# AN INTRODUCTION TO PARALLEL COMPUTING

## CHAPTER 4 BANDWIDTH, LATENCY AND GPGPUS

BY FRED ANNEXSTEIN

# 4. BANDWIDTH, LATENCY AND GPGPUS

## CHAPTER OBJECTIVES

1. Understand the issues bandwidth and latency issues underlying the impracticality of the PRAM model.
2. Understand the meaning of Moore's Law and Little's Law and the impact of these laws on the design of GPUs
3. Understand the design of the CUDA software platform for GPGPUs.
4. Understand how to write, compile, and execute a CUDA program to sum vectors and generate images in parallel.

## FROM PRAM TO PRACTICAL PROGRAMMING

In the last chapter we had several examples of parallel algorithms written in the context of the ideal PRAM parallel programming model. However, in many practical applications one must maintain the locality of references and minimize communication overhead to achieve high performance on both CPUs and clusters of GPUs. The PRAM model is not rich enough to reflect modern GPUs that with careful programming can support much higher memory access rates - and FLOP rates - better than the fastest CPUs can.

To approach the peak throughput and memory performance, a computation must be structured around contiguous sequential memory accesses. This is good news for accelerating linear algebra operations such as adding two large vectors, where data is naturally accessed sequentially. However, algorithms that do a lot of reference pointer chasing will suffer from poor random-access performance, since random memory locations are likely non-local.

## LATENCY VERSUS BANDWIDTH

The peak throughput of a parallel computation is tied closely to two machine parameters known as l*atency and bandwidth*.

Latency is defined as the time needed to complete a task from the time the instruction was issued -– for example, the latency for a memory loads is the time between a read/write instruction and the ready to read point. Similarly, the latency of rendering a graphic image is that time between the render function call and the ready for display point.

Bandwidth is the rate of task throughput measured by the number of repeated tasks completed per unit time -– for example, the bandwidth of global memory transfers to localized shared memory is the completion time of the API call to transfer a data array and measured in bytes per second. Similarly, we use the bandwidth of graphic frames per second as a measure of the throughput of a rendering engine.

## MOORE'S LAW AND LITTLE'S LAW

Moore's law is the observation that the number of transistors in a dense integrated circuit (IC) doubles about every two years for same unit cost. Moore's law has held up well over 5 decades. For hardware designers, Moore's Law is a guide that favors optimizing bandwidth over latency when designing parallel computers. Designers place smaller, faster transistors in greater densities. However, this comes at a cost of communication over longer wires and potentially increases latency by a factor associated with speed of light. Deploying large sized queues will increasing the memory bandwidth at the cost of a slower latency.

Little’s law is an observation about the behavior of large size queues in a stable system. The law states that the long-term average number of tasks in a stable system is equal to the long-term average task arrival rate multiplied by the average time that a task spends in the system.

If the rate at which tasks enter the system is the same as the rate at which the task exit, then the system is said to be stable. By contrast, a task arrival rate exceeding an exit rate would represent an unstable system, where the number of tasks in the queue would gradually increase without bound.

Little's Law tells us that in a stable system the average number of tasks in the queue L, is the effective arrival rate λ, times the average task completion time W.

Little’s Law ~ $L = \lambda W$

Little’s Law shows the correlation between average latency and throughput. Hence, minimizing average latency is a optimization strategy.  We can optimize parallel computing throughput by *maximizing occupancy*. In particular, a programmer may improve their designs by attempting to maximize the number of outstanding memory references.

## THE DESIGN OF CUDA AND GPGPUS

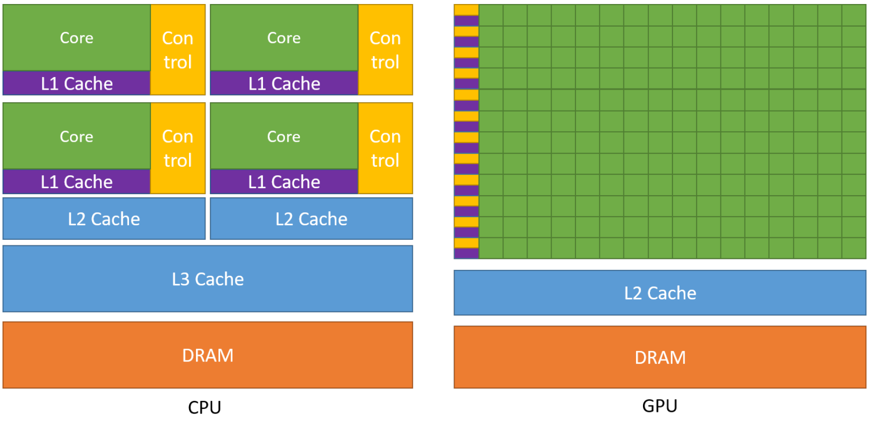
The goal of GPGPU is to apply GPUs for general computing purposes by providing a much higher instruction throughput and memory bandwidth than a CPU within a similar price and power envelope. Many applications leverage these higher capabilities to run code faster on the GPU than on multi-core CPUs.

GPU hardware optimizes for graphic memory bandwidth by employing techniques which in effect “hides” the average latency by bundling many memory tasks together. The difference in capabilities between GPUs and CPUs exists because they are designed with different goals in mind. CPUs are designed to excel at fast thread execution using shared memory. The GPU is designed to excel at executing thousands of slow threads in parallel. GPUs can amortize the slower single-thread performance to achieve greater throughput.

The GPU is specialized for highly parallel computations and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control.

The schematic below shows an example of the distribution of chip resources for a CPU versus a GPU. Note that the GPU design devotes more transistors to data processing, for example, as used processing floating-point arrays which does not depend on large data caches or complex flow control.

In practice, a parallel application has a mix of parallel parts and sequential parts, so large systems are designed with a mix of GPUs and CPUs in order to maximize overall performance.



## THE DESIGN OF CUDA

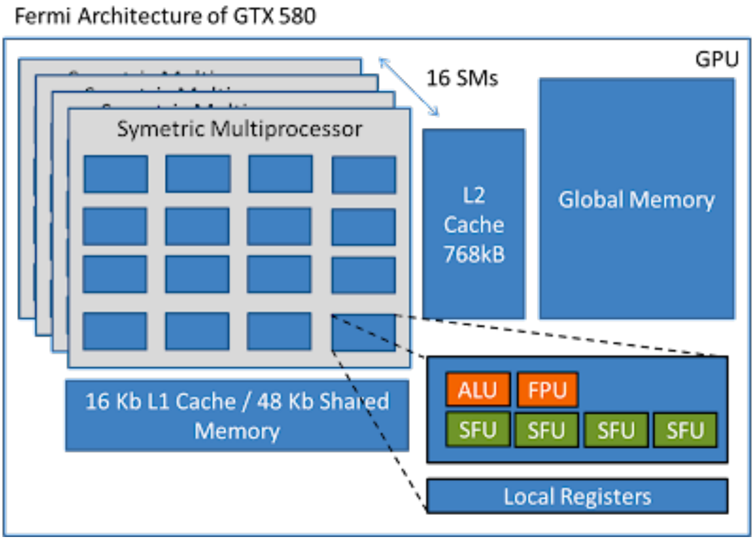
CUDA is a parallel computing platform and application programming interface (API) that allows software to use certain types of Nvidia GPUs for general purpose processing. CUDA is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels. CUDA is designed to work with programming languages such as C++ and Fortran. This accessibility makes it easier for specialists in parallel programming to use GPU resources which does not require advanced skills in graphics hardware programming.

Within the CUDA model there are three key abstractions - a hierarchy of thread groups, shared memories, and barrier synchronization. This model is exposed to the programmer via a minimal set of language extensions.

The CUDA model provides programmers with a rich set of options, such as  fine-grained data parallelism, thread parallelism, coarse-grained data parallelism, and task parallelism. CUDA programmers are guided to task decomposition and agglomeration by partitioning the computational problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block using shared memory.

This decomposition preserves language expressivity by allowing threads to cooperate when solving each sub-problem, and at the same time enables automatic scalability. Indeed, each block of threads can be scheduled on any of the available multiprocessors within a GPU, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of multiprocessors and only the runtime system needs to know the physical multiprocessor count.

The GPU is composed of hundreds of pipelined cores. The cores are grouped to



computation units, which NVidia calls *symmetric multiprocessors***.** Each computation unit is assigned a unit of work. CUDA supported GPU architectures can span a wide market range by simply scaling the number of symmetric multiprocessors and memory partitions.

## PROGRAMMING CUDA KERNELS

CUDA allows the programmer to define functions called *kernels* that are replicated and executed N times in parallel by N different *CUDA threads.* A kernel is defined using the `__global__` declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using an `<<<...>>>` *execution configuration* syntax. Each thread that executes the kernel is given a unique *thread ID* that is accessible within the kernel through built-in variables.

As an illustration, the following sample code, using the built-in variable `threadIdx`, adds two vectors *A* and *B* of size *N* and stores the result into vector *C. E*ach of the *N* threads that execute `VecAdd()` performs one pair-wise 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);
    ...
}
```

For programmer convenience, threadIdx is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional block of threads, respectively, and called a thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume.

The index of a thread and its thread ID relate to each other in a straightforward way: For a one-dimensional block, they are the same; for a two-dimensional block of size (Dx, Dy),the thread ID of a thread of index (x, y) is (x + y Dx); for a three-dimensional block of size (Dx, Dy, Dz), the thread ID of a thread of index (x, y, z) is (x + y Dx + z Dx Dy).

As an example, the following code adds two matrices A and B of size NxN and stores the result into matrix C.

```
// 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);
    ...
}
```

There is a limit to the number of threads that can run per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads.

However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks.

The number of thread blocks in a grid is usually dictated by the size of the data being processed, which typically exceeds the number of processors in the system.

The number of threads per block and the number of blocks per grid specified in the <<<...>>> syntax can be of type int or dim3. Two-dimensional blocks or grids can be specified as in the example above.

Each block within the grid can be identified by a one-dimensional, two-dimensional, or three-dimensional unique index accessible within the kernel through the built-in blockIdx variable. The dimension of the thread block is accessible within the kernel through the built-in blockDim variable.

## USING OSC ONDEMAND  FOR CUDA

We will now download, compile, and run some basic CUDA text and graphic codes. Our examples come from the book “Cuda By Example”. Go to Canvas to download the book and the source code as a zip file. You can then file transfer to OSC.

OnDemand provides a web-based File Explorer that can be used to upload and download files to your OSC home directory or project directory, and copy, delete, rename, and edit files.

Here is a tutorial video that gives a overview of OnDemand's file management client's capabilities and how to utilize them.

```
  1.Go to https://ondemand.osc.edu  and
    login into OSC using your
    new ucn****username and password.
  2. Using Files Dropdown at top of page
    go to the home directory and upload
    your zip file with source code.
  3. Back at Ondemand page, at the top
    you will seem the
    Interactive Apps dropdown menu.
  4. Select Owens or Pitzer Desktop –
    select 1 node for 1 hour and a vis
    node type.
  5. Find the Terminal application under
    System Tools
```

Here are the commands to use on the OSC terminal to unzip source code, compile, and run a program called **`enum_gpu.cu`**. In the output you will see details about the GPU configuration

```
$ unzip cuda_by_example.zip
$ module load cuda
$ cd cuda_by_example/chapter03
$ nvcc enum_gpu.cu
$ ./a.out

   --- General Information for device 0 ---
Name:  Tesla P100-PCIE-16GB
Compute capability:  6.0
Clock rate:  1328500
Device copy overlap:  Enabled
Kernel execution timeout :  Enabled
   --- Memory Information for device 0 ---
Total global mem:  17071800320
Total constant Mem:  65536
Max mem pitch:  2147483647
Texture Alignment:  512
   --- MP Information for device 0 ---
```

```
Multiprocessor count:  56
Shared mem per mp:  49152
Registers per mp:  65536
Threads in warp:  32
Max threads per block:  1024
Max thread dimensions:  (1024, 1024, 64)
Max grid dimensions:  (2147483647, 65535, 65535)
```

The output tells us various things about the GPU for which we will develop and run kernel or device codes. We can write code to run a maximum of 1024 threads on each of 56 multiprocessors (mp), each mp with a limited amount of 49K bytes of shared memory. So per thread there is 4K of fast shared memory to use in our code.