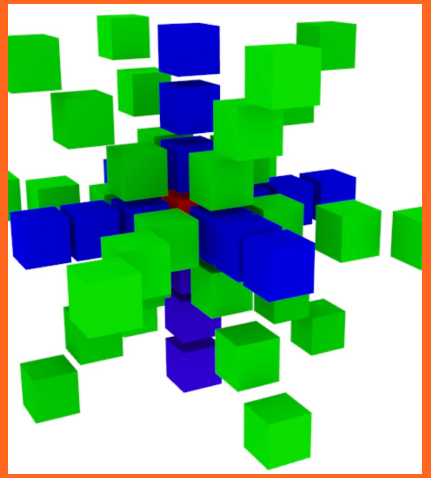
CS-E400201 - Special Course in Computer Science D: Modern High-performance Computing Tools

Hybrid computing using GPUs

Maarit Käpylä

maarit.kapyla@aalto.fi

Aalto University
School of Science



Recap

The two trajectories resulting from the power wall

Multicore processors (core==CPU)

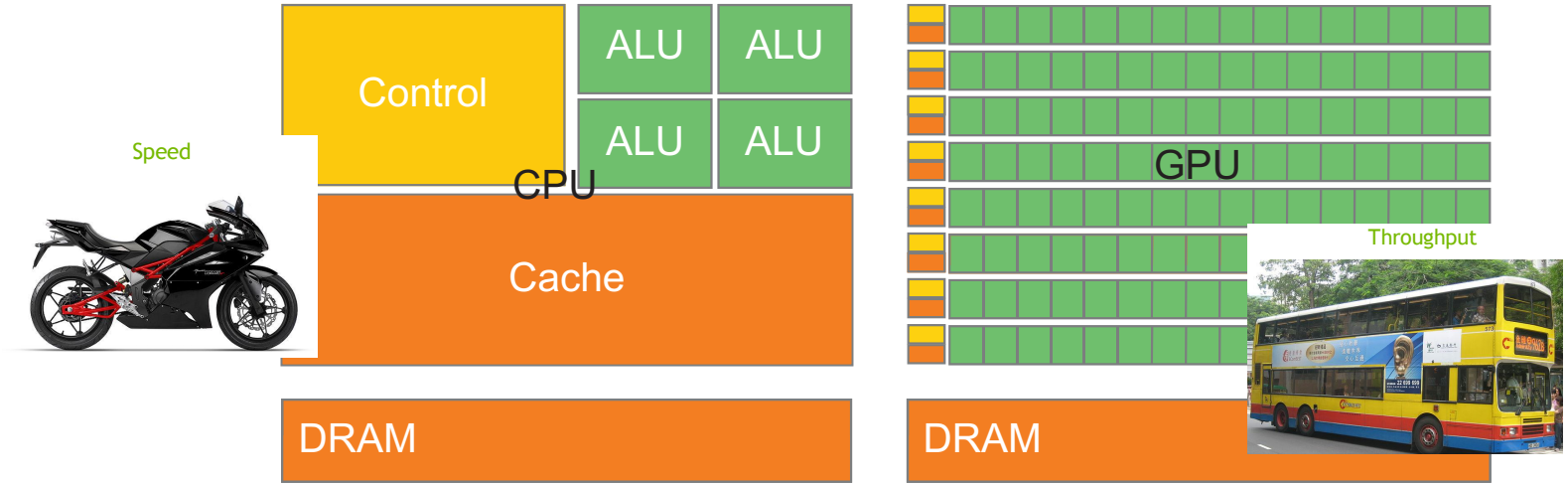
Lecture 5

Multi-thread processors (e.g. processors with GPUs)

This material

Schematic comparison

Adapted from [1]



Optimising **sequential** execution

Optimising **parallel** throughput

Peak throughput of GPUs about 10x higher than multi-core CPUs

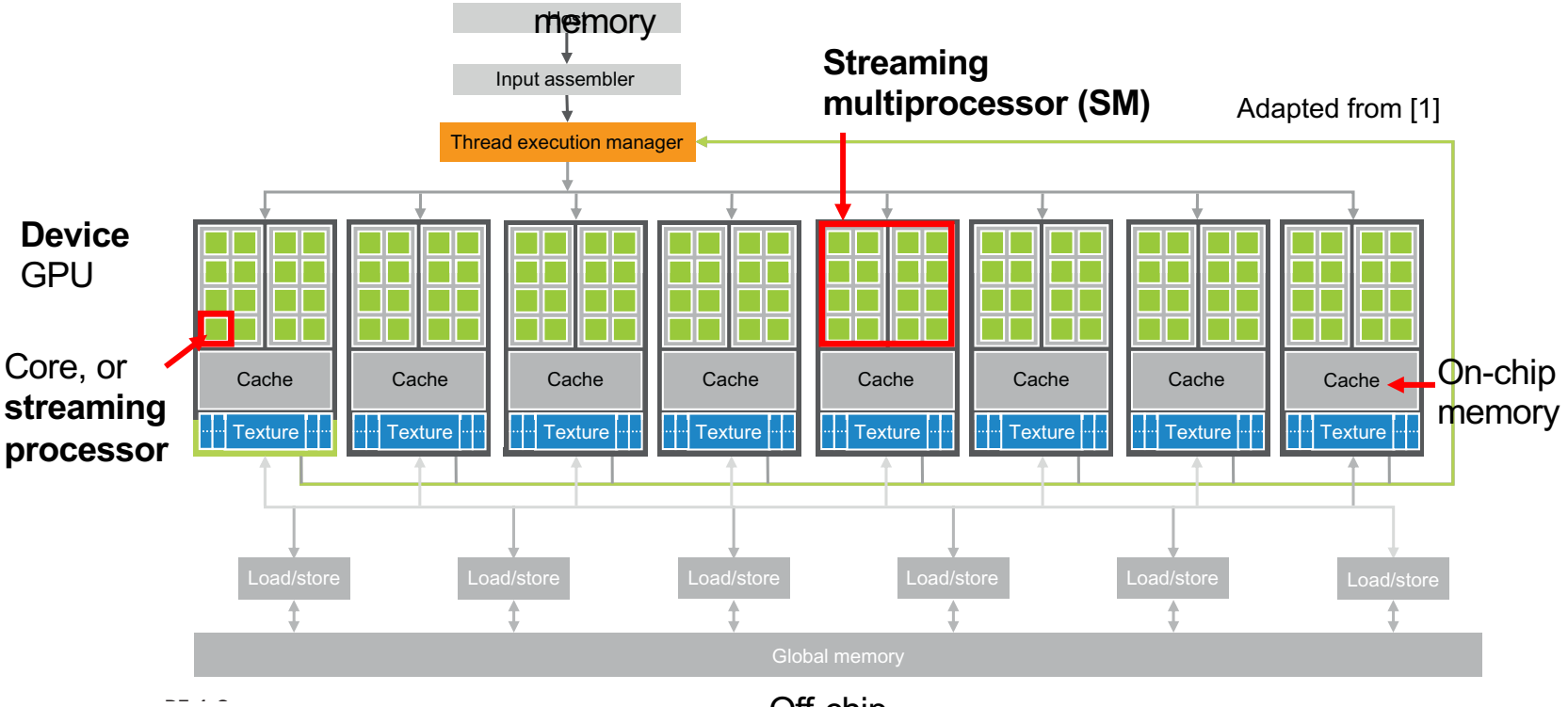
DRAM access speed 10x higher for GPUs (made for gaming)

Leading idea of large-scale computation

Execute sequential parts on CPUs
and parallel parts faster on GPUs;
Communications between GPUs
using MPI

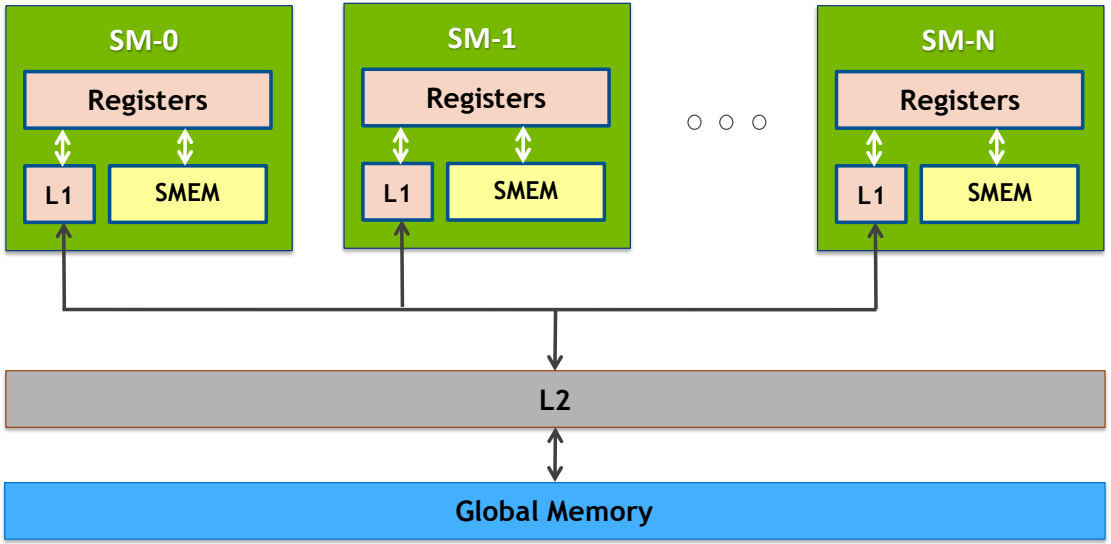
Schematic model of a GPU

CPU (**host**); has its own memory, but can access global device memory



Memory hierarchy

- Memory transfers between host and **device global memory** have the highest latency (as bad as 100x the smem); to be minimized
- Access to **shared memory** and **registers** have much lower latency
  - Registers are seen by single threads
  - Shared memory is for fast communication between threads in a block
- The sizes of the shared memory and registers are very limited.



Chapter 4 of Programming parallel computers teaches you how to make efficient code by optimizing the memory usage; please read through

https://ppc.cs.aalto.fi/ch4/v1/, …/v2 and …/v3

The GPUs in Triton

Card	total amount	nodes	architecture	compute threads per GPU	memory per card	CUDA compute capability	Slurm feature name	Slurm gres name
Tesla K80*	12	gpu[20-22]	Kepler	2x2496	2x12GB	3.7	`kepler`	`teslak80`
Tesla P100	20	gpu[23-27]	Pascal	3854	16GB	6.0	`pascal`	`teslap100`
Tesla V100	40	gpu[1-10]	Volta	5120	32GB	7.0	`volta`	`v100`
Tesla V100	40	gpu[28-37]	Volta	5120	32GB	7.0	`volta`	`v100`
Tesla V100	16	dgx[1-7]	Volta	5120	16GB	7.0	`volta`	`v100`
Tesla A100	28	gpu[11-17]	Ampere	7936	80GB	8.0		`a100`

gpuamd1	1	Dell PowerEdge R7525	2021	rome avx avx2 mi100	2x8 core AMD EPYC 7262 @3.2GHz	250GB DDR4-3200	EDR	3x MI100 32GB

The GPUs in Triton

Card	total amount	nodes	architecture	compute threads per GPU	memory per card	CUDA (*) compute capability	Slurm feature name	Slurm gres name
Tesla K80*	12	gpu[20-22]	Kepler	2x2496	2x12GB	3.7	`kepler`	`teslak80`
Tesla P100	20	gpu[23-27]	Pascal	3854	16GB	6.0	`pascal`	`teslap100`
Tesla V100	40	gpu[1-10]	Volta	5120	32GB	7.0	`volta`	`v100`
Tesla V100	40	gpu[28-37]	Volta	5120	32GB	7.0	`volta`	`v100`
Tesla V100	16	dgx[1-7]	Volta	5120	16GB	7.0	`volta`	`v100`
Tesla A100	28	gpu[11-17]	Ampere	7936	80GB	8.0		`a100`
gpuamd1	1	Dell PowerEdge R7525	2021	rome avx avx2 mi100	2x8 core AMD EPYC 7262 @3.2GHz	250GB DDR4-3200	EDR	3x MI100 32GB

(*) https://en.wikipedia.org/wiki/CUDA#Version_features_and_specifications

Kepler architecture

Kepler SM

- Each SM has its own control units, registers, execution pipelines, caches
- Many cores per SM; how many is architecture dependent
- Special-function units (cos/sin/tan, etc.)
- Shared memory/L1 cache
- Thousands of 32-bit registers
- Double precision units with architecture variable ratio; in Kepler 3:1, nowadays more DPUs.

Pascal architecture

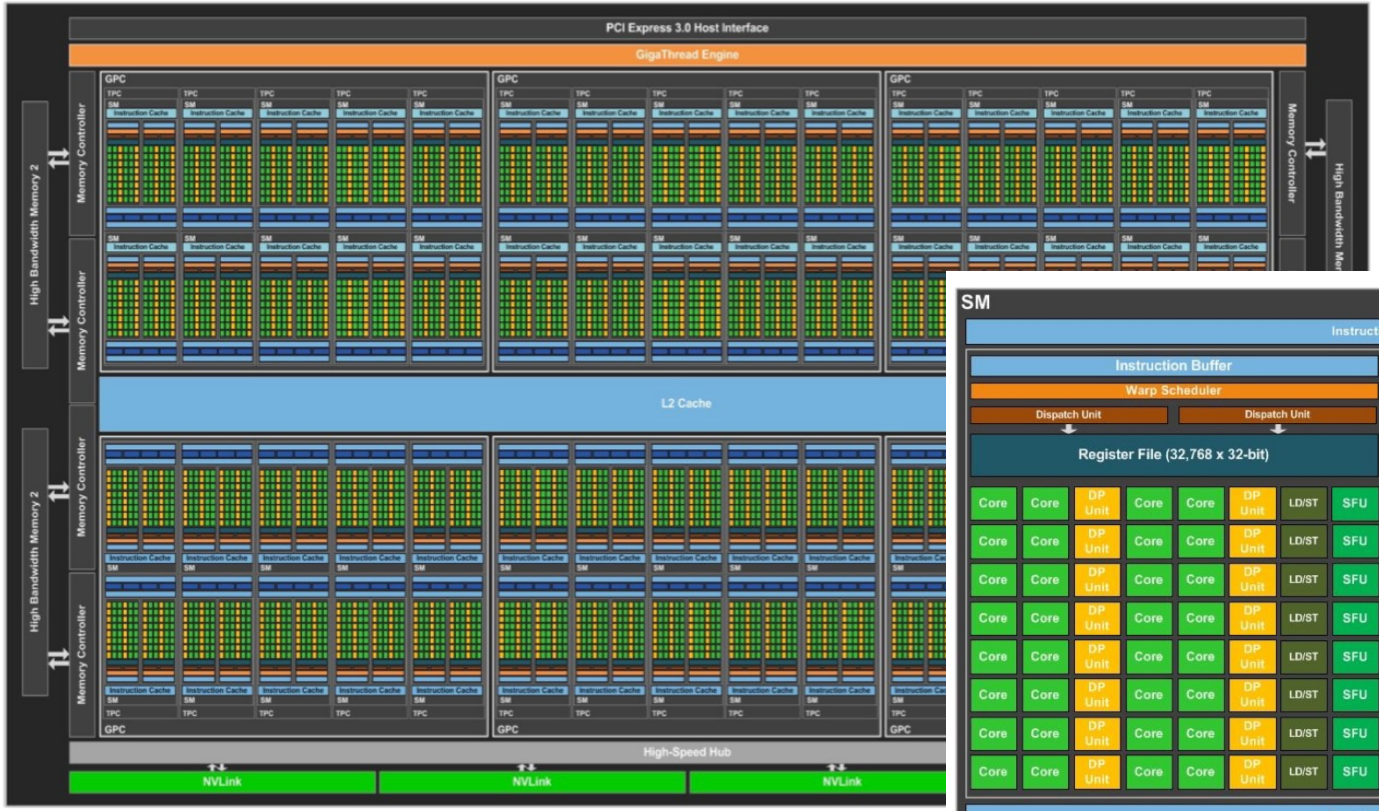


Figure 7. Pascal GP100 Full GPU with 60 SM Units

2:1 ratio of DPUs/FP32 cores

Programming models

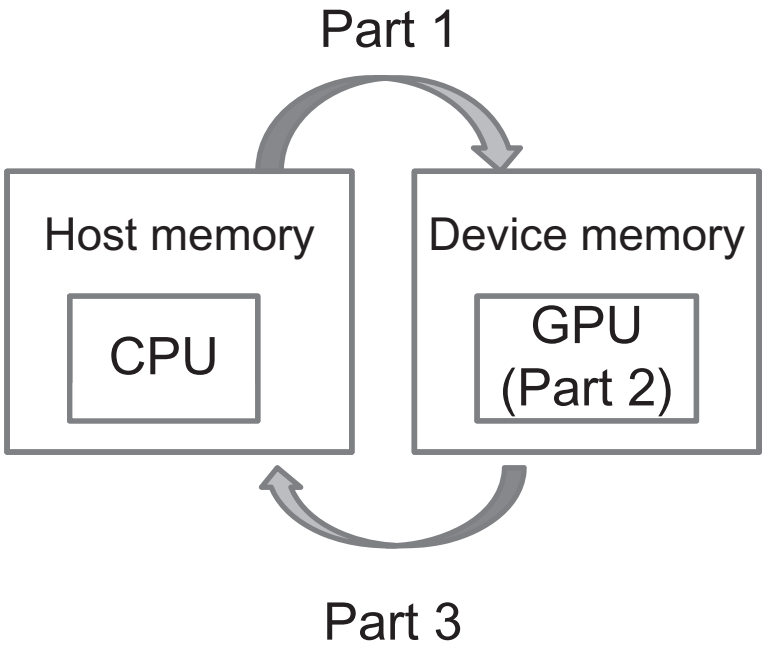
- CUDA (NVIDIA)
- Radeon Open Compute (ROCm) (AMD)
- HIP
- OpenCL
- OpenACC OpenMP
- …

For openCL examples, please refer to

https://ppc.cs.aalto.fi/ch4/v0opencl/

CUDA Execution model

- Main program is executed by the CPU
- CPU needs to communicate with the GPU (Part I)
  - Upload the data to the GPU memory
  - Upload program to the GPU
- Wait (or do something useful) for the GPU to finish computations (Part 2)
- Fetch the results back from the GPU memory (Part 3)



Memory transfers between host and device can be the bottleneck

CUDA programming model

Block

- **threads** that run on the same streaming multiprocessor (SM) form **blocks**;
- they communicate with each other through **shared memory** **located on the SM**;

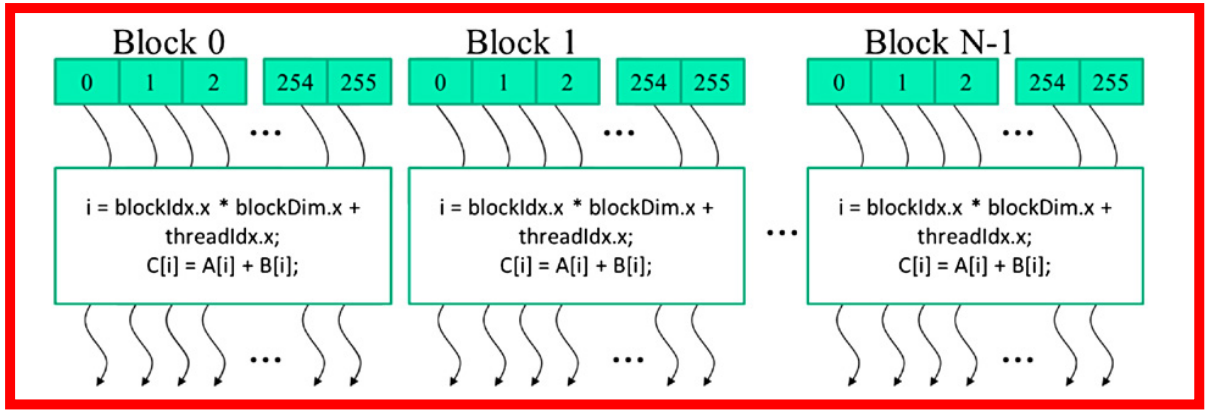
Grid

- Blocks are grouped into a **grid**; both threads and blocks have a **unique identification number**

Kernel

- Is a function that gets **executed in parallel on each thread**;
- Are executed as a grid of thread blocks

How to Identify who is who and operating on which part of the data?



CUDA programming model

Block

Phone number *Area code*

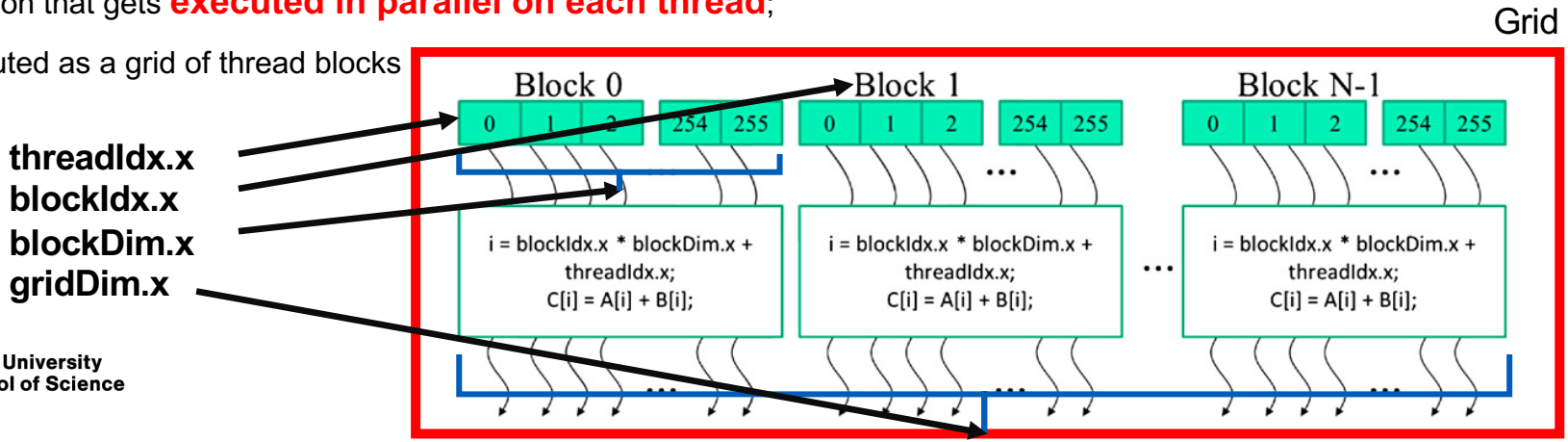
- **threads** that run on the same streaming multiprocessor (SM) form **blocks**;
- they communicate with each other through **shared memory** **located on the SM**;

Grid *Phonebook*

- Blocks are grouped into a **grid**; both threads and blocks have a **unique identification number**

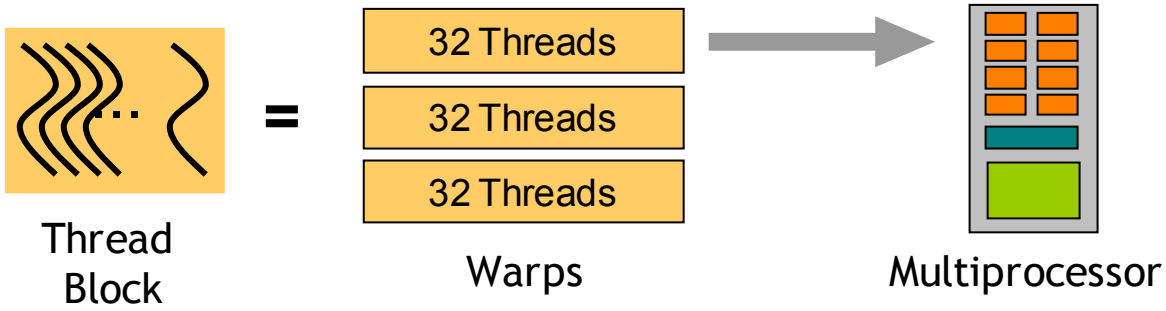
Kernel *Call the number*

- Is a function that gets **executed in parallel on each thread**;
- Are executed as a grid of thread blocks



CUDA concept of warps

- **Thread blocks are divided into warps; can be implementation dependent. In NVIDIA GPUs warps have 32 threads, in AMD's 64.**
- **Warps are physically executed in parallel on the SMs in "SIMD"-like manner.**



https://ppc.cs.aalto.fi/ch4/v1/

Programming model in practise

Let us illustrate the difference of a normal C program and a CUDA one by adding together to numbers

C program (full)

```
// Compute vector sum h_C =
h_A+h_B
void vecAdd(float* h_A, float*
h_B, float* h_C, int n)
{
for (int i = 0; i < n; i++)
    h_C[i] = h_A[i] + h_B[i];
}

int main() {
// Memory allocation for h_A,
h_B, and h_C // I/O to read h_A
and h_B, N elements each ...
vecAdd(h_A, h_B, h_C, N);
```

Cuda (host code)

```
// Compute vector sum h_C = h_A+h_B
void vecAdd(float* h_A, float* h_B, float* h_C, int
n) // "h_"refers to host
{
int size = n * sizeof(float);
float *d_A, *d_B, *d_C; //Pointers to device mem, hence start
with "d_"
cudaMalloc((void **) &d_A, size); // Allocating device mem
cudaMemcpy(d_A,h_A,size,cudaMemcpyHostToDevice);
//Copying data over to device mem
cudaMalloc((void **) &d_B, size); // Same stuff for B
cudaMemcpy(d_B,h_B,size,cudaMemcpyHostToDevice);
cudaMalloc((void **) &d_C, size); // Allocation C that'll hold the
result
vecAddKernel<<<256,256>>>(d_A,d_B,d_C, n);
cudaMemcpy(h_C,d_C,size,cudaMemcpyDeviceToHost);
//Copying result to host
cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

Prgramming model in practise

CUDA (device code)

Kernel function

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition

__global__
void vecAddKernel(float* A, float* B, float* C, int n) {

int i = blockDim.x*blockIdx.x + threadIdx.x;

if(i<n) C[i] = A[i] + B[i];

}
```

blockDim.x=dimension of the blocks requested

blockIdx.x=Block ID amongst all blocks reserved

threadIdx.x=Unique identified of the thread in a block

Step by step autopsy of the CUDA code

Allocation of global device memory

cudaMalloc((void) &DevPtr, size_t size)**

- **Address of a pointer to the allocated object in device memory**
- **Size of allocated object in terms of bytes**

cudaFree(DevPtr)

Frees object from device global memory

- **Pointer to freed object**

Step by step autopsy of the CUDA code

Transferring data to/from device global memory

cudaMemcpy(void* dst, const void* src, size_t count, cudaMemcpyKind kind)

- **o Pointer to destination**
- **o Pointer to source**
- **o Number of bytes copied**
- **o Type/Direction of transfer:**
  - **cudaMemcpyHostToDevice**
  - **cudaMemcpyDeviceToHost**

Step by step autopsy of the CUDA code

Calling the kernel function

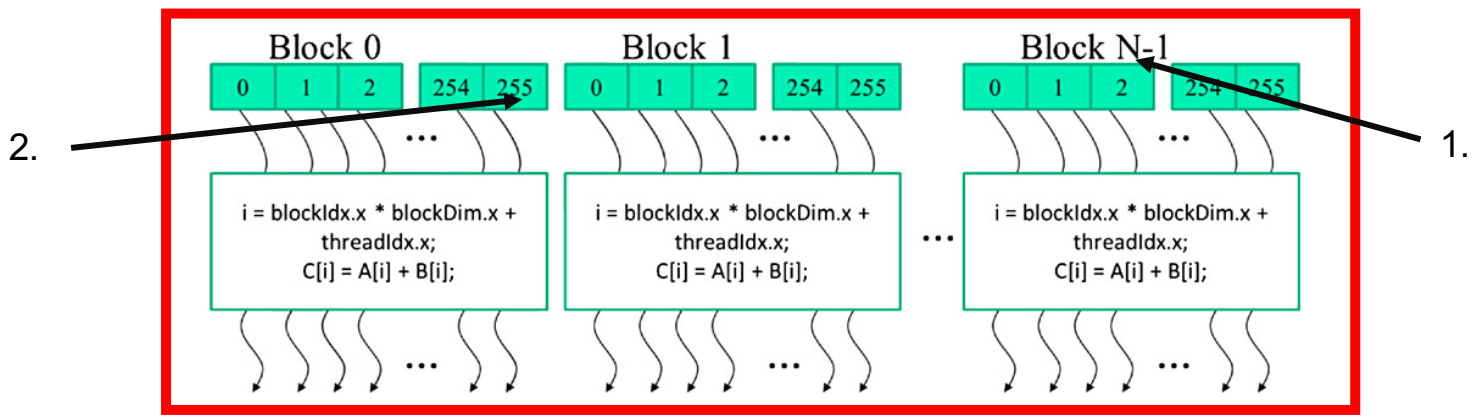
Number of BLOCKS. Now hardcoded, but in reality should depend on n, e.g. using **ceil(n/256.0)**

Number of THREADS in a Block. Max number of threads is 1024

```
vecAddKernel<<<256,256>>>(d_A,d_B,d_C, n);
```

execution configuration parameters

Traditional C function arguments



1. **exec. config param.: Number of blocks**
2. **exec. config param.: Number of threads**

Again works like a phonenumber: areacode-number

Step by step autopsy of the CUDA code

Construction of the kernel function

Cuda (device code)

Kernel function

The order of execution is random

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition

__global__
void vecAddKernel(float* A, float* B, float* C, int n) {

int i = blockDim.x*blockIdx.x + threadIdx.x;

if(i<n) C[i] = A[i] + B[i];

}
```

blockDim.x=dimension of the blocks requested

blockIdx.x=Block ID amongst all blocks reserved

threadIdx.x=Unique identified of the thread in a block

Two ***built-in variables*** that enable threads to identify themselves amongst others and know their own data area.

With the ceil function we might have reserved extra threads, hence now we need to prevent their execution with this if

CUDA C keywords for function declaration.

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

vecAdd is a bad candidate for a CUDA code

- You do a lot of data transfers between the host and device
- Very little computations
- Your CUDA code will be performing worse than a sequential code; there should always be more to compute than communicate to make a reasonable application on GPUs. Remember the ACC model!
- You are REALLY encouraged try this out.

GPU/vecAdd_CPU.c
GPU/vecAdd_GPU.cu

Generalization to multidimensional grids

The autopsied example case was dealing with one-dimensional thread blocks. Generally, however, the exec. config params

```
KernelFunction<<<dimGrid, dimBlock>>>(...);
```

dimGrid and dimBlock are dim3 type, which is a C struct with three unsigned integer fields: *x*, *y*, and *z* specifying the sizes of the three dimensions. Less than three dimensions are chosen by setting the size of the unused dimensions to 1.

```
dim3 dimGrid(2, 2, 1);
dim3 dimBlock(4, 2, 2);
```

Generalisation to multidimensional grids

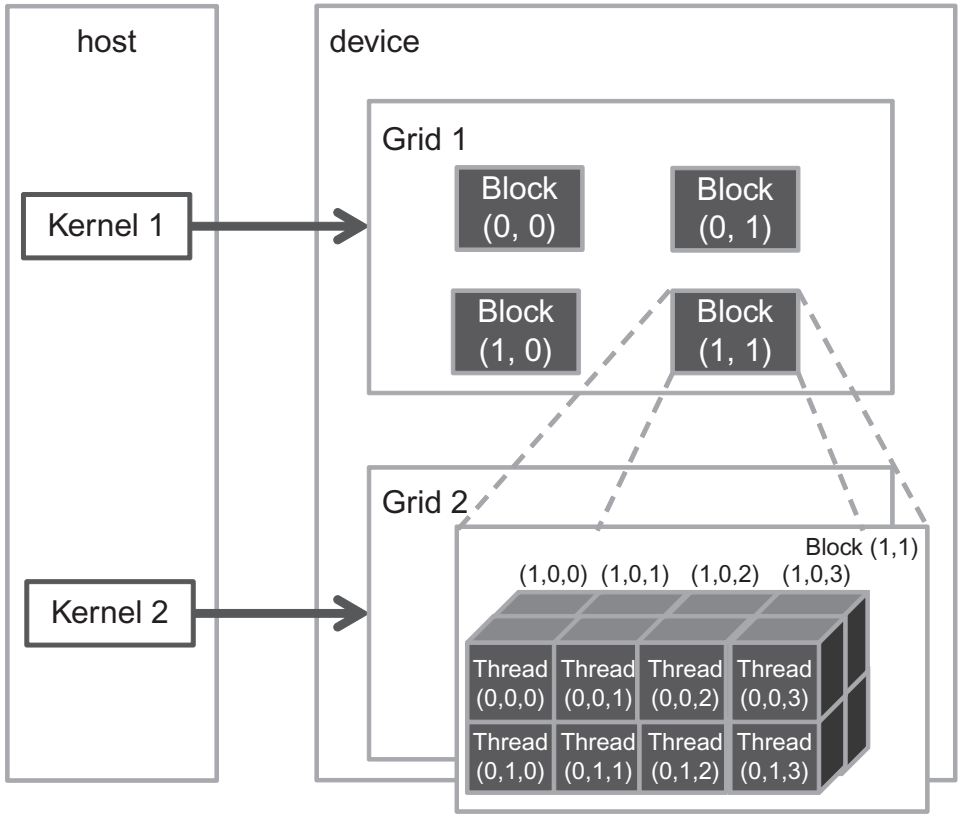
```
dim3 dimGrid(2, 2, 1);
dim3 dimBlock(4, 2, 2);

KernelFunction<<<dimGrid, dimBlock>>>(...);
```

Launch of a kernel makes the following structures available

blockDim.x, blockDim.y, blockDim.z
threadIdx.x, threadIdx.y, threadIdx.z
blockIdx.x, blockIdx.y, blockIdx.z

which tell the placement of the thread in the hierarchy.



Adapted from [1]

Brief intro to shared mem programming model

Static shared memory device code

```
__global__
void staticReverse(int *d, int n) {
__shared__ int s[64];
int t = threadIdx.x;
int tr = n-t-1;
s[t] = d[t];
__syncthreads();
d[t] = s[tr];
}
```

Dynamic shared memory device code

```
__global__
void dynamicReverse(int *d, int n) {
extern __shared__ int s[];
int t = threadIdx.x;
int tr = n-t-1;
s[t] = d[t];
__syncthreads();
d[t] = s[tr];
}
```

Calling this kernel from the host:

```
dynamicReverse<<<1, n, n*sizeof(int)>>>(d_d, n);
```

Third execution configuration parameter allocating the shared memory

CUDA streams

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
increment<<<1,N>>>(d_a);
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

- Kernel calls are asynchronous; after the kernel is launched, the code returns to the host
- CUDA calls are blocking or synchronous, such as cudaMemcpy
- All device operations run in a stream; if no stream is specified, the default (or “null”) stream is used.

CUDA streams

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
increment<<<1,N>>>(d_a);
DoSmtghOnHost();
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

- Overlapping host and device tasks is trivial due to the asynchronous nature of the kernel calls.
- How to make CUDA calls concurrently, f. ex. the computation and data transfers in the above example, requires further techniques with the concept of streams.

CUDA streams

```
cudaStream_t stream1;
cudaError_t result;
result = cudaStreamCreate(&stream1);
result = cudaStreamDestroy(stream1);
```

Non-default streams in CUDA are

- Declared (1st line),
- Created (3rd line), and
- Destroyed (4th line)

in host code as above.

CUDA streams

result = cudaMemcpyAsync(d_a, a, N, cudaMemcpyHostToDevice, streamN)

- Async data transfers can be accomplished by CUDA functions such as cudaMemcpyAsync, cudaMemcpy2DAsync(), and cudaMemcpy3DAsync(), where the 5th argument is the stream identifier.

  increment<<<1,N,0,streamX>>>(d_a)

- Kernel calls to be executed on non-default stream will have to specify the stream identifier as the 4th argument. The third argument is to declare the allocation of shared memory, here none is requested, hence 0.

CUDA streams

1. cudaDeviceSynchronize();
2. cudaStreamSynchronize(stream);
3. cudaEventSynchronize(event) (ADVANCED)

- **Since all operations in non-default streams are non-blocking with respect to the host code, you need to synchronize the host code with stream operations.**
- **Ways relevant to us:**
  1. the host code is blocked until **all previously issued operations on the device** have completed
  2. The host thread is blocked **until all previously issued operations in the specified stream** have completed

GPU/Blocking_Async.cu

How to run on multiple GPUs?

- **Nowadays commonly possible, also in Triton.**
- **You ask for multiple GPUs using --gres=gpu:N** , where **N** stands for number of requested GPUs. **Use N>1** to reserve more than one GPU. See example codes and scripts in code git repo GPU/X. Here, for short:

srun -p courses -A courses --gres=gpu:teslap100:1 ./exec1

srun -p courses -A courses --gres=gpu:teslap100:4 ./exec2

How to setup a code for multiple GPUs and MPI? GPU/sheet6/src/main.cu, reduce-multi.cu and reduce-mpi.cu

How to run on multiple GPUs?

Two general cases:

- **GPUs within a single network node: data transfers through peer-to-peer or shared host memory**
  - **peer-to-peer**: cudaDeviceEnablePeerAccess(…), cudaDeviceCanAccessPeer(…), cudaMemcpyPeerAsync(…) **[advanced, not needed to solve Sheet 6].**
  - Host launches **streams on different devices** and **collects** the results.
- **GPUs across network nodes**
  - Communication through **CUDA-aware MPI**

How to run on multiple GPUs?

cudaError_t cudaGetDeviceCount(int* count)

Returns the number of devices

cudaError_t cudaSetDevice(int device)

Device on which the active host thread should execute the device code.

cudaError_t cudaGetDevice(int* device)

Returns the device on which the active host thread executes the device code.

CUDA-aware MPI

The most likely case, as MPI tends to be SOOO complicated

```
//MPI rank 0
cudaMemcpy(s_buf_h,s_buf_d,size,
        cudaMemcpyDeviceToHost);
MPI_Send(s_buf_h,size,MPI_CHAR,1,
        100,MPI_COMM_WORLD);
//MPI rank 1
MPI_Recv(r_buf_h,size,MPI_CHAR,0,
        100,MPI_COMM_WORLD,
        &status);
cudaMemcpy(r_buf_d,r_buf_h,size,
        cudaMemcpyHostToDevice);
```

Or can we perhaps do this, and life becomes wonderful?

```
//MPI rank 0
MPI_Send(s_buf_d,size,MPI_CHAR,1,100,
        MPI_COMM_WORLD);
//MPI rank 1
MPI_Recv(r_buf_d,size,MPI_CHAR,0,100,
        MPI_COMM_WORLD, &status);
```

Yes, this is how it works!!!!!!!

Thanks to Unified Virtual Addressing (UVA) feature in CUDA; read more from [5]

srun -p courses -A courses --gres=gpu:teslap100:4 -n 4 -N 1 ./exec_comp_with_MPI

Useful reading

[1] David Kirk & Wen-Mei Whu: “Programming massively parallel processors”, third edition, 2017, Morgan Kaufmann, Cambridge, USA

[2] https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf

[3] https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[4] https://ppc.cs.aalto.fi/ch4/v1/, …/v2 and …/v3

[5] https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/