Workshop: School of Data Science and Computation Thinking, Stellenbosch University

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

Martin BUCHER
(bucher@sun.ac.za)

LECTURE 2

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 2 Outline

1. Review of C dynamic memory allocation
2. C error checking
3. Vector addition
4. Compiling and running CUDA programs
5. Timing CPU and GPU programs
6. Blocks, warps, and threads
7. Memory hierarchy on CPUs
8. Exercise for next time (and a few new references)

Pointers, malloc, free and all that (I)

```
#include <stdlib.h>
#include <stdio.h>

int main(){
  float x=13.;  // x is a value, a real number represented using 4 bytes
  float *y;  // y is the address (in memory) of a number of the type float
  y=&x; //  now y points to x, whose value is 13. & is the address of operator
  printf("The value of x is %f.\n", x);
  printf("The value of pointed to by y  is %f.\n", *y);
  //  * = dereferencing operator. (value at the memory location pointed to)

  // printing pointers

  printf("The address of x is %p.\n", &x);
  printf("y points to the memory location %p.\n", y);
  return 0;
}

(base) apcmc2227:~ mabucher$ ./a.out
The value of x is 13.000000.
The value of pointed to by y  is 13.000000.
The address of x is 0x16fce7588.
y points to the memory location 0x16fce7588.
```

Pointers, malloc, free and all that (II)

```
#include <stdlib.h>
#include <stdio.h>

int main(){ //Pointer arithmetic

  float a[4]={0., 1., 2., 3.};
  // a is a pointer to a[0]
  //  *(a+2) is the same as a[2]
  printf("a[2] = %f \n", a[2] );
  printf("*(a+2) = %f \n", *(a+2));

  return 0;

}
```

```
(base) apcmc2227:cudaCourseSlides mabucher$ ./a.out
a[2] = 2.000000
*(a+2) = 2.000000
```

& is the "address of" operator. Maps variable indicating contents of a memory location $\rightarrow$ pointer (aka memory address).
* is the "dereferencing" operator, which maps pointer variable to its contents in memory.

Pointers, malloc, free and all that (III)

```
#include <stdlib.h>
#include <stdio.h>
int main(){  // Variable size arrays (length unknown as compile time)
   int len = 100;
   float *a= malloc(len*sizeof(float));
   // allocates number of bytes requested and returns pointer
   if ( a == NULL ){  // Error checking
     fprintf(stderr,"Memory allocation failed. Exiting.\n");
     exit(-1); // exit with error
   }
   for(int j=0;j<len;j++)
      *(a+j)=2.*j;
   for(int j=0;j<len;j++)
      printf("a[%d] = %f \n", j, a[j]);
   free(a);
   return 0;
}
(base) apcmc2227:cudaCourseSlides mabucher$ ./a.out
a[0] = 0.000000
a[1] = 2.000000
****************
a[99] = 198.000000
```

Java/Python philosophy: "Pointers are the root of all evil."
Memory leaks are nasty hard-to-find bugs.

Pointers, malloc, free and all that (IV)

The following is a C preprocessor directive that replaces every occurence of DIM with 10. The preprocessor is run before compilation.

```
#define DIM 10
```

The following

```
float a_h[DIM];
```

is (almost) equivalent to

```
a_h=(float*) malloc(DIM*sizeof(float));
```

The macro

```
sizeof( )
```

evaluates to the number of bytes occupied by the type specified by its argument, which can be an instrinsic type or a user defined type.

```
(base) apcmc2227:cudaCourseSlides mabucher$ cat struct_examp.c
#include <stdio.h>
#include <stdlib.h>

struct extFloatArray { double *data;
                       int n; };

typedef struct extFloatArray ExtFloatArray;

void ExtFloatArrayPrint(ExtFloatArray *myArray);

int main(){
  struct extFloatArray *myArray=malloc(sizeof(struct extFloatArray));
  ExtFloatArray *myArray2=malloc(sizeof(ExtFloatArray));
  myArray->n=10;
  myArray->data=malloc(myArray->n*sizeof(float));
  for(int i=0; i<10; i++)
    (myArray->data)[i]=(float) i;
  ExtFloatArrayPrint(myArray);
  return 0;
}

void ExtFloatArrayPrint(ExtFloatArray *myArray){
  for(int i=0;i<myArray->n;i++)
    printf("%d %e \n", i, (myArray->data)[i]);
}
```

Compiling and Running a C program (simplest example)

```
(base) apcmc2227:cudaCourseSlides mabucher$ gcc struct_examp.c
(base) apcmc2227:cudaCourseSlides mabucher$ ./a.out
0 0.000000e+00
1 1.000000e+00
2 2.000000e+00
3 3.000000e+00
4 4.000000e+00
5 5.000000e+00
6 6.000000e+00
7 7.000000e+00
8 8.000000e+00
9 9.000000e+00
```

Querying Device Properties (I)

The following is a C structure defined in one of the CUDA header files:

```
struct cudaDeviceProp {
                char name[256];
                cudaUUID_t uuid;
                size_t totalGlobalMem;
                size_t sharedMemPerBlock;
                int regsPerBlock;
                int warpSize;
                size_t memPitch;
                int maxThreadsPerBlock;
                int maxThreadsDim[3];
                int maxGridSize[3];
                int clockRate;
******

}
```

Everything you would ever want to know about the GPU running on your computer is described by a member variable of this structure.

Querying Device Properties (II)

https://github.com/martinabucher/cudaCourse/blob/main/cudaEnhancedHelloWorldBis

Snippet from the parallel hello world code where the device properties structure for device with number best_device is obtained.

```
cudaDeviceProp prop;
cudaError_t err;
// *****
err=cudaGetDeviceProperties (&prop, best_device);
if (err != cudaSuccess){
  printf("Error: unable to probe device %d.\n", best_device);
  exit(-1);}
// *****
int major=prop.major, minor=prop.minor; // get cuda version
// *****
```

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

```
__global__ void addVectorsKernel(float *c_d, float *a_d,float *b_d, int sz);
```

is the kernel function that adds two vectors of the type float.

The operation is

$$c = a + b$$

where a, b and c are vectors of the type float (a 4 byte floating point data type).

We distinguish between the copies on the host (aka the CPU) labelled with the suffix `_h` and on the device (aka the GPU) labelled with the suffix `_d`.

The function prototype

```
__global__ void addVectorsKernel(float *c_d, float *a_d,float *b_d, int sz);
```

declares a function to be run on the device (GPU). A function called from the host code that runs on the device is known as a kernel function. A device function is called from code running on the device and runs on the device.

The declaration keywords __global__, __device__, and __host__ are CUDA C/C++ language extensions.

The kernel function has three pointer arguments, pointing to the address in memory where the three vectors start. sz is the length of the vector.

A kernel function cannot return arguments and thus must be of the void type. Results must be placed in memory on the device and be moved back to the CPU explicitly if needed after the kernel call has completed. Here the first argument is the address of the return value, and the last three are for the input data.

Here is the code defining the global function:

```
__global__ void addVectorsKernel(float *c_d, float *a_d,float *b_d, int sz){
  // Here the built-in variables (which are structures of three integers x, y,
  // are used to compute a unique index for each thread.
  int index=blockDim.x*blockIdx.x+threadIdx.x;
  c_d[index]= a_d[index]+ b_d[index];
}
```

In normal C in a CPU code, we would look over index, as in the following kernel code where a CPU calculation is done for comparison:

```
  for(int i=0; i<DIM; i++)
    c_h2[i]=a_h[i]+b_h[i];
```

But in the kernel function each index is allotted its own thread, and the statement

```
int index=blockDim.x*blockIdx.x+threadIdx.x;
```

calculates the index corresponding to the thread.

Each block has blockDim.x threads, and a thread is identified by the ordered pair (blockIdx.x,threadIdx.x)

The kernel function call

```
addVectorsKernel<<<numBlocks,threadsPerBlock>>>(c_d,a_d,b_d, sz);
```

differs in syntax from an ordinary C function call, which might look like

```
addVectorsKernel(c_d,a_d,b_d, sz);
```

because the number of blocks and threads per block is also declared in the i language extension <<<numBlocks,threadsPerBlock>>>.

Each block is matched assigned to a streaming multiprocessor. A block contains many threads. Registers and the arithmetic processing unit reside on the thread or physically on the core. Instructions are dispatched from the streaming multiprocessor to its cores. Each of the cores of the streaming multiprocessor executes the same instruction at the same time.

What about warps?

There can be more blocks than there are streaming multiprocessors. When there a
more blocks than can be loaded onto the GPUs SMs,

ETC

On the previous slide we assumed that the data is already on the device, and the result of the calculation sits on the device. We now

```
float *a_h, *b_h, *c_h;
a_h=(float*) malloc(DIM*sizeof(float));
b_h=(float*) malloc(DIM*sizeof(float));
c_h=(float*) malloc(DIM*sizeof(float));
if ( a_h == NULL || b_h == NULL || c_h == NULL ){
  printf("Error: malloc failed. Exiting.\n");
  exit(-1);  //  'exit' is defined in stdlib.h --- the program
             //  terminates with return status -1, meaning unsuccessful.
}

// CODE TO INITIALIZE VECTORS

float *a_d, *b_d, *c_d;
errCheck(cudaMalloc((void**) &a_d,DIM*sizeof(float)));
errCheck(cudaMalloc((void**) &b_d,DIM*sizeof(float)));
errCheck(cudaMalloc((void**) &c_d,DIM*sizeof(float)));
```

which (without error checking) is like

```
cudaMalloc((void**) &a_d,DIM*sizeof(float));
cudaMalloc((void**) &b_d,DIM*sizeof(float));
cudaMalloc((void**) &c_d,DIM*sizeof(float));
```

Defensive programming

```
#include <stdlib.h>
#include <stdio.h>

int main(){
  cudaError_t err;
  cudaDeviceProp prop;
  int device=0; int count; int major, minor;

  err=cudaGetDeviceCount(&count);
  if (err != cudaSuccess ){
    printf("Error: cudaGetDeviceCount call failed.\n");
    exit(0);
  }
  if (count==0){
    printf("Error: No CUDA enabled devices found.\n");
    exit(0);
  }
  ****
```

cuda.... functions return an integer error code (of the type cudaError) and these should be checked and errors should be handled.

Defensive programming without clutter

The C/C++ preprocessor and inline function (for example placed in a header file)

```
#define errCheck(command)       errCheck2((command),#command,__FILE__,__LINE__)

inline void errCheck2(int command, const char *commandString, const char *file,
    int value=command;
    if ( value != cudaSuccess ){
      printf("%s  in file %s at line %d \n", commandString, file, line);
      printf("Error: program aborting.\n");
      exit(0);
    }
}
```

allows us to wrap a CUDA function so that error checking with semi-informative error messages result using the following syntax

```
errCheck(cudaMalloc((void**) &a_d,DIM*sizeof(float)));
```

Compiling with make

```
$cat Makefile

NVCC=/usr/local/cuda-12/bin/nvcc

SRC=$(shell ls *.cu)

EXECUTABLES=${SRC:.cu=}

all: ${EXECUTABLES}

NVCCFLAGS= -O3 -gencode arch=compute_80,code=sm_80 -std=c++11

%:%.cu helper.cuh
 $(NVCC) -o $@ $(NVCCFLAGS) $<

clean:
 rm ${EXECUTABLES}

and simply typing make will compile all the *.cu files in the directory

$make
```

make simply executes the instructions in the Makefile.

Some reflections on the speed of our GPU vector addition program

Why is it stupid to do vector addition using GPUs as we have done? This is a good simple teaching example, but is not what one would want to do in real world programming.

Do we expect our code to be faster than a CPU code? Why or why not?

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)
```

Passwordless login

```
(base) apcmc2227:cudaCourseSlides mabucher$ ssh sungpu
Last login: Tue Feb 18 12:25:56 2025 from 80.214.17.124
bucher@stoertebeker:~$ cat .ssh/authorized_keys

ecdsa-sha2-nistp256 AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNTYAAABBBLyjaGC

bucher@stoertebeker:~$ exit
logout
Connection to stoertebeker.sun.ac.za closed.
(base) apcmc2227:cudaCourseSlides mabucher$ cat ~/.ssh/config

Host sungpu
        HostName stoertebeker.sun.ac.za
        User bucher
        IdentityFile /Users/mbucher_1/.ssh/ronaldo_ecdsa

(base) apcmc2227:~ mabucher$ ls .ssh
authorized_keys
config
ronaldo_ecdsa
ronaldo_ecdsa.pub
(base) apcmc2227:~ mabucher$ cat .ssh/ronaldo_ecdsa.pub
ecdsa-sha2-nistp256 AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNTYAAABBBLyjaGC
```

Generating a Public/Private Key Pair

```
(base) apcmc2227:~ mabucher$ ssh-keygen -t ecdsa
Generating public/private ecdsa key pair.
Enter file in which to save the key (/Users/mbucher_1/.ssh/id_ecdsa): klaus
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in klaus
Your public key has been saved in klaus.pub
The key fingerprint is:
SHA256:zscescaUG68TLTVRW2OQrksTUrzogakwsmvQkw/LrPI mabucher@apcmc2227.local
The key's randomart image is:
+---[ECDSA 256]---+
|           .  o+o.|
|            o...o.|
|.o.=    o Xo+     |
|.o+ .    +.+      |
|+oE       o.      |
+----[SHA256]-----+
(base) apcmc2227:~ mabucher$ ls -tr1
klaus_key
klaus_key.pub
```

`klaus_key` contains the private key; `klaus_key.pub` contains the public key. These files are placed in your .ssh hidden subdirectory of the main directory.

Some more good references:

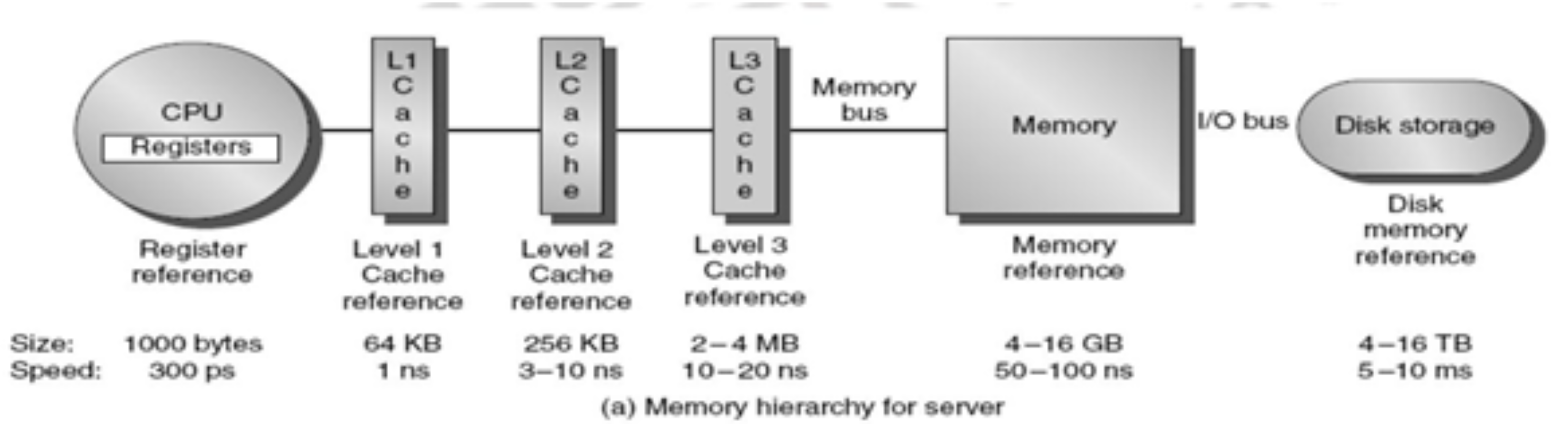
- CUDA C++ Programming Guide
  https://docs.nvidia.com/cuda/cuda-c-programming-guide/#programming-model

  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

Memory Hierachy (on a CPU machine)



(a) Memory hierarchy for server

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

memHierarchy.png

From Wikipedia, Memory hierarchy

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 paralell 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 to 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.

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`

Setting up course computer accounts

IT services wants us to use passwordless logins for the course accounts on:

`stoertebeker.sun.ac.za`

You need to generate an ssh (Secure shell) private/public key pair and send me the public key to

`bucher@sun.ac.za`

PLEASE DO NOT SEND ME THE PRIVATE KEY. When I have set up the account, I will send you a confirmation email with instructions on how to connect.
On a UNIX box a key is generated as follows: (See next slide)
I suggest you use an empty passphrase. Just press return when queried for a passphrase.

Generating an ssh key pair

```
(base) apcmc2227:~ mabucher$ ssh-keygen -t ecdsa
Generating public/private ecdsa key pair.
Enter file in which to save the key (/Users/mbucher_1/.ssh/id_ecdsa): wigbold
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in wigbold
Your public key has been saved in wigbold.pub
The key fingerprint is:
SHA256:NiDhLSKGH1GLy04KoTYhGwdoWtXV7rzrik8Kkfg59DQ mabucher@apcmc2227.local
The key's randomart image is:
+---[ECDSA 256]---+
|o .o+. ...       |
|oo.+ +.   .      |
|O+= = o  .       |
|=O.= + .  .      |
|+o* + E So       |
|o+.o = o .o      |
|. . = . .  .     |
|     o +  .      |
|      o.ooo.     |
+----[SHA256]-----+
(base) apcmc2227:~ mabucher$ ls -1 wig*
wigbold
wigbold.pub
```

Connecting from a Windows Box

Please see the webpage from the CHPC:

`https://wiki.chpc.ac.za/guide:connect`