 2019 International Conference on Advanced Trends in Information Theory (ATIT), pp. 143–148. IEEE.

3. Skuratovskyj, R. V., Dudnyk, A. S., & Kvashuk, D. M. Vlastyvosti skruchenoyi kryvoyi edvarsa, podilnist yiyi tochky navpil i yix zastosuvannya v kryptografiyi. Problemy informatyzaciyi ta upravlinnya, 4(60), 71–78.

4. Dudnik, A. S., Cholishkina, O. G., & Lutsky, M. G. (2018). Analysis of the technology of blocking application between network internet technology for processing and storage of results of measurements. *Molodyi vchenyi*, 57(5), 183.

5. Bhangale, U., Durbha, S. S., King, R. L., Younan, N. H., & Vatsavai, R. (2017). High performance GPU computing based approaches for oil spill detection from multi-temporal remote sensing data. *Remote Sensing of Environment*, 202, 28–44.

6. Chen, X., Wang, C., Tang, S., Yu, C., & Zou, Q. (2017). CMSA: a heterogeneous CPU/GPU computing system for multiple similar RNA/DNA sequence alignment. BMC Bioinformatics, 18(1), 315.

7. Domínguez, J. M., Barreiro, A.J.C. Crespo, O. García-Feal, & Gómez-Gesteira, M. (2016). Parallel CPU/GPU Computing for smoothed particle hydrodynamics models. In Recent Advances in Fluid Dynamics with Environmental Applications, pp. 477–491. Springer International Publishing.

8. Doulgerakis, M., Eggebrecht, A., Wojtkiewicz, S., Culver, J., & Dehghani, H. (2017). Toward real-time diffuse optical tomography: accelerating light propagation modeling employing parallel computing on GPU and CPU. Journal of Biomedical Optics, 22(12), 125001.

9. Dubey, S. P., Kini, N. G., Kumar, M. S., & Balaji, S. (2016). Ab initio protein structure prediction using GPU computing. *Perspectives in Science*, 8, 645–647.

10. Jung, J., & Bae, D. (2018). Accelerating implicit integration in multi-body dynamics using GPU computing. *Multibody System Dynamics*, 42(2), 169–195.

11. Ke, J., Sowmya, A., Guo, Y., Bednarz, T., & Buckley, M. (2016, November). Efficient GPU computing framework of cloud filtering in remotely sensed image processing. In Digital Image Computing: Techniques and Applications (DICTA), 2016 International Conference on, pp. 1–8. IEEE.

12. Kim, K., Lee, S., Yoon, M. K., Koo, G., Ro, W. W., & Annavaram, M. (2016, March). Warped-preexecution: A GPU pre-execution approach for improving latency hiding. In High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on, pp. 163–175. IEEE.

13. Kimanius, D., Forsberg, B. O., Scheres, S. H., & Lindahl, E. (2016). Accelerated cryo-EM structure determination with parallelisation using GPUs in RELION-2. Elife, 5, e18722.

14. Kirk, D. B., & Wen-Mei, W. H. (2016). Programming massively parallel processors: a hands-on approach. Morgan Kaufmann. ISBN: 9780123914187, Elsevier.

15. Li, C., Yang, Y., Feng, M., Chakradhar, S., & Zhou, H. (2016, November). Optimizing memory efficiency for deep convolutional neural networks on GPUs. In High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for, pp. 633–644. IEEE.

16. Ma, Y., Chen, L., Liu, P., & Lu, K. (2016). Parallel programming templates for remote sensing image processing on GPU architectures: design and implementation. *Computing,* 98(1-2), 7–33.

17. Mantas, J. M., De la Asunción, M., & Castro, M. J. (2016). An introduction to GPU computing for numerical simulation. In Numerical Simulation in Physics and Engineering, pp. 219–251. Springer International Publishing.

18. Sundfeld, D., Havgaard, J. H., Gorodkin, J., & De Melo, A. C. (2017, March). CUDA-Sankoff: using GPU to accelerate the pairwise structural RNA alignment. In 2017 25th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 295–302. IEEE.

19. Wu, Y., Song, J., Ren, K., & Li, X. (2017). Research on Log GP based parallel computing model for CPU/GPU cluster. In Information Technology and Intelligent Transportation Systems, pp. 409–420. Springer International Publishing.