

Introduction to Parallel Computation with GPUs: Programming with JAX, NUMBA, THURST, and CUDA

Martin BUCHER
(bucher@sun.ac.za)
LECTURE 3

Course dates (4-5pm SAST):

Tu 18 Feb	Th 20 Feb
Tu 25 Feb	Th 27 Feb
Tu 4 March	Th 6 March
Tu 11 March	Th 13 March

Zoom link:

<https://u-paris.zoom.us/j/82740807191?pwd=8yAGlP3R6VKbsjNCIeRjJU3u0AQtU6.1>

Course webpage:

<http://www.sun.ac.za/english/data-science-and-computational-thinking/parallelcomputation>
<https://github.com/martinabucher/cudaCourse>

Lecture 3 Outline

1. Computer Accounts – Status
2. Assignment
3. Vector addition
4. Review: Timing CPU and GPU programs
5. Blocks, warps, and threads
6. Memory hierarchy on CPUs
7. Exercise for next time (and a few new references)

Course Accounts—Status

1. For all the people, who have written to me at buchersun.ac.za, accounts have been successfully created.
2. Some people have reported difficulties connecting from certain sites on campus but were able to connect successfully from a mobile phone hotspot or elsewhere on campus. IT has been contacted and the restrictions (eg from eduroam for the hostels) should be lifted by sometime today (Tuesday).
3. Please do not run any large or long jobs, as there are many of us sharing this single computer.
4. Information about connecting has been sent to all registrants by email. If you still need an account or encounter difficulties, please contact me.

Practical Exercise

In this lecture you learned how a very simple example works: how to add two vectors. The full source code can be found at:

<https://github.com/martinabucher/cudaCourse/blob/main/mbVecAdd.cu>

and you also need the header file and Makefile

<https://github.com/martinabucher/cudaCourse/blob/main/helper.cuh>

<https://github.com/martinabucher/cudaCourse/blob/main/Makefile>

You can download files on the command line as follows:

```
$ wget https://github.com/martinabucher/cudaCourse/blob/main/mbVecAdd.cu
```

With the vector addition example, you should have all the tools to do matrix multiplication in parallel on the GPU.

You may also want to look at the example:

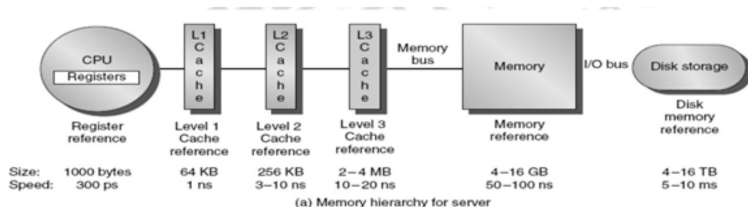
<https://github.com/martinabucher/cudaCourse/blob/main/cudaEnhancedHelloWorld.cu>

Matrix Multiplication Exercise

1. Generate two random matrices of adjustable size on the host.
2. Copy the data to the device.
3. Multiply the matrices together on the device.
4. Transfer the data back to the host.
5. Do the same computation on the host and compare the results.
6. Finally time the two computations for various matrix sizes and compare.

Warning: Your first implementation is going to be very slow. We will look at why.

Memory Hierachy (on a CPU machine)



For more details, see:

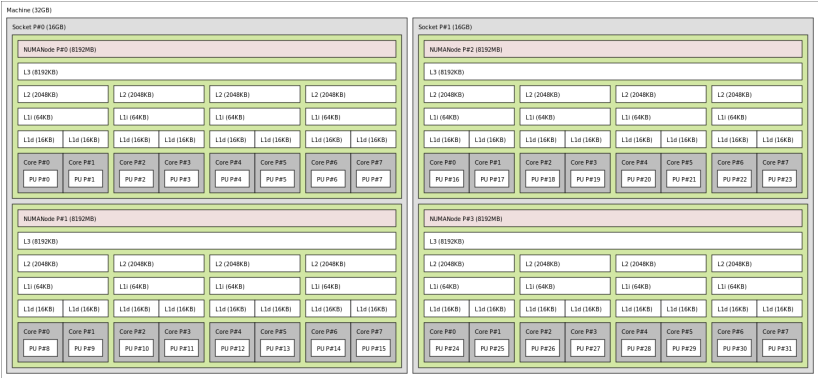
<https://www.cs.umd.edu/~meesh/411/CA-online/chapter/memory-hierarchy-design-basics/index.html>

Memory Hierachy (on a CPU machine)

- ▶ Fast memory is expensive and takes up more space. SRAM is fast. DRAM is slower but inexpensive. Its slowness can be mitigated by send larger chunks of data.
- ▶ The time required to fetch an item from memory is variable. If the item is already on a cache near the CPU, it can be very fast. If it needs to be fetched from main memory it can take a large number of cycles.
- ▶ There are two parameters characterizing the access to a particular level of memory: latency and bandwidth. Latency is the time between the request and arrival of the first data. Bandwidth is the max rate at which data can be transferred.
- ▶ For typical (but not all) CPU programs, the programmer need not worry about the hardware details of memory access. Typically memory access is much better than the worst case scenario because of locality both in space and time.
- ▶ In parallel programming memory is much more of an issue that must explicitly be considered. (More on this later.)

Memory Hierachy (on a CPU machine)

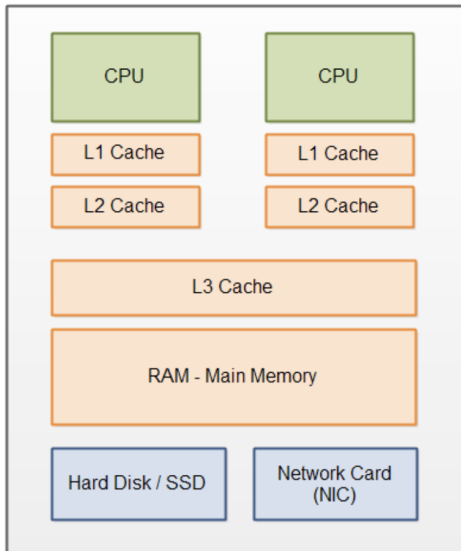
AMD Bulldozer Processor Schematic



From Wikipedia, Memory hierarchy

Memory Hierachy (on a CPU machine)

[From : <https://dev.to/larapulse/cpu-cache-basics-57ej>]



Cache/Memory Latencies: Some Numbers

[Adapted from : <https://dev.to/larapulse/cpu-cache-basics-57ej>]

L1 Cache Latency:

Typical latency ranges from 1 to 3 cycles, which is extremely fast.

L2 Cache Latency:

Typical latency ranges from 4 to 10 cycles, depending on CPU architecture.

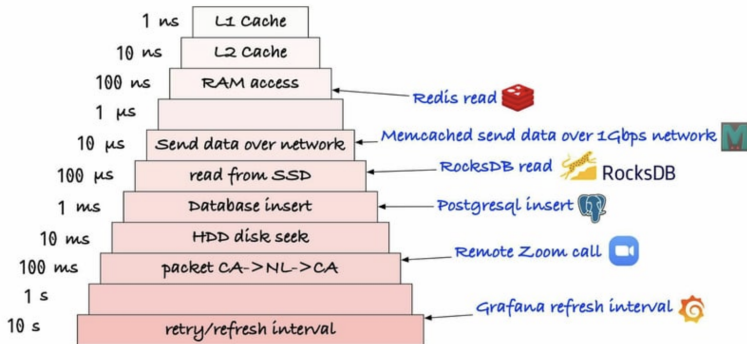
L3 Cache Latency:

Typical latency ranges from 10 to 40 cycles, depending on CPU and cache design.

RAM (Main Memory) Latency:

RAM latency can vary widely, but it typically ranges from 60 to 100 cycles
RAM access times are several orders of magnitude slower than L1 cache.

Latency with other storage media



Memory Latency on CPUs: Summary

- ▶ Memory access time can vary widely depending on which level of cache or memory the requested item is situated.
- ▶ Instruction execution can become stalled as a result of waiting for needed data to arrive and can slow program execution by several orders of magnitude.
- ▶ An optimized program will to the extent possible use data from the lower level caches and only rarely access higher level caches or main memory.
- ▶ prefetch operations can be used to bring data into a low-level (fast) cache anticipating its use later.
- ▶ Unfortunately, much of this sort of optimization is very hardware specific.

Compiler Specific Language Extensions (here GCC)

<https://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/Target-Builtins.html#Target-Builtins>

5.50 Built-in Functions Specific to Particular Target Machines

On some target machines, GCC supports many built-in functions specific to those machines. Generally these generate calls to specific machine instructions, but allow the compiler to schedule those calls.

Alpha Built-in Functions

ARM iWMMXt Built-in Functions

ARM NEON Intrinsics

Blackfin Built-in Functions

FR-V Built-in Functions

X86 Built-in Functions

MIPS DSP Built-in Functions

MIPS Paired-Single Support

MIPS Loongson Built-in Functions

Other MIPS Built-in Functions

picoChip Built-in Functions

PowerPC AltiVec Built-in Functions

SPARC VIS Built-in Functions

SPU Built-in Functions

Example: using prefetch

<https://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/Other-Builtins.html>

| Built-in Function: void __builtin_prefetch (const void *addr, ...)

```
for (i = 0; i < n; i++)
{
    a[i] = a[i] + b[i];
    __builtin_prefetch (&a[i+j], 1, 1);
    __builtin_prefetch (&b[i+j], 0, 1);
    /* ... */
}
```

Low-Level Libraries

A common solution is to use libraries such as the Basic Linear Algebra Subprogram library and the Intel Math Kernel Library. Interfaces in the form of function prototypes are standardized, but the implementation (eg in a *.so shared library) are tailored to the particular hardware available.

Subroutines may also sense the hardware on the machine and based on this information choose the routine actually used.

Please read:


https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms

and on the Intel Math Kernel Library

https://en.wikipedia.org/wiki/Math_Kernel_Library

GPU—Some numbers about the processor we are using

Ampere compute capability 8.9

	GPU features	Nvidia Tesla P100	Nvidia Tesla V100	Nvidia A100
	GPU codename	GP100	GV100	GA100
	GPU architecture	Pascal	Volta	Ampere
	Compute capability	6.0	7.0	8.0
	Threads / warp	32	32	32
	Max warps / SM	64	64	64
	Max threads / SM	2048	2048	2048
	Max thread blocks / SM	32	32	32
	Max 32-bit registers / SM	65536	65536	65536
	Max registers / block	65536	65536	65536
	Max registers / thread	255	255	255
	Max thread block size	1024	1024	1024
	FP32 cores / SM	64	64	64
	Ratio of SM registers to FP32 cores	1024	1024	1024
	Shared Memory Size / SM	64 KB	Configurable up to 96 KB	Configurable up to 164 KB

From the wikipedia Ampère architecture article

GPU-Structure of a Kernel Call

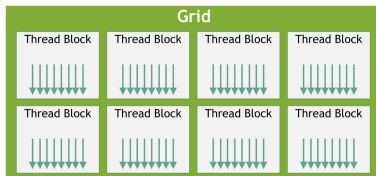


Figure 4: Grid of Thread Blocks

Recall the kernel function call syntax

```
kernelFunctionName<<<blocks,threadsPerBlock>>>  
    (function arguments);
```

- ▶ Blocks are assigned to Streaming Multiprocessors (SM). Blocks can include more threads than there are cores per SM.
- ▶ Threads are organized into warps of 32 threads
- ▶ Each warp has its own instruction counter.
- ▶ Warps can be swapped to hide memory latency
- ▶ Blocks that cannot fit on the GPU are held in waiting.

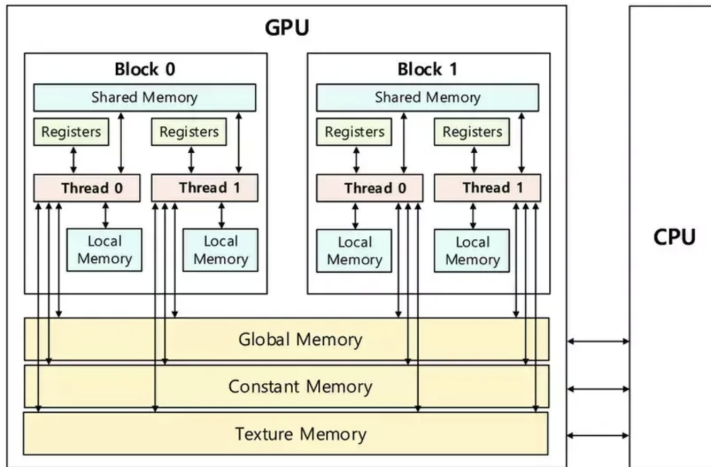
The same program can run on different types of hardware, having differing numbers of slots for blocks (SMs).



Figure 3: Automatic Scalability

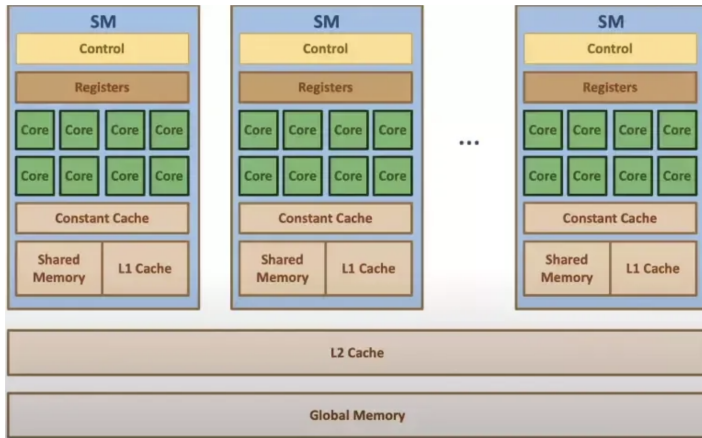
SIMT (Single Instruction Multiple Thread) vs SIMD (Single Instruction Multiple Data)
[as on a Cray with special vector instructions]

GPU Memory Layout



https://www.researchgate.net/figure/Structure-of-GPU-memory_fig2_365898686

GPU (a finer-grained representation of the same)



https://www.youtube.com/watch?v=ZQKMZIP3Fzg&ab_channel=OnurMutluLectures

Some comments (to be elaborated on later)

- ▶ The programs we looked at so far (i.e., vector addition and basic matrix multiplication) do not require synchronization: once the data has been loaded from the CPU to GPU, each thread (and block of threads) can do its work independently (and asynchronously).
- ▶ Only when the kernel call is completed is synchronization necessary, before the results are copied back to the CPU.

Asynchronous relation between CPU and GPU

- ▶ The default is for the kernel functions all to belong to the same stream.
- ▶ From the point of view of the CPU, the kernel function returns once the work has been dispatched to the GPU.
- ▶ Each stream creates a queue of this to be done, and unless there is a synchronization function called, the CPU and GPU will go about doing their work asynchronously.
- ▶ The GPU do not share a common clock. Nor do they share a common memory.
- ▶ Data must be explicitly shuttled back and forth between the CPU (host) and GPU (device), and this involves overheads.
- ▶ Error handling is more difficult. Because for a kernel function call to return successfully, it must merely be placed on the stream queue on the device without error.
- ▶ If an error occurs during the execution of the kernel function, this in general will not prevent the kernel function from returning with a successful error code.
- ▶ Rather an error code must be fetched later, as can be see in the example codes.
- ▶ Kernel functions if timed in the usual CPU way will appear to run almost instantaneously. This is why CUDA events must be created in order to time GPU kernel functions.

Here is the code to determine whether a kernel has executed successfully:

```
addVectorsKernel<<<numBlocks,threadsPerBlock>>>(c_d,a_d,b_d, sz);  
// ERROR CHECKING  
cudaError_t err = cudaGetLastError();  
if ( err != cudaSuccess ){  
    printf("CUDA Error: %s\n", cudaGetErrorString(err));  
    exit(-1);  
}
```

The following command creates a barrier on the stream forcing all the work in the stream queue to be completed before the function returns:

```
errCheck(cudaDeviceSynchronize());
```

which without the error wrapping is equivalent to

```
cudaDeviceSynchronize();
```

Persistence of memory on the device (aka the GPU)

- ▶ The global memory on the device persists across kernel calls. Caches are not addressed directly, so this is not an issue for understanding the outcome of a program.
- ▶ The shared memory (shared between the threads of a block) does not persist as new blocks are loaded and old blocks are completed.
- ▶ Synchronisation between threads of the same block is possible within a single kernel call. But there is no possibility to synchronize across blocks in CUDA.
- ▶ This is quite unlike other parallel computing architectures [e.g., MPI (Message Passing Interface commonly used on Beowolf clusters, where all threads (or processor) are at the same level.]
- ▶ This is a limitation of the GPU architecture, but to allow synchronisation accross blocks within a kernel call would require more expensive hardware.
- ▶ In practice, the way to synchronize accross blocks is to use more than one kernel call. The next kernel call in a stream does not start until all the blocks of the previous kernel call have completed.
- ▶ Kernel calls involve little overhead, so this is often not an inefficient solution.

A trivial example of inter-block synchronisation

<https://github.com/martinabucher/cudaCourse/blob/main/cudaEnhancedHelloWorld.cu>

```
__global__ void helloWorldKernel(){
    for(int i=0;i<10;i++){
        if (threadIdx.x==i )
            printf("Hello world from device, block= %d,
                thread=%d \n", blockIdx.x, threadIdx.x);
        __syncthreads();
    }
}
```

Note that this kernel was launched with a single block.

```
helloWorldKernel<<<1,10>>>>();
```

We would not have been able to synchronize with more than one block.

Some random additional comments

- ▶ Debugging parallel programs is more difficult than CPU programs.
- ▶ I do most of my CPU debugging with `printf(...)` statements, or even better `fprintf(stderr,...)` statements, which have the advantage that they are not buffered. The buffer is flushed from `stderr` to the screen straightaway. This is simple and unsophisticated, but for most purposes suffices.
- ▶ The latter does not work for GPU kernel functions. Only a subset of the C/C++ standard library is implemented in CUDA. `printf(...)` is implemented but `frprintf(stderr,...)` is not.
- ▶ One cannot communicate with the host during a kernel call. Rather the `printf` output is buffered, and only when the kernel returns without crashing is the message sent to `stdout`. This means that `printf` statements cannot be used to find where a kernel function crashed. Also `fflush(stdout);` is not implemented.
- ▶ We will in later lectures see how to debug with a cuda extension of GDB.

Timing CPU and GPU programs

The following is a good reference for C/C++ standard library functions:

<https://en.cppreference.com/w/>

<https://en.cppreference.com/w/cpp/chrono/c/clock>

Timing CPU programs

```
#include <time.h>

clock_t tStart=clock();
for(int i=0; i<DIM; i++)
    c_h2[i]=a_h[i]+b_h[i];
clock_t tEnd=clock();
float timeH_ms=(1000.*(tEnd-tStart))/((float) CLOCKS_PER_SEC );
```

Timing GPU kernels

```
float time_ms;
cudaEvent_t      startBis, stopBis;
errCheck(cudaEventCreate(&startBis));
errCheck(cudaEventCreate(&stopBis));

errCheck(cudaEventRecord(startBis, 0));
int sz=DIM;
int threadsPerBlock=1024;
int numBlocks=ceil(DIM/threadsPerBlock);
addVectorsKernel<<<numBlocks,threadsPerBlock>>>(c_d,a_d,b_d, sz);
    // ERROR CHECKING
cudaError_t err = cudaGetLastError();
if ( err != cudaSuccess ){
    printf("CUDA Error: %s\n", cudaGetErrorString(err));
    exit(-1);
}
errCheck(cudaEventRecord(stopBis, 0));
errCheck(cudaEventSynchronize(stopBis));
cudaEventElapsedTime(&time_ms, startBis, stopBis);
printf("Kernel function time (ms) = %g\n", time_ms)
```

Some more good references:

- ▶ CUDA C++ Programming Guide
<https://docs.nvidia.com/cuda/cuda-c-programming-guide/#programming-model>

Parts of this frequently updated document are very clear; other parts are verbose and opaques.

Please read the first four sections:

1. Introduction
2. Programming Model
3. Programming Interface
4. Hardware Implementation

PLEASE COPY THIS LINK TO THE COURSE WEBPAGE
WHERE COURSE MATERIALS WILL BE MADE AVAILABLE AS
WE GO ALONG

`http://www.sun.ac.za/english/data-science-and-computational-thinking/
parallelcomputation`

See also:

`https://github.com/martinabucher/cudaCourse`