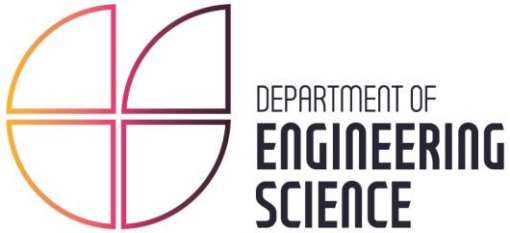


An Introduction to HPC and Scientific Computing

Lecture eight: An introduction to the CUDA programming language.

Karel Adámek

Oxford e-Research Centre,
Department of Engineering Science

Overview

In this lecture we will learn about:

- What is CUDA?
- The basics of writing a GPU accelerated code.
- Parallelisation in CUDA.
- How to map data in CUDA to GPU hardware.

What is CUDA

- Is abbreviation for “Compute Unified Device Architecture”
- Language based on C/C++
- Exposes GPU parallel capabilities for general purpose computing, while retaining high performance
- Enables heterogeneous computations and programming
- API allows to manage GPUs and their memory (allocate, copy, free)
- Fortran is supported as well

General purpose computing on GPU
or GPGPU means that GPU can be used for computations as well as for graphics. First introduced into GPU to improve image quality.

Heterogeneous computing
refers to systems which use different kinds of processor architectures to achieve higher performance or energy efficiency. Different architectures are better suited for different tasks.

API
is “application programming interface” and in this case it is a piece of software which simplifies communication with the GPU.

Where is CUDA

What do you need?

NVIDIA driver
CUDA toolkit

What do you get?

Driver is a low-level software that controls the graphics card

Toolkit:

- nvcc CUDA compiler
- Nsight IDE plugin for Eclipse or Visual Studio
- Nsight Compute/Systems - profiling tools
- cuda-gdb - debugging tools
- several libraries (cuFFT, cuBLAS, cuRAND, ...)

SDK:

- lots of demonstration examples
- some error-checking utilities
- not officially supported by NVIDIA
- almost no documentation

Hardware perspective

A typical node (computer) configuration is to have a CPU communicating with one or more GPUs through PCIe.

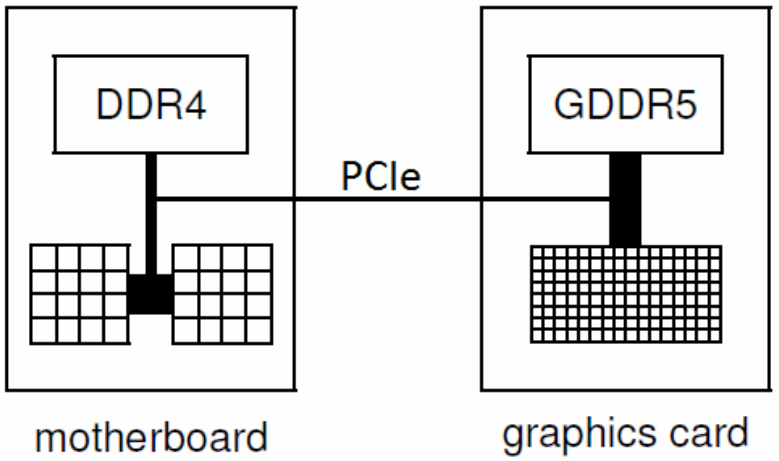
In general

CPU (host) has lower bandwidth to memory, less cores but much more memory available.

GPU (device) has higher bandwidth, many more cores, less memory.

CPU is optimized to minimize latency, ideal for command and control and some computations.

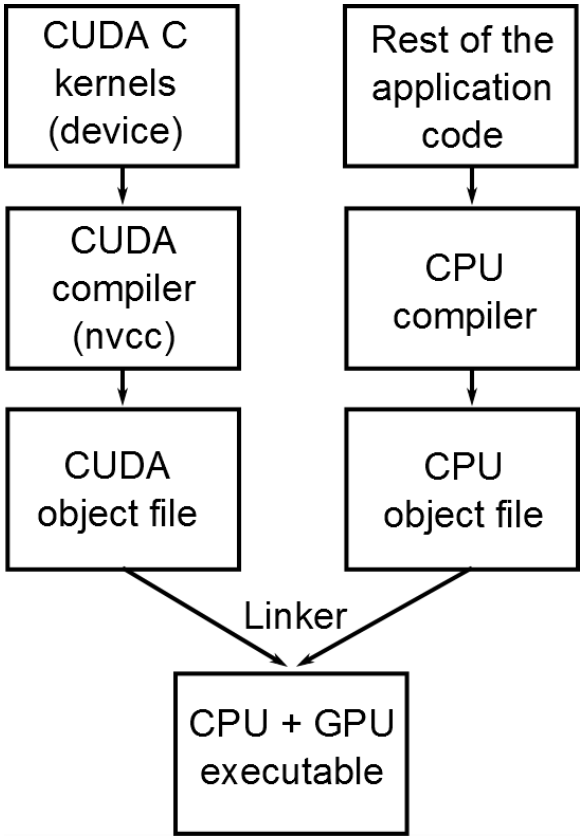
GPU is optimized for throughput, ideal for data parallel computations.



PCIe
is a bus connecting the host and device, it has the potential to be a problem:
CPU bandwidth: ~300GB/s
GPU bandwidth: ~1500GB/s (A100)
PCIe bandwidth: ~32GB/s (4.0)

How this affects us?

- Code is divided into host part and device part (both can be in one file)
- Any device code must be in the form of a CUDA kernel
- Host code constitutes the rest of the application
- host<->device interactions is user-managed
- Additional bottleneck (PCIe, …) must be considered



‘Hello world’ in CUDA

CPU code for fancy ‘Hello world’

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

void helloworld(void) {
            printf("Hello world!\n");
}

int main(void) {

            helloworld();

            return (0);
}
```

GPU code for ‘Hello world’

```
#include <stdio.h>
#include <stdlib.h>
// we have to include few more things
#include <cuda.h>
#include <cuda_runtime.h>
#include <cuda_runtime_api.h>

__global__ void helloworld_GPU(void){
  printf("Hello world!\n");
}

int main(void) {

  // run CUDA kernel
  helloworld_GPU<<<1,1>>>();

  return (0);
}
```

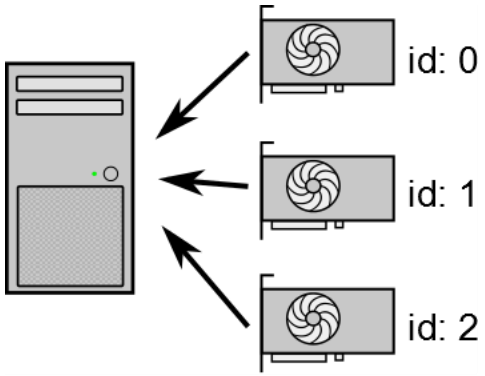
GPU does things bit differently:

- The function **`helloworld_GPU`** is special function which is run on GPU and it is called **kernel**.
- Before we launch any GPU kernel it needs to be configured. This is done by **`<<< ... >>>`** syntax.
- Run CUDA kernel

‘Hello world’ in CUDA – device initialisation

Before we can run our code we have to initialize the device first!

A node (or PC) can have more then one GPU. That is why each GPU has its id. In our code we want to use GPU with id 0.



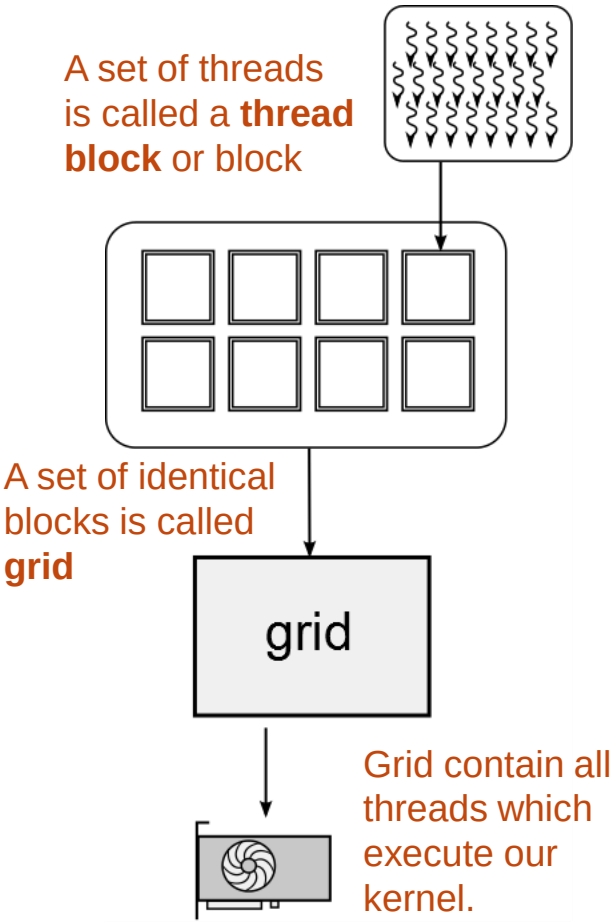
How to get information about what GPU are available? Run **`nvidia-smi`** on linux.

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 375.26                 Driver V
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M.
|===============================+======================+======================|
|   0  GeForce GTX 1080    Off  | 0000:01:00.0     On |                  N/A |
| 21%   32C    P0    37W / 180W |      0MiB /  8112MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  TITAN X (Pascal)    Off  | 0000:06:00.0     Off |                  N/A |
|  0%   27C    P0    50W / 250W |      0MiB / 12189MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
```

GPU code for ‘Hello world’

```
#include <stdio.h>
#include <stdlib.h>
// we have to include few more things
#include <cuda.h>
#include <cuda_runtime.h>
#include <cuda_runtime_api.h>

__global__ void helloworld_GPU(void){
  printf("Hello world!\n");
}

int main(void) {
  // initiate GPU
  int deviceid = 0; // using GPU with id 0
  int devCount;
  // gets number of GPU available
  cudaGetDeviceCount(&devCount);
  // check if we have enough GPUs
  if(deviceid<devCount) {
    // tell CUDA that we want to use GPU 0
    cudaSetDevice(deviceid);
  }
  else return(1);

  // run CUDA kernel
  helloworld_GPU<<<1,1>>>();

  return (0);
}
```

Expressing parallelism in CUDA



Kernel is a set of instructions executed by a single thread, i,.e. something like a serial code for a CPU written in C.

To achieve parallelism and good performance on GPUs we have to launch a lot of these threads. Each thread executes the same kernel.

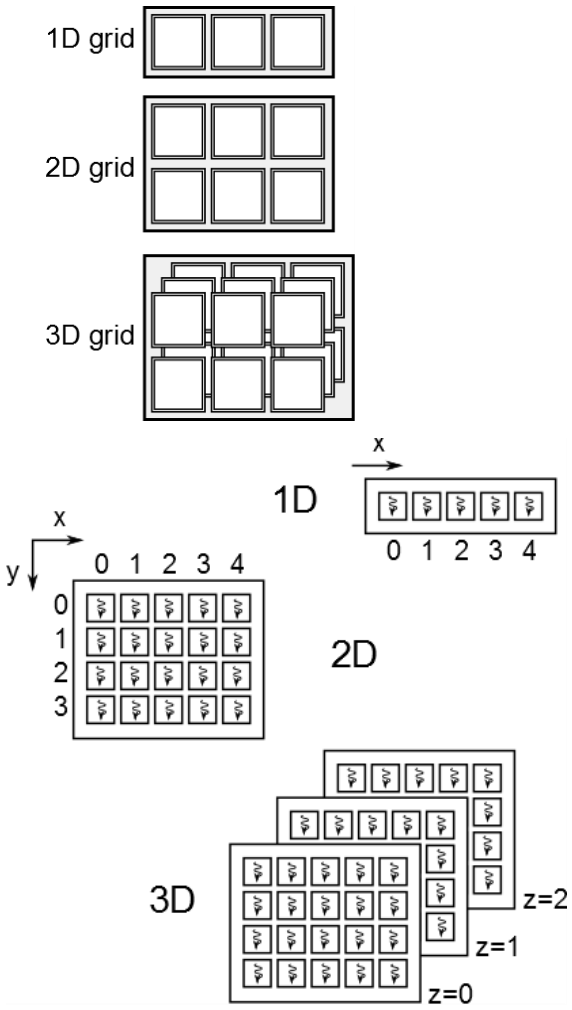
We can control how many threads execute our kernel by setting up a **grid**.

Each kernel has its own grid.

The hierarchy of threads:

- Threads are finest in granularity
- Blocks contain threads which can cooperate with each other
- Size of the block is limited to 1024 threads by architecture
- Because 1024 is not enough threads for GPU we create identical blocks to increase parallelism, i.e. increase the number of threads

Linking threads (blocks) and data



We can launch a lots of threads, doing the same thing. But how do we point them to different data?

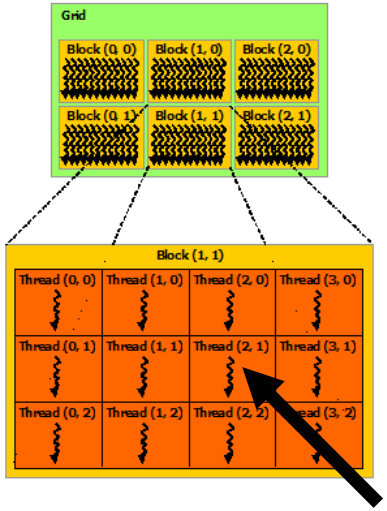
- By using their coordinates in three dimensions
- Every block has three dimensional coordinates within its grid and,
- Every thread from a block has three dimensional coordinates within that block.

Each thread has access to pre-set three dimensional variables:

- **`threadIdx`**
- **`blockIdx`**
- **`blockDim`**
- **`gridDim`**

Using these we can assign work for each thread.

Example of a grid



Values of preset variable for this thread are:
threadIdx.x = 2
threadIdx.y = 1
threadIdx.z = 0

blockDim.x = 4
blockDim.y = 3
blockDim.z = 1

gridDim.x = 3
gridDim.y = 2
gridDim.z = 1

UNIVERSITY OF OXFORD
Oxford e-Research Centre

Mapping grid onto hardware

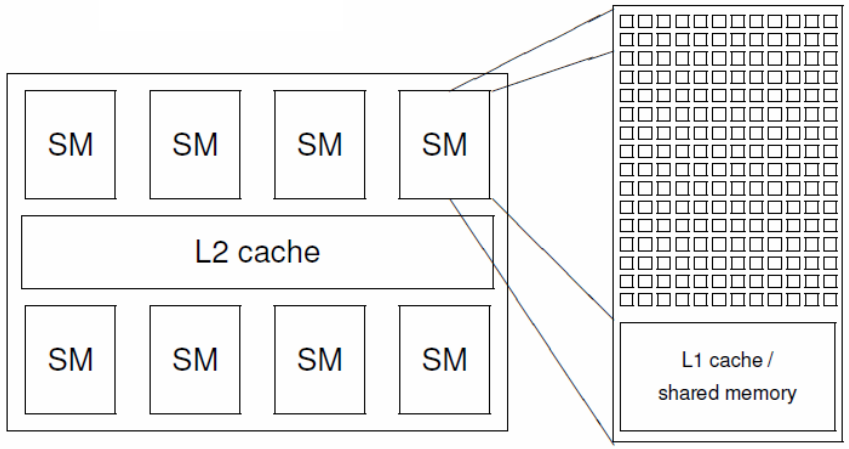
Grid is an abstraction. We cannot control what is executed where on a hardware level.

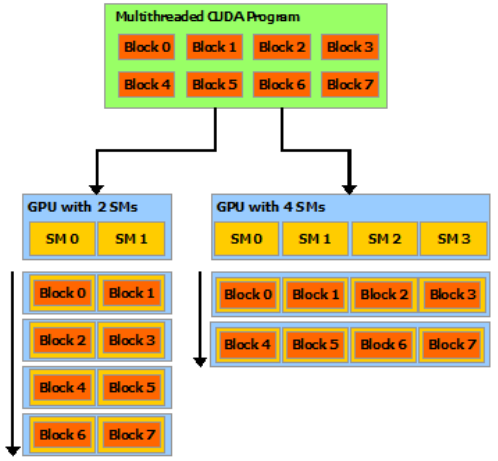
Threads:

- Threads are executed in groups of 32 called warps. (SIMD units have 32 lanes)
- Only threads from the same block can cooperate with each other efficiently.

Blocks:

- A single block execute on a single SM, blocks do not migrate.
- All concurrently running blocks on the same SM share its resources. Utilization of these resources is summarized in occupancy.
- GPU is based on latency hiding, in order to hide threads/blocks which are waiting for resources GPU needs a lot of them lined up. Spawn as many threads/blocks as possible.





How blocks are mapped to SMs is not decided by us. This also ensures scalability across different GPUs.

CUDA kernel - definition

CUDA kernel is C-like function, that is, when called, its executed on many threads in parallel.

Kernel is defined by **`__global__`** declaration specifier. Kernel is executed by each thread from CUDA grid in parallel.

Kernel must return `void`. Kernel cannot communicate with host directly, only through device memory.

Kernel is written from a point of view of a single thread, i.e. like serial code on CPU. However threads can access shared resources with all dangers that it entails in a parallel environment.

Kernel can call function which are executed on the device.

Grid for given kernel is specified by **`<<<...>>>`** execution configuration syntax.

```
// Example of kernel definition
__global__ void my_kernel(args){
  int a = 1; // variable local to the thread
  for()...; // C-like syntax
  if()...;
  sqrt = sqrt(2); // special functions
  C = A + B; // math
}
```

```
// Example of kernel definition
// for vector addition
__global__ void vector_add(
    float *d_C, float *d_A, float *d_B
){
  int index=blockIdx.x*blockDim.x + threadIdx.x;
  d_C[index] = d_A[index] + d_B[index];
}
```

UNIVERSITY OF OXFORD

Oxford e-Research Centre

CUDA kernel - execution

Execution configuration syntax:

Required:

- **Gd** is number of blocks in three dimensions in the grid.
- **Bd** is number of threads in three dimension per block.
- Both could be `int` for one dimensional case.

Optional (not discussed in this lecture):

- **Ns** is size of shared memory in bytes
- **S** is CUDA stream

```
// Kernel definition
__global__ void my_kernel(args);

// dim3 is three component vector
dim3 Gd(Gx,Gy,Gz);
dim3 Bd;
Bd.x = Bx; Bd.y = By; Bd.z = Bz;
size_t Ns;
cudaStream_t S;
my_kernel<<< Gd, Bd, Ns, S >>>(args);
```

```
// Example of kernel definition
// for vector addition
__global__ void vector_add(
    float *d_C, float *d_A, float *d_B
){
  int index=index=blockIdx.x*blockDim.x + threadIdx.x;
  d_C[index] = d_A[index] + d_B[index];
}
```

```
// Example of simplest configuration (using int
// instead of dim3 and omitting optional parameters
int nBlocks = 1;
int nThreads = 100;
Vector_add<<< nBlocks, nThreads >>>(d_C, d_A, d_B);
```

```
// Example of 2D configuration, omitting optional
// parameters
dim3 Gd(2, 2, 1);//unused dimension is set to 1
dim3 Bd(32, 4, 1);
Vector_add<<< Gd, Bd >>>(d_C, d_A, d_B);
```

```
// Example of 3D configuration, omitting optional
// parameters
dim3 Gd(2, 2, 2);
dim3 Bd(32, 4, 4);
Vector_add<<< Gd, Bd >>>(d_C, d_A, d_B);
```

Function calls in CUDA

CPU code for fancy 'Hello world'

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

void helloworld(void) {
            printf("Hello world!\n");
}

int main(void) {

            helloworld();

            return (0);
}
```

- __global__ defines a CUDA kernel cannot be combined with others declarations.
- __device__ defines function resident on the device and callable from device only.
- __host__ defines function resident on the host and callable from host only. It is a default, but must be mentioned if combined with __device__.

Examples of functions in CUDA

```
//
// __device__
// Executed on the device,
// callable from the device only,
// cannot be combined with __global__.
__device__ helloworld(void){
  printf("Hello world!\n");
}


// __host__
// Executed on the host,
// callable from the host only,
// cannot be combined with __global__.
__host__ helloworld_host(void){
  printf("Hello world!\n");
}


// Executed on the host when called by the host
// and on the device if called from the device.
// Does not serve as a kernel, cannot be called
// from host and execute on the device.
__host__ __device__ helloworld_both(void){
  printf("Hello world!\n");
}
```

UNIVERSITY OF OXFORD

Oxford e-Research Centre

Scheduling – order of execution

Blocks and threads (warps) are scheduled for execution when resources they need become available.

There is no guaranteed order of execution.

- Threads within a block can be executed concurrently
- Blocks from one grid can be executed concurrently
- Grids on different GPU can be executed concurrently

Only (reasonable) means of cooperation is among threads from the same block. These threads can:

- Access shared resources – shared memory
- Synchronize – synchronization means that it is guaranteed that all threads had finish their instructions before synchronization point. This does not mean there is guaranteed order of execution before or after synchronization.

❖ Warps are executed in lock step, i.e. all 32 threads from one warp are executing same instruction. This is only partially true with new Volta architecture.

Scheduling – ‘Hello world’ example

Blocks

```
            Helloworld_device]$
Hello world from block 5!
Hello world from block 8!
Hello world from block 1!
Hello world from block 6!
Hello world from block 2!
Hello world from block 7!
Hello world from block 3!
Hello world from block 0!
Hello world from block 4!
Hello world from block 9!
            Helloworld_device]$
Hello world from block 5!
Hello world from block 4!
Hello world from block 8!
Hello world from block 1!
Hello world from block 6!
Hello world from block 2!
Hello world from block 7!
Hello world from block 3!
Hello world from block 0!
Hello world from block 9!
```

Threads from one warp

```
Hello world from block 0 and thread 0!
Hello world from block 0 and thread 1!
Hello world from block 0 and thread 2!
Hello world from block 0 and thread 3!
Hello world from block 0 and thread 4!
Hello world from block 0 and thread 5!
Hello world from block 0 and thread 6!
Hello world from block 0 and thread 7!
Hello world from block 0 and thread 8!
Hello world from block 0 and thread 9!
```

Warps

```
Hello world from block 0 and warp 0!
Hello world from block 0 and warp 2!
Hello world from block 0 and warp 3!
Hello world from block 0 and warp 6!
Hello world from block 0 and warp 7!
Hello world from block 0 and warp 1!
Hello world from block 0 and warp 8!
Hello world from block 0 and warp 5!
Hello world from block 0 and warp 9!
Hello world from block 0 and warp 4!
```

Vector addition example

What is vector addition?

$$C = A + B$$

Where ***A, B, C*** are vectors of length ***n***. If we rewrite this by elements we get

$$C_0 = A_0 + B_0$$
$$C_1 = A_1 + B_1$$
$$\dots$$
$$C_n = A_n + B_n$$

We can map this to threads for example like this:
Thread 0 calculates C_0
Thread 1 calculates C_1
…
Thread n calculates C_n

Another way:
Thread 0 calculates elements from C_0 to C_{m-1}
Thread 1 calculates elements from C_m to C_{2m-1}
Thread 2 calculates elements from C_{2m} to C_{3m-1}
…
Thread n/m calculates from C_{n-m} to C_n

Yet another way:
Threads 0 to 31 (a warp) would calculate elements from C_0 to C_{m*32-1} with step 32
Threads 32 to 63 (next warp) would calculate elements from C_{m*32} to C_{2m*32} with step 32
…

Do these mappings differ in any way beside the way we map them?

On vector addition we will demonstrate typical processing flow of the GPU code.

Processing flow of GPU accelerated code

1. **Initiate host** – declare variables, allocate memory on the host, load data
2. Initiate device - allocate memory, set environment variables
3. Transfer data to device
4. Run GPU kernel
5. Transfer data back to host

host code for vector addition

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <cuda_runtime_api.h>

int main(void) {
  // What we want to calculate is C = A + B, where
  // A, B, C are vectors of size N
  size_t N = 1; // scalar
  float *h_A, *h_B, *h_C;

  // allocate host memory
  h_A = (float*) malloc(N*sizeof(float));
  h_B = (float*) malloc(N*sizeof(float));
  h_C = (float*) malloc(N*sizeof(float));

  // initiate host data
  for(size_t i=0; i<N; i++) {
    h_A[f] = i + 1.0f;
    h_B[f] = i + 1.0f;
    h_C[f] = 0;
  }

  ...
}
```

Processing flow of GPU accelerated code

1. **Initiate host** – declare variables, allocate memory on the host, load data
2. Initiate device - allocate memory, set environment variables
3. Transfer data to device
4. Run GPU kernel
5. Transfer data back to host

With added error checking. **NULL** is one way of saying the pointer is empty. If memory cannot by initialized pointer is returned empty (handed).

host code for vector addition

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <cuda_runtime_api.h>

int main(void) {
  // What we want to calculate is C = A + B, where
  // A, B, C are vectors of size N
  size_t N = 1; // scalar
  float *h_A=NULL, *h_B=NULL, *h_C=NULL;

  // allocate host memory
  h_A = (float*) malloc(N*sizeof(float));
  h_B = (float*) malloc(N*sizeof(float));
  h_C = (float*) malloc(N*sizeof(float));
  if(h_A==NULL || h_B==NULL || h_C==NULL)
    return(1);

  // initiate host data
  for(size_t i=0; i<N; i++) {
    h_A[f] = i + 1.0f;
    h_B[f] = i + 1.0f;
    h_C[f] = 0;
  }

  ...
}
```

Processing flow of GPU accelerated code

1. Initiate host – declare variables, allocate memory on the host, load data
2. **Initiate device** - allocate memory, set environment variables
3. Transfer data to device
4. Run GPU kernel
5. Transfer data back to host

host code for vector addition

```
...
int main(void) {
  ...

  // initiate GPU
  int deviceid = 0;
  int devCount;
  cudaGetDeviceCount(&devCount);
  if(deviceid<devCount) cudaSetDevice(deviceid);
  else return(1);

  // define device variables
  float *d_A, *d_B, *d_C;

  // allocate memory on the device
  cudaMalloc(&d_A, N*sizeof(float));
  cudaMalloc(&d_B, N*sizeof(float));
  cudaMalloc(&d_C, N*sizeof(float));

  ...
}
```

Processing flow of GPU accelerated code

1. Initiate host – declare variables, allocate memory on the host, load data
2. **Initiate device** - allocate memory, set environment variables
3. Transfer data to device
4. Run GPU kernel
5. Transfer data back to host

With added error checking. The `cudaError_t` is variable type for storing result of CUDA calls. If the call was successful it returns **`cudaSuccess`**.

host code for vector addition

```
...
int main(void) {
  ...

  // initiate GPU
  int deviceid = 0;
  int devCount;
  cudaGetDeviceCount(&devCount);
  if(deviceid<devCount) cudaSetDevice(deviceid);
  else return(1);

  // define device variables
  float *d_A, *d_B, *d_C;

  // allocate memory on the device
  cudaError_t err_code;
  err_code = cudaMalloc(&d_A, N*sizeof(float));
  if(err_code!=cudaSuccess) return(1);
  err_code = cudaMalloc(&d_B, N*sizeof(float));
  if(err_code!=cudaSuccess) return(1);
  err_code = cudaMalloc(&d_C, N*sizeof(float));
  if(err_code!=cudaSuccess) return(1);
  cudaMemset(d_C, 0, N*sizeof(float));

  ...
}
```

Processing flow of GPU accelerated code

1. Initiate host – declare variables, allocate memory on the host, load data
2. Initiate device - allocate memory, set environment variables
3. **Transfer data to device**
4. **Run GPU kernel**
5. **Transfer data back to host**
6. **Clean up**

We are launching kernel on grid containing one thread. Not very parallel of us.

host code for vector addition

```
...
int main(void) {
  ...

  // transfer data from host to device
  cudaMemcpy(d_A, h_A, N*sizeof(float),
    cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, N*sizeof(float),
    cudaMemcpyHostToDevice);

  // run CUDA kernel for vector add
  vector_add<<<1,1>>>(d_C, d_A, d_B);

  // transfer result to host
  cudaMemcpy(h_C, d_C, N*sizeof(float),
    cudaMemcpyDeviceToHost);

  // free memory on the host and the device
  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);
  free(h_A);
  free(h_B);
  free(h_C);

  return(0);
}
```

Processing flow of GPU accelerated code

1. Initiate host – declare variables, allocate memory on the host, load data
2. Initiate device - allocate memory, set environment variables
3. Transfer data to device
4. Run **GPU kernel**
5. Transfer data back to host
6. Clean up

Kernel for vector addition

```
// kernel definition for vector addition
__global__ void vector_add(
    float *d_C, float *d_A, float *d_B
){
  int index=blockIdx.x*blockDim.x + threadIdx.x;
  d_C[index] = d_A[index] + d_B[index];
}
```

We use pre-set variables we have discussed before (here in **red**) to calculate which data thread should work with.

Unified memory

Data can be accessed using only one pointer on the host and as well as on the device.

Advantages:

- Unified memory simplifies CUDA code as you do not have to explicitly move data to where they are needed. There is one allocation and one deallocation.
- Simplifies usage of data structures.

Disadvantages:

- Loosing control – there might be unwanted copies. Like array C which we set on the host to zero.
- Hiding bottlenecks from view

host code for vector addition

```
int main(void) {
  // initiate GPU
  ...

  // Declare variables
  size_t N = 1;
  float *A, *B, *C;

  // allocate unified memory
  cudaMallocManaged(&A, N*sizeof(float));
  cudaMallocManaged(&B, N*sizeof(float));
  cudaMallocManaged(&C, N*sizeof(float));

  // initiate host data
  for(size_t f=0; f<N; f++) ...

  // run CUDA kernel for vector add
  vector_add<<<1,1>>>(C, A, B);

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

  // free memory on the host and the device
  cudaFree(A);
  cudaFree(B);
  cudaFree(C);

  return(0);
}
```

UNIVERSITY OF OXFORD

Oxford e-Research Centre

Indexing – 1D vector addition

Each thread has access to pre-set three dimensional variables:

- **`threadIdx`**
- **`blockIdx`**
- **`blockDim`**
- **`gridDim`**

Using these we can assign work for each thread.

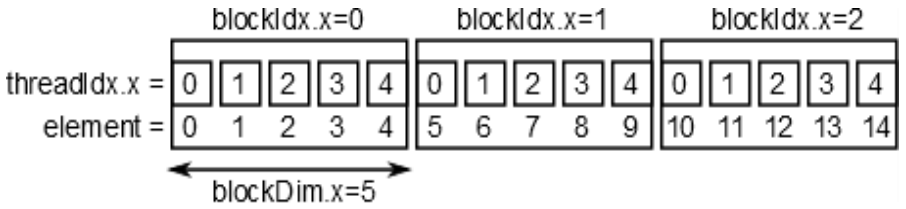
Example of vector addition:
Input is 1D array of floats

```
float *A;
```

If we launch **`vector_add`** kernel like this:

```
vector_add<<<N/5,5>>>(…)
```

For **`N=100`** grid will have 20 blocks each with 5 threads.



Indexing data like this:

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

Assign 1st block to first 5 elements, 2nd block to next 5 elements (starting with 5th element) and so on.

```
// kernel for vector addition
__global__ void vector_add(
    float *d_C, float *d_A, float *d_B
){
  int index=blockIdx.x*blockDim.x + threadIdx.x;
  // index has different value for each thread
  // for 0th block, 3rd thread index=3
  // for 1st block, 3rd thread index=8
  // ...
  d_C[index] = d_A[index] + d_B[index];
}
```

UNIVERSITY OF OXFORD
Oxford e-Research Centre

Two other ways of doing vector addition

The difference between two other ways of mapping vector addition onto a GPU is that a single thread is processing **M** elements but with different step size.

The step size is important because it determines if threads are accessing memory in **coalesced** or **uncoalesced** manner.

We say that threads from one warp (threads which are next to each other 0, 1, 2, …) has coalesced memory access, if they are accessing data which are next to each other.

This is because memory delivers data in chunks of N bytes. We call these chunks **cachelines**. Every time a thread requests data, surrounding data comes along with it, even if we do not used them. If we can process these data as well we have them for 'free'.

Indexing data will be similar.

```
#define M 32
// Kernel definition for vector addition where
// thread access data with step 1
__global__ void vector_add_step1(
    float *d_C, float *d_A, float *d_B
){
  for(int f=0; f<M; f++){
    int index=M*blockIdx.x*blockDim.x +
              M*threadIdx.x + f;
    d_C[index] = d_A[index] + d_B[index];
  }
}

// Kernel definition for vector addition where
// thread access data with step blockDim.x
__global__ void vector_add_stepbig(
    float *d_C, float *d_A, float *d_B
){
  for(int f=0; f<M; f++){
    int index=M*blockIdx.x*blockDim.x + threadIdx.x
              + f*blockDim.x;
    d_C[index] = d_A[index] + d_B[index];
  }
}

// kernel configuration is same for both
vector_add_*<<<N/(M*M),M>>>(...)
```

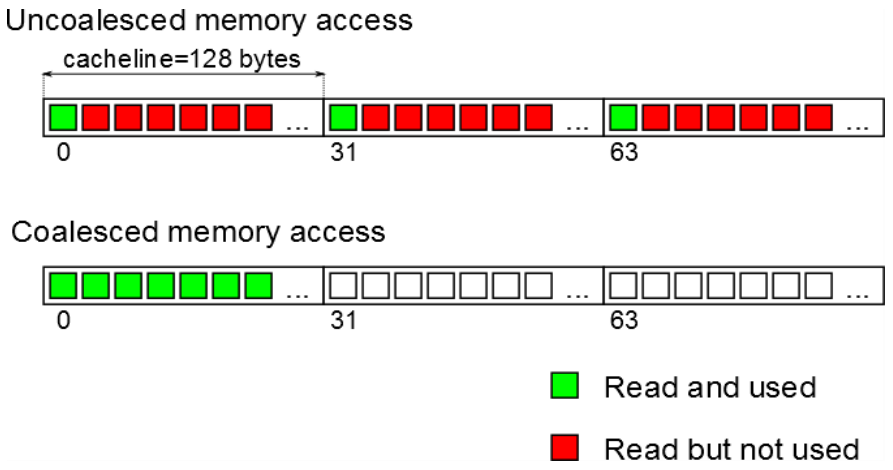
Performance difference

On NVIDIA GTX 1080 for N=67 M points

Uncoalesced memory access execution time: 28.77 ms

Coalesced memory access execution time: 2.29 ms

Speed up **12x**

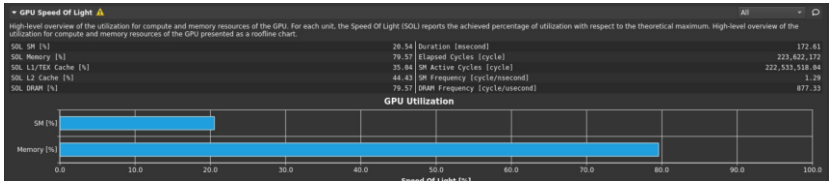


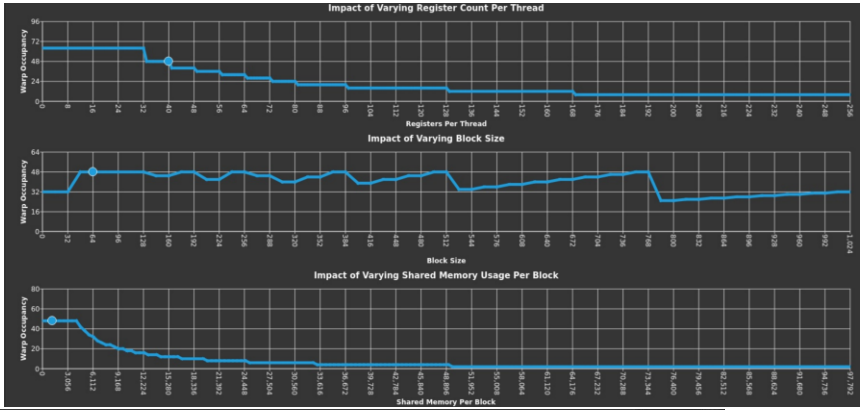
Data are read from memory in blocks of 128 bytes (32 floats), even if we request one value all others are delivered anyway. Coalesced memory access is better because it uses these ‘free’ values.

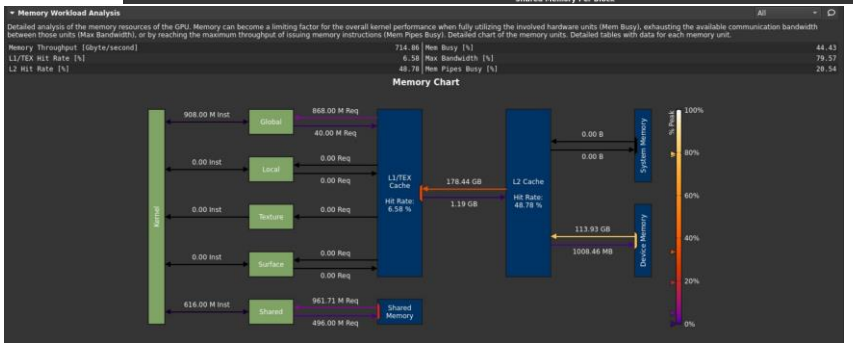
Visual profiler – Nsight Compute/Systems

Great tool for evaluating performance bottlenecks. It provides you with graphical interface in which you can explore performance of your code.

- Compute bound or memory bound?
- Occupancy calculation
- Memory bandwidth utilisation
- Compute utilisation
- Stall reasons







What have we learnt?

- How to write a basic code with GPU kernels.
- Basics of how to use Unified memory or manage memory ourselves.
- How to launch kernels using blocks and threads.
- Basics of how to use blocks and threads ids to map data to threads.
- That NVIDIA visual profiler is a good thing to have.

Further reading

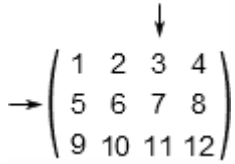
- CUDA Toolkit Documentation (https://docs.nvidia.com/cuda/index.html)
- NVIDIA whitepapers (https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf)
- Code samples from SDK

In the next lecture…

Scientific computations on GPUs!

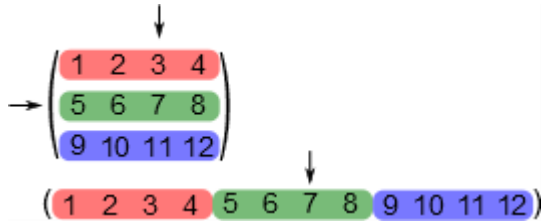
Linearisation (flattening) of an array

Matrix



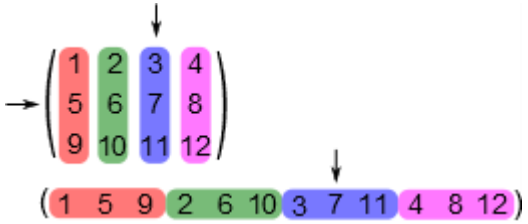

```
int nRows=3;
int nColumn=4;

float M[nRows][nColumns];
M[1][2]==7;
```

Row major format




```
// numbers from rows are next to each other
float M[nRows*nColumns];
M[1*nColumns + 2] == 7;
```

Column major format

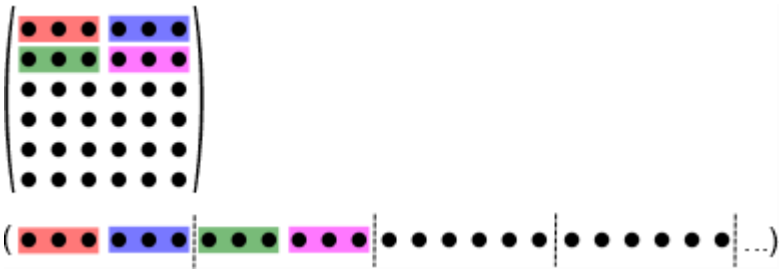



```
// numbers from columns are next to each other
float M[nRows*nColumns];
M[2*nRows + 1] == 7;
```

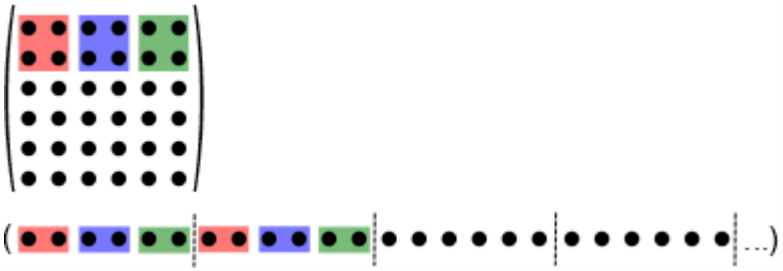
UNIVERSITY OF OXFORD

Oxford e-Research Centre

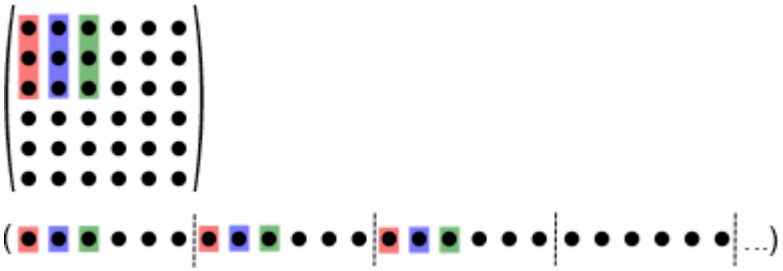
Indexing – 2D matrix addition



```
dim3 nBlocks, nThreads;
// N columns, M rows
int N, M;

// linear indexing = we ignore that it is a matrix
int index=blockIdx.x*blockDim.x + threadIdx.x;

int nThreads=3;
matrix_add<<<(N*M)/nThreads, nThreads>>>(...)
```



```
// 2D indexing, we selecting sub-matrices
int index_x = blockIdx.x*blockDim.x + threadIdx.x;
int index_y = blockIdx.y*blockDim.y + threadIdx.y;
int index   = index_y*N + index_x;

dim3 nThreads(2,2,1);
dim3 nBlocks;
nBlocks.x = N/nThreads.x;
nBlocks.y = M/nThreads.y;
nBlocks.z = 1;
matrix_add<<< nBlocks, nThreads>>>(...)
```

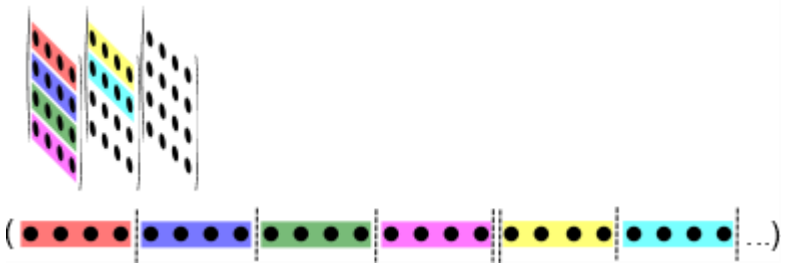


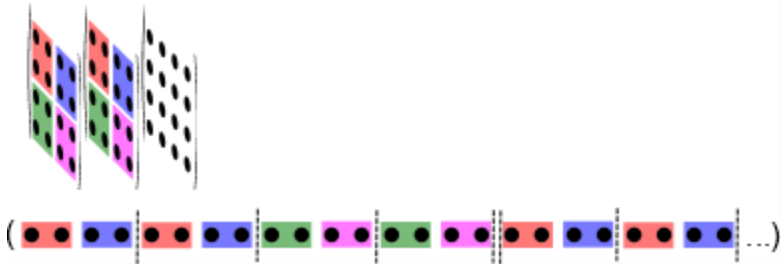
```
// reading row major as column major
int index_x = blockIdx.x*blockDim.x + threadIdx.x;
int index_y = blockIdx.y*blockDim.y + threadIdx.y;
int index   = index_y*N + index_x;

dim3 nThreads(1,3,1);
...
```

UNIVERSITY OF OXFORD

Oxford e-Research Centre

Indexing – 3D tensor addition





```
dim3 nBlocks, nThreads;
// N columns, M rows, K matrices
int N, M, K;

// linear indexing = we ignore that it is a tensor
int index=blockIdx.x*blockDim.x + threadIdx.x;

int nThreads=4;
matrix_add<<<(N*M*K)/nThreads, nThreads>>>(...)
```

```
// 3D indexing, we selecting sub-tensors
int index_x = blockIdx.x*blockDim.x + threadIdx.x;
int index_y = blockIdx.y*blockDim.y + threadIdx.y;
int index_z = blockIdx.z*blockDim.z + threadIdx.z;
int index   = index_z*N*M + index_y*N + index_x;

dim3 nThreads(2,2,2);
dim3 nBlocks;
nBlocks.x = N/nThreads.x;
nBlocks.y = M/nThreads.y;
nBlocks.z = K/nThreads.z;
matrix_add<<< nBlocks, nThreads>>>(...)
```

Why use anything else then linear indexing?

- Because getting coordinates x,y,z from linear index when you need them is a pain.

```
// getting coordinates x,y,z from linear index
int z = index%(N*M);
int y = (index - z*N*M)%N;
int x = index - z*N*M - y*N;
```

Dealing with arbitrary size data

What if we have data which are not multiple of our number of threads?

- Round number of blocks up
- Inside kernel check is threads are reading or writing inside allocated memory
- It depends on indexing scheme used

```
// kernel for tensor addition
__global__ void vector_add(float *d_C, float *d_A,
float *d_B, int N, int M, int K){
  int index_x = blockIdx.x*blockDim.x + threadIdx.x;
  int index_y = blockIdx.y*blockDim.y + threadIdx.y;
  int index_z = blockIdx.z*blockDim.z + threadIdx.z;
  int index = index_z*N*M + index_y*N + index_x;
  // since number of blocks is rounded up some
  // threads might read or write outside allocated
  // memory. We have to check the boundaries.
  if(index_x < N && index_y < M && index_z < K)
    d_C[index] = d_A[index] + d_B[index];
}
```

```
int N=113, M=113, K=113;

dim3 nThreads(32,4,4);
dim3 nBlocks;
// round number of blocks up
nBlocks.x = (N+nThreads.x-1)/nThreads.x;
nBlocks.y = (M+nThreads.y-1)/nThreads.y;
nBlocks.z = (K+nThreads.z-1)/nThreads.z;

Tensor_add<<<nBlocks, nThreads>>>(...);
```