A few notes on parallel programming with CUDA

Using parallel computing can significantly speed up execution and in many cases can be quite straightforward to implement. These notes focus on those simple cases with the aim of giving a very basic introduction to CUDA programming. For this reason, the notes stay away (or considerably simplify) more delicate questions.

The times that I have given below for the various code were obtained on my laptop with a GeForce GTX 1060. But as usual those times may vary from one run to another and will change on a different computer. Hence they should only be seen as a rough indication of performance.

1 Basic considerations

CUDA is a platform to issue commands to the GPU or Graphics Processor Unit. It can easily be integrated within several languages such as C++, Fortran, Python or C like here.

At first glance, coding in CUDA hence presents many similarities with OpenMP, namely

- Parallel calculations are performed by starting Threads and by specifying what each Thread should do based on its number identification.
- Since the Pascal architecture was introduced, memory is unified which means that you do not have to specify yourself the memory transfers. Locality and proper data dependencies are critical like on every parallel architecture.
- A CUDA code is written by adding specific commands to a classical C program. A specific compiler is needed; in Linux, we will use nvcc which is based on the C compiler, gcc. nvcc actually calls gcc so that it is possible to pass options to it. So for example, you could use OpenMP with CUDA with the command

```
nvcc -Xcompiler -fopenmp name_program
```
However GPU’s are very specific devices and you should remain aware that it is a separate processor with its own memory. The GPU architecture is also very different from the typical Intel processor that we have been discussing in class. One way of seeing it is as a grid of many cores, each capable of performing simple arithmetic operations (much better at simple precision for most card). This is well adapted to graphics operation and to many elementary calculations in linear algebra as we will see.

Some key elements to keep in mind are

- Because they are separate devices, the CPU and the GPU can run operations at the same time. Proper synchronization is hence important so that one device does not needlessly wait for the other (but still does so when necessary!).

- Even if you do not have to require them, memory transfers still have to occur: If you initialize an array on the CPU and then perform calculations on it on the GPU then the array will have to be moved to the GPU memory. Those transfers are slow and it is important to minimize them.

- Even on a laptop, we will use many more threads than we typically would with OpenMP. To mimic the physical architecture of the GPU, threads are also hierarchized and are grouped into so-called thread blocks.

2 Working through a simple SAXPY example

I am going to follow here, with all my thanks, the simple and clear introduction written by M. Harris at

https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/

The abbreviation SAXPY of “Single-Precision A.X Plus Y” refers to a very simple type single precision float operations where we multiply vector X by the scalar A and add to it the vector Y. It is often used in parallel computing examples (as we saw with OpenMP) because it has essentially no data dependency between threads.

2.1 The basic C code

Our goal is simply to add two vectors x and y created and initialized as arrays in C. Therefore our starting code is the following, cuda_example0.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

// function to add the elements of two arrays
void add(int n, float *x, float *y)
{
    for (int i = 0; i < n; i++)
        y[i] = x[i] + y[i];
}

int main(void)
{
    int N = 1<<20; // 1M elements

    float *x = (float*) malloc(N*sizeof(float));
    float *y = (float*) malloc(N*sizeof(float));

    // initialize x and y arrays on the host
    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }

    clock_t start;
    clock_t end;

    //Let us time the execution
    start=clock();
    // Run kernel on 1M elements on the CPU
    add(N, x, y);
    end=clock();

    // Check for errors (all values should be 3.0f)
    float maxError = 0.0f;
    for (int i = 0; i < N; i++)
maxError = fmax (maxError, fabs(y[i]-3.0f));
printf( "Max error: %f \n", maxError);
printf("Time for kernel=%f\n", (double) (end-start)/ CLOCKS_PER_SEC);

// Free memory
free(x);
free(y);

return 0;
}

Running this code on my laptop yielded a time of execution of around 0.01 s (with the usual variations).

2.2 A serial implementation in CUDA

Let us first modify the code so that it can be run on the GPU without trying to parallelize the calculations yet.

We need to modify the routine add to signal to the compiler that it can be run on the GPU. This is done by adding a global command so our routine becomes

```c
__global__
void add(int n, float *x, float *y) {
    for (int i = 0; i < n; i++)
        y[i] = x[i] + y[i];
}
```

The second change that we have to perform concerns the arrays x and y. We were using before malloc to assign CPU memory space for x and y. This memory may not be accessible from the GPU though so we also need to change that and use instead cudaMallocManaged which allocates unified memory that can be accessed from either CPU or GPU. The syntax is slightly different and we call

```c
float *x, *y;

cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));
```
As for malloc, we have to free the memory space when it is no longer used, with the command cudaFree.

This leads to the new code, cuda_examplewrong.cu

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

// function to add the elements of two arrays
// CUDA Kernel function to add the elements of two arrays on the GPU
__global__
void add(int n, float *x, float *y)
{
    for (int i = 0; i < n; i++)
        y[i] = x[i] + y[i];
}

int main(void)
{
    int N = 1<<20; // 1M elements

    // Allocate Unified Memory -- accessible from CPU or GPU
    float *x, *y;
    cudaMallocManaged(&x, N*sizeof(float));
    cudaMallocManaged(&y, N*sizeof(float));

    // initialize x and y arrays on the host
    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }
    clock_t start;
    clock_t end;
```
// Let us time the execution
start=clock();
// Run kernel on 1M elements on the GPU
add<<<1, 1>>>(N, x, y);
end=clock();

// Check for errors (all values should be 3.0f)
float maxError = 0.0f;
for (int i = 0; i < N; i++)
    maxError = fmax (maxError, fabs(y[i]-3.0f));
printf( "Max error: %f \n", maxError);

printf("Time for kernel=%f\n", (double) (end-start)/ CLOCKS_PER_SEC);

// Free memory
cudaFree(x);
cudaFree(y);

return 0;
}

Observe the way we call the kernel
add<<<1, 1>>>(N, x, y);
which calls 1 thread on 1 block.

2.3 Correcting the previous code

When running the previous code on my laptop, I obtain a time of execution of 0.00004 s, which seems already incredible with just one thread. But moreover there is an obvious problem as the max error is now 1...

What happened here? We made a classical synchronization error: We started the kernel on the GPU but after sending the command the CPU is pursuing its normal execution without waiting for the GPU to finish.

Thus when we start checking for errors, the execution is actually not yet done on the GPU and of course there are at least some elements $y[i]$ who still have their old value of 2. Hence the error is 1.

We also easily understand why our time is so good as well. We have not measured the time that the GPU took for the operations but just the time for the CPU to call the kernel...
We need to add the command

cudaDeviceSynchronize();

so that the CPU will wait till the GPU is done. And we need to do that
before measuring the end time or checking for errors...

The new correct code is

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

// function to add the elements of two arrays
// CUDA Kernel function to add the elements of two arrays on the GPU
__global__
void add(int n, float *x, float *y)
{
    for (int i = 0; i < n; i++)
        y[i] = x[i] + y[i];
}

int main(void)
{
    int N = 1<<20; // 1M elements

    // Allocate Unified Memory -- accessible from CPU or GPU
    float *x, *y;
    cudaMallocManaged(&x, N*sizeof(float));
    cudaMallocManaged(&y, N*sizeof(float));

    // initialize x and y arrays on the host
    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }
```
clock_t start;
clock_t end;

//Let us time the execution
start=clock();
// Run kernel on 1M elements on the GPU
add<<<1, 1>>>(N, x, y);

// Wait for GPU to finish before accessing on host
cudaDeviceSynchronize();

//calculate the time only now!
end=clock();

// Check for errors (all values should be 3.0f)
float maxError = 0.0f;
for (int i = 0; i < N; i++)
    maxError = fmax(maxError, fabs(y[i]-3.0f));
printf( "Max error: %f \n", maxError);

printf("Time for kernel=%f\n", (double) (end-start)/ CLOCKS_PER_SEC);

// Free memory
cudaFree(x);
cudaFree(y);

    return 0;
}

When running the new code, there is no error but the time now is around 0.15 s which is catastrophically bad but not so surprising: We are only using one thread and a lot of time is spent transferring memory.

Note that we can also use the profiler nvprof that gives many other useful informations. nvprof shows that the memory transfer is not much and that we should add many threads.

2.4 Using several threads

To improve execution speed, we need to start adding threads. For the moment let us keep one thread block. Unfortunately for us, CUDA does not
have a simple for command like OpenMP so we will have to do it ourselves.

We have seen that in order to do that we need the index of each thread and the total number of them. In CUDA we call

```c
int index = threadIdx.x;
int numthrds = blockDim.x;
```

to obtain in index, the index of the thread and in stride the total number of threads in the block.

We then have to decide how to parallelize. A first simple possibility is to use strides and make our kernel

```c
__global__
void add(int n, float *x, float *y)
{
    int index = threadIdx.x;
    int numthrds = blockDim.x;
    for (int i = index; i < n; i+=numthrds)
        y[i] = x[i] + y[i];
}
```

We can then call 1024 threads with for example

```
add<<<1, 1024>>>(N, x, y);
```

Please be careful that CUDA uses blocks of threads containing multiple of 32 threads so it is often better to avoid entering any random number here (even though it still compiles).

You will find the corresponding code in cuda_example2.cu.

With 1024 threads I’ve obtained a time of 0.0019 s. We are 5 times faster than with the first serial code and about 100 faster than with 1 thread.

Trying with other numbers of threads, I got the following times

<table>
<thead>
<tr>
<th>Threads number</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>0.01 s</td>
</tr>
<tr>
<td>64</td>
<td>0.006 s</td>
</tr>
<tr>
<td>128</td>
<td>0.004 s</td>
</tr>
<tr>
<td>256</td>
<td>0.0028 s</td>
</tr>
<tr>
<td>512</td>
<td>0.0022 s</td>
</tr>
<tr>
<td>1024</td>
<td>0.0019 s</td>
</tr>
<tr>
<td>2048</td>
<td>0.0019 s</td>
</tr>
</tbody>
</table>
As you can see things scale relatively well till 256 threads. After that the gains diminish to stop completely after 1024 threads. So we will now have to use several blocks as well.

Before doing that, we can try to perform another parallel approach by using blocks

```c
__global__
void add(int n, float *x, float *y)
{
    int index = threadIdx.x;
    int numthrds = blockDim.x;
    int m=n/numthrds;
    for (int i = (index*m); i < ((index+1)*m); i+=1)
        y[i] = x[i] + y[i];
}
```

This new code, cuda_example2mod.cu, produces the following times

<table>
<thead>
<tr>
<th>Threads number</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>0.011 s</td>
</tr>
<tr>
<td>64</td>
<td>0.006 s</td>
</tr>
<tr>
<td>128</td>
<td>0.0043 s</td>
</tr>
<tr>
<td>256</td>
<td>0.0045 s</td>
</tr>
<tr>
<td>512</td>
<td>0.0045 s</td>
</tr>
<tr>
<td>1024</td>
<td>0.0047 s</td>
</tr>
</tbody>
</table>

As you can see, we have almost the same times up to 128 threads but the performance is significantly worse after that...

2.5 Full parallelizing with blocks and threads

We now want to call a certain number of blocks and of threads per blocks with

```c
add<<<numblocks, blocksize>>>(N, x, y);
```

which calls numblocks of threads with blocksize threads per block. This makes use of the thread hierarchy.

We now need to update our implementation of the kernel, for instance with

```c
__global__
void add(int n, float *x, float *y)
```
{ int index = blockIdx.x*blockDim.x + threadIdx.x;
    int numthrds = blockDim.x*gridDim.x;
    for (int i = index; i < n; i += numthrds)
        y[i] = x[i] + y[i];
}

We see the new parameters blockDim.x which gives the number of threads in our block and blockIdx.x giving the index of the block. gridDim.x gives the total number of blocks.

We can easily build a global thread index thanks to that and calculate the total number of threads.

You may wonder how large should we try to have numthrds. And essentially the answer is as large as possible! So if we have $N$ computations to perform and 256 threads per block, we should try to take $N/256$ blocks. We’ll try to be a bit careful here just in case $N$ is not a multiple of 256 and actually code something like

```c
int blocksize = 256;
int numblocks = (N+blocksize-1) / blocksize;
```

The new code is called cuda_example3.cu and in that case it runs in about 0.0015 s (whether you take 256 or more threads per block has very little impact here). You were probably expecting a better number here and to put it in perspective it is time to revisit the question of memory transfer between CPU and GPU.

In general because the memory is unified, it is not easy to know how much time is spent on memory transfers. As a matter of fact when my laptop runs the code, it does not immediately transfer all of the arrays $x$ and $y$ but instead it tries to optimize it by transferring only what is needed. Thus even when running nvprof, it is not immediately possible to separate the time spent calculating from the time spent transferring...

Instead we are going to modify the code to force it to prefetch the arrays from the CPU before timing. In general this is a bad idea (it is better if some of the time spent on memory transfer is spent concurrently with the calculations) but it allows us to better profile our code.

I have put the corresponding code in cuda_example4.cu and it now clocks at a bit less than 0.001 s indicating that the memory transfer already takes a third of the time. In fact even this new time is misleading and we can now be much more precise by running nvprof instead of trying to clock things ourselves.
On my laptop nvprof indicates that GPU activities, that is the calculations, only took 80 µs. Prefetching was 700 µs on average but synchronizing took also 700 µs (to check that all calculations were done, gather the results...).

So in fact our parallel calculations are very fast, actually a bit faster than an old K80 server (and only 3 times as slow as a new P100)! What takes time is all the rest... This also shows the usefulness of more precise profiling tools in a case like this; clock measures well the CPU time but obviously fails here. Instead there exist specific command to time execution within the GPU: You should in particular check the CUDA event API.

2.6 A few more words on calling kernels
Here we have been using a 1d structure but in general we could work with 2d or 3d ones. Cuda has a special structure, dim3, precisely for this. So

\[
\text{dim 3 block(124,124,8), threadgrid(16,16,4)}
\]

creates a block with \(124 \times 124 \times 8\) threads, a corresponding threadgrid in each block and can be called by

\[
\text{add}<<<\text{block, threadgrid}>>>(N, x, y);
\]

There are many cases where it is easier to work with threads and blocks indexed in 2 or 3 dimensions.

The index of a thread in a block is now given by the 3 numbers ThreadIdx.x, ThreadIdx.y and ThreadIdx.z. Similarly the index and dimension of a block are obtained by blockIdx.x, blockIdx.y, blockIdx.z, blockDim.x, blockDim.y, blockDim.z.

In addition CUDA supports adaptive parallel computing in the sense that the number of running threads does not need to be fixed in advance. This is very simple as you can call a GPU kernel from inside a thread.

Though this goes beyond these simple notes, it is very convenient when doing mesh refinement in scientific computing. If each thread performs computations in a grid mesh, you can start the code with a very coarse grid. Whenever a thread evaluates that it is too coarse, it simply calls more...

This is also used for some hierarchical or tree-like type of meshes.

2.7 The CUBLAS library
CUDA offers a library of already implemented kernel functions for basic linear algebra operations. It is hence a good comparison point for us here.
You can check the following webpage for more details on CUBLAS:


Conveniently, CUBLAS includes a Saxpy routine which is just what we need... This routine is called by

\[ \text{cublasSaxpy(handle, N, &al, x, 1, y, 1);} \]

where \( x \) and \( y \) are arrays of size \( N \), composed of floats. \&al is a pointer on a float and handle is a variable of specific type which has to be initialized. This routine stores in \( y \) the result of \( al.x + y \).

To use it we would hence have the following cuda_example_cublas.cu code

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <iostream>
#include "cublas_v2.h"

int main(void)
{
    int N = 1<<20; // 1M elements

    cublasHandle_t handle; // CUBLAS context

    // Allocate Unified Memory -- accessible from CPU or GPU
    float *x, *y;

    cudaMallocManaged(&x, N*sizeof(float));

    cudaMallocManaged(&y, N*sizeof(float));

    // initialize x and y arrays on the host
    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }
```

13
cublasCreate (& handle );  // initialize CUBLAS context
clock_t start;
clock_t end;

//Let us time the execution
start=clock();

cublasSaxpy(handle,N,&al,x,1,y,1);  //does y=al*x+y, with x,y N vectors.

// Wait for GPU to finish before accessing on host
cudaDeviceSynchronize();

//calculate the time only now!
end=clock();

// Check for errors (all values should be 3.0f)
float maxError = 0.0f;
for (int i = 0; i < N; i++)
    maxError = fmax (maxError, fabs(y[i]-3.0f));
printf( "Max error: %f \n", maxError);
printf("Time for kernel=%f\n", (double) (end-start)/ CLOCKS_PER_SEC);

// Free memory
cudaFree(x);
cudaFree(y);
cublasDestroy(handle );  // destroy CUBLAS context

return 0;
}

The code has to be compiled with the right option
nvcc cuda_example_cublas.cu -o cuda_cublas -lcublas

If we time it we obtain 0.0010 s but as before this is very unprecise. Through nvprof, we can learn that the GPU time on the kernel is a bit more
than 80 $\mu$s. This is actually pretty much what we were able to obtain so we were not so bad...

### 3 An example with reduction

The previous example was a very simple case where each thread could operate completely independently. To go further, we need to consider a (still relatively elementary) case with some dependency.

We hence consider the following question: Given a vector $x$ of length $N$, calculate $\sum_{i=1}^{N} x_i$.

#### 3.1 Serial implementation and first considerations

The serial implementation is very simple and reads for example in `cuda_sum0.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

// function to add the elements of an arrays
void sumx(int n, float *x, float *result)
{
    *result=0;
    for (int i = 0; i < n; i++)
        *result += x[i];
}

int main(void)
{
    int N = 1<<20; // 1M elements
    float *x = (float*) malloc(N*sizeof(float));

    // initialize x and y arrays on the host
    for (int i = 0; i < N; i++)
        x[i] = 1.0f;
}
```

```c
15
```
float result; //will contain the sum

clock_t start;
clock_t end;

//Let us time the execution
start=clock();
// Run kernel on 1M elements on the CPU
sumx(N, x, &result);
end=clock();

// Check for errors (all values should be 3.0f)
float Error = (float) N-result;
printf("Max error: \%f \n", Error);

printf("Time for kernel=\%f\n", (double) (end-start)/ CLOCKS_PER_SEC);

// Free memory
free(x);

return 0;
}

When running it we find an execution time of 0.0035 s for the kernel which is consistently fast given the simplicity of the arithmetic operations.

On the other hand this is not so simple to parallelize anymore because as we have seen for OpenMP we need to do a reduction.

### 3.2 One block, many threads with tree-like reduction

Let us for the moment consider only one block (we will see why in the next subsection) and assume that \( N \) is a power of 2 (to make our code as straightforward as possible).

The basic idea as we have seen before in class is to perform a tree like reduction: At the lowest level \( M \) threads (with \( M \leq N/2 \)) will start to perform partial sums of \( x \). For example if \( M = N/2 \) then thread number \( i \) could sum \( x_{2i} + x_{2i+1} \).

Then at the next level we repeat the operation by using only \( M/2 \) threads to sum over the resulting partial sums, and so on at the next levels.

This leads to the code in cuda_sum1.cu reproduced here
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <iostream>

#define THREADS 256

// function to add the elements of an arrays
__global__
void sumx(int n, float *x, float *result)
{
    //We need a shared array to store all intermediary calculations
    __shared__ float intermedsum[THREADS];

    int index = threadIdx.x;
    int numthrds = blockDim.x;

    if (index == 0){
        *result=0.0f;
    }

    __syncthreads(); //we need to wait till all threads are done

    float localsum=0.0f; //initialize the local sum to 0

    int l=n/numthrds; // define the length of x that each thread has to solve

    for (int i = (index*l); i < ((index+1)*l); i++)
    {
        localsum += x[i];
    }

    intermedsum[index]=localsum; //store the local sum

    __syncthreads(); //we need to wait till all threads are done
//now we can start the tree
int level=2; //indicates at which level of the tree we are starting from bottom
while(level<(numthrds+1)) // stop when we’re higher than max number of threads
{
    int stride=level/2; //stride between a thread at this level and the next thread

    if ((index % level)==0) //check if the thread is still used at that level
    {
        intermedsum[index]+=intermedsum[index+stride];
    }
    level=2*level; //go to next level
    __syncthreads(); //but first again wait till all threads are done
}
//at the end of this loop only thread0 should have saved the final sum in intermedsum
if (index==0)
{
    *result=intermedsum[0]; //so we can finally save it!
}
}

int main(void)
{
    int N = 1<<20; // 1M elements

    float *x;
cudaMallocManaged(&x, N*sizeof(float));
    float *result;
cudaMallocManaged(&result,sizeof(float));

    // initialize x and y arrays on the host
    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
    }

    *result=0.0f;
clock_t start;
clock_t end;
// Prefetch the data to the GPU
int device = -1;
cudaGetDevice(&device);
cudaMemPrefetchAsync(x, N*sizeof(float), device, NULL);

// Let us time the execution
start=clock();
// Run kernel on 1M elements on the GPU
sumx<<<1,THREADS>>>(N, x,result);

cudaDeviceSynchronize();

end=clock();

printf("result: %f \n", *result);

// Check for errors (all values should be 3.0f)
float Error = (float) N-*result;
printf("Max error: %f \n", Error);

printf("Time for kernel=\f\n", (double) (end-start)/CLOCKS_PER_SEC);

// Free memory
cudaFree(x);
cudaFree(result);

return 0;
}

This code is as predicted one level of complication above what we have done before. We will discuss it extensively in class... But note a few key points

- We need a array, called intermedsum, to store all intermediary calculations and we need to make sure that it is shared so that all threads can see it: Hence the \_shared\_ command. As usual when defining an array in c the dimensions need to be fixed so we define the threads number at the beginning (we could also try to replace this by a dynamic allocation).

- The execution has to be synchronized at the right critical points. This
is done through the _syncthreads() instruction.

- There are two steps in the calculations: One with no data dependency where each thread sums \( x \) on its own block; this is also the most computationally costly. The next step adds the partial sums.

When run on my laptop the code announces a time of 0.001 s for the kernel. As before this could include many things but nvprof announces 950 \( \mu s \) for the kernel. This is 3 times as fast as the CPU implementation but not very impressive. Of course as we have seen before improving this time requires the use of several blocks.

3.3 Hybrid CPU-GPU implementation with many blocks

Our first impulse here could simply be to follow the steps of our SAXPY code: Simply call several blocks and modify our kernel to correctly define a global index of each thread.

Unfortunately that does not work... The problem lies with the GPU architecture. At the top are blocks, each block is actually composed of warps which are group of 32 threads.

Threads inside the same warp are actually automatically synchronized. Warps within the same block can be synchronized with _syncthreads(). But _syncthreads() does not synchronize threads belonging to different blocks. And as a matter of fact there are no simple commands to do anything like that as there are actually no easy way for the GPU to synchronize its blocks...

Therefore there are no easy ways to build up our partial sums on the tree to the level of blocks.

The simplest approach here is hence not to try! We are only going to do the tree up the level of one block. Since we have several blocks, we will have many partial sums. But we can store them in an array, return the array to the CPU and finish summing on the CPU.

Since most of the calculations have been performed on the GPU, this should still be reasonably fast. Here is the code in cuda_sum2.cu

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <iostream>
```
#define THREADS 256

// function to add the elements of an array on CPU
void sumcpu(int n, float *x, float *result)
{
    *result=0;
    for (int i = 0; i < n; i++)
        *result += x[i];
}

// function to add the elements of an array
__global__
void sumx(int n, int numblocks, float *x, float *partial)
{
    // We need a shared array to store all intermediary calculations
    __shared__ float intermedsum[THREADS];

    int index = threadIdx.x;
    int blockindex = blockIdx.x;
    int indexglobal= blockIdx.x*blockDim.x + threadIdx.x; // global index of the thread

    int numthrds = blockDim.x;

    if (index == 0){
        partial[blockindex]=0.0f;
    }

    __syncthreads(); // we need to wait till all threads are done

    float localsum=0.0f; // initialize the local sum to 0

    int l=n/(numblocks*numthrds); // define the length of x that each thread has to solve

    for (int i = (indexglobal*l); i < ((indexglobal+1)*l); i++)
    {
        localsum += x[i];
    }
intermedsum[index]=localsum; //store the local sum
__syncthreads(); //we need to wait till all threads are done

//now we can start the tree
int level=2; //indicates at which level of the tree we are starting from bottom
while(level<(numthrds+1)) // stop when we're higher than max number of threads
{
    int stride=level/2; //stride between a thread at this level and the next thread

    if ((index % level)==0) //check if the thread is still used at that level
    {
        intermedsum[index]+=intermedsum[index+stride];
    }
    level=2*level; //go to next level
    __syncthreads(); //but first again wait till all threads are done
}
//at the end of this loop only thread0 should have saved the final sum in intermedsum[0]
if (index==0)
{
    partial[blockindex]=intermedsum[0]; //so we can finally save it!
}

int main(void)
{
    int N = 1<<20; // 1M elements

    float *x;
    cudaMallocManaged(&x, N*sizeof(float));
    float *result;
    cudaMallocManaged(&result,sizeof(float));
    *result=0.0f;

    int M=1<<8; //size of x handled by each thread;
    int numblocks = (N+M*THREADS-1) / (M*THREADS); //define the number of blocks
float *partial;
cudaMallocManaged(&partial,numblocks*sizeof(float)); //allocates the array of partial sums

// initialize x array on the host
for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
}

clock_t start;
clock_t end;

// Prefetch the data to the GPU
int device = -1;
cudaGetDevice(&device);
cudaMemPrefetchAsync(x, N*sizeof(float), device, NULL);
cudaMemPrefetchAsync(partial, numblocks*sizeof(float), device, NULL);

//Let us time the execution
start=clock();

// Run kernel on 1M elements on the GPU
sumx<<<numblocks,THREADS>>>(N,numblocks, x,partial);
cudaDeviceSynchronize();

end=clock();

clock_t start2;
clock_t end2;

//now we need to add the partial sums
start2=clock();
sumcpu(numblocks,partial,result);
end2=clock();

printf( "result: %f \n", partial[0]);
// Check for errors (all values should be 3.0f)
float Error = (float) N-*result;
printf( "Max error: %f \n", Error);

printf("Time for kernel=%f\n", (double) (end-start)/ CLOCKS_PER_SEC);
printf("Time for second kernel=%f\n", (double) (end2-start2)/ CLOCKS_PER_SEC);

// Free memory
cudaFree(x);
cudaFree(partial);
cudaFree(result);

return 0;
}

This new version follows precisely the previous one for the kernel. Indeed the only difference is that we keep track of which block we are in and the result is store at the appropriate index in the variable partial. Because we have more threads, the length of x dealt by each thread is now smaller (so the definition of the variable l is updated as well).

In fact there is one thing that may seem strange at first look: We still have the same definition for our variable intermedsum. This is because shared variables are actually only shared within the threads of the same blocks. This is again connected to the architecture of the GPU, even though in this case it makes our life a bit easier...

When run, this code announces a time of 0.0006 s for the GPU kernel and 0.00002 for the second CPU kernel. Of course the CPU kernel does not have much work to do so it is naturally very fast.

We should be legitimately skeptical of the first time by now given the large number of blocks. The use of nvprof reveals that the actual time length of the calculations is in fact around 100 µs so that we have gained a factor 10.

You are welcome to play with the code and in particular to try to optimize the length of x summed per thread which is essentially stored in the variable M. The optimum seems to be around $M = 2^4 = 16$ with around 60 µs for the execution.

There are actually many ways in which we have not been optimal here but there go beyond the very limited scope of these notes...
3.4 How to run everything on the GPU

We do not expect to gain much time by doing the final calculations on the GPU but there can still be very compelling reasons to do so.

For example, if we want to use our code as a subroutine, then necessarily having a memory exchange between CPU and GPU can be seriously detrimental. It could in particular force such a transfer several times in a portion of code which could otherwise have been run entirely on the GPU...

But there still remains the problem of how to synchronize between blocks. The simplest way to do that is by actually calling a kernel. When calling a kernel after a kernel, this automatically forces the synchronization within the GPU.

So we can simply change the function sumcpu to a GPU kernel and then call each kernel in turn. But of course we may do much better by parallelizing as well this new kernel sumfinal. We will not try to do anything complicated here as there is not anyway much to gain. We can instead simply re-use our first kernel where we were only using one block.

You will find the following code in cuda_sum3.cu which does not contain any surprises. When profiling it with $M = 16$, I found on my laptop a time of 60 $\mu$s for the first kernel and 7 $\mu$s for the second.

You might wonder here if we are forced to call each kernel one after the other within main.

As we mentioned before, it is in fact possible to call a kernel from within a kernel but only on some GPU (with compute_35 architecture or above).

Instead it is possible to define a function with __device__. It can then be called by another function on the GPU (either global or another device). You cannot assign to it any parameters, that is you cannot change the number of blocks or number of threads per block.

3.5 How does it compare?

The final question is how poorly we’ve done in this more complicated example. We are going to use CUBLAS again which has a function sum (summing the absolute value of the components but since all $x_i$ are positive this does not matter).

This is called as before and return a time of around 35 $\mu$s or roughly half of what we were able to obtain.