Åbo Akademi University

IT00CG19-3002
GPU Programming

Slide set #2: GPU Programming Model

Fall 2023, period 1

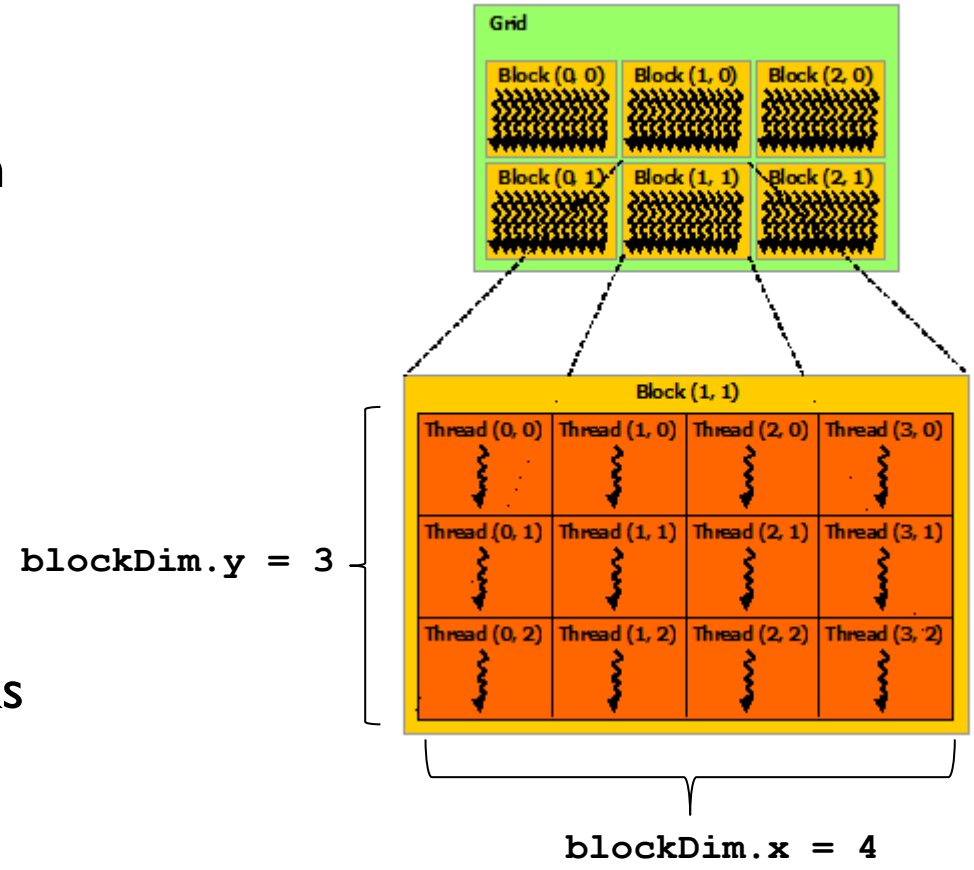
Jan Westerholm
Åbo Akademi University

GPU Programming Model

- GPUs are designed to run many small independent threads.
- GPUs have hardware for half, single and double precision floating point arithmetic/transcendental functions, in addition to integers and chars.
- Moore's law is nowadays seen in the growth of the number of parallel execution units, not in the increased clock rate of a sequential program.

GPU Programming Model

- Basic GPU programming model:
  - Design your program to run many *threads*, each operating on small amounts of data.
  - Threads are grouped into *blocks*, and blocks are grouped into a *grid* of blocks.



Simple Program: Adding Two Arrays

- Let's see how the threads work by looking at a simple program, the "Hello world" for GPUs.
- We are given two arrays A, B of length N, containing single precision floating point values:

  A[i] , B[i] , $i = 0, 1, .., N-1$
- Our task: To add each element of A to the corresponding element of B into the sum array C:

  $C[i] = A[i] + B[i]$, $i = 0, 1, .., N-1$
- This is of course straightforward on a CPU, using e.g. a for loop.

```
for ( int i = 0; i < N; ++i) c[i] = a[i]+b[i];
```

Åbo Akademi University

Array Addition

- For the GPU we need to specify how many threads we'd like to have, and what each thread is doing.
- Assumption: The value of the length N of the array is known only at program run time, not at the program design time.
- Design:
  - We launch N threads, each thread adding one element from A and B to the sum array C
  - thread ID = array index

Åbo Akademi University

Threads, Blocks, Grid

- Threads **must** be structured as *thread blocks* and the thread blocks must be structured as a *grid*
- Max # of threads in one thread block (this may change, depending on the GPU's compute capability):

  1024 x 1024 x 64 3–dim!
- Max total # of threads in one thread block: 1024
- Max # of thread blocks on the GPU (may change!):

  2147483647 x 65535 x 65535 3–dim!
- Simplest case: one-dimensional thread block, and a one-dimensional grid of blocks:

  (threads in block) x 1 x 1 and (blocks in grid) x 1 x 1

Åbo Akademi University

Threads, Blocks, Grid

- If our arrays A,B,C are shorter than 1024 elements, N < 1024, then
  - N threads in the thread block
  - one thread block is enough
- If our arrays are longer than 1024, then
  - Choose the number of threads in the thread blocks to be **`integer`***32 <= 1024, e.g. 1024, 512, 256, 128, 64, 32
  - Now calculate how many thread blocks you need in order to have at least N threads
  - There will be some threads in the last thread block that should do nothing. E.g. N = 1000, # of threads in block = 256, # of threads blocks = 1000/256 = 3.91, choose 4 thread blocks, in total 4*256=1024 threads.

Threads, Blocks, Grid

- Why should thread blocks have their number of threads as multiples of 32?
  - Threads are executed *synchronously* in bunches of 32 = ***warp***
  - All threads in the warp must have their data *ready* before the warp can run. *Ready* means the data has been copied from the main memory to one of the thread registers.
  - GPUs have caches, cache lines are 4 B x warp size = 128 B
  - GPU resources can be fully utilized when these parameters are observed
- How to calculate the # of thread blocks

  = ceil(N/threadsInBlock)

  = (N + threadsInBlock – 1)/threadsInBlock

Åbo Akademi University

Array Addition

```
// GPU kernel code for each thread to add two arrays of length N, C = A+B
// we have 1D-dimensional thread blocks
// kernels have the identifier __global__, type is void
__global__ void VecAdd(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];
}

// CPU Host code
int main(int argc, char *argv[])
{
int N = ...;
size_t arraybytes = N * sizeof(float);
// Allocate input vectors h_A and h_B in host memory
float* h_A = (float*)malloc(arraybytes);
float* h_B = (float*)malloc(arraybytes);
float* h_C = (float*)malloc(arraybytes);
// Initialize input arrays h_A, h_B
...
```

Åbo Akademi University

Array Addition

```
// Allocate arrays in GPU device memory
float* d_A; cudaMalloc(&d_A, arraybytes);
float* d_B; cudaMalloc(&d_B, arraybytes);
float* d_C; cudaMalloc(&d_C, arraybytes);
// Copy arrays A and B from host memory to device memory
cudaMemcpy(d_A, h_A, arraybytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, arraybytes, cudaMemcpyHostToDevice);

// Size of thread blocks and thread grid, then run kernel
int threadsInBlock = 256;
int blocksInGrid = (N + threadsInBlock - 1) / threadsInBlock;

VecAdd<<<blocksInGrid, threadsInBlock>>>(d_A, d_B, d_C, N);
// Copy result from device memory to host memory
// h_C contains the result in host memory
cudaMemcpy(h_C, d_C, arraybytes, cudaMemcpyDeviceToHost);
// Free device memory
cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
// Free host memory ...
}
```

Kernel Specification

- A GPU thread program is expressed as a *kernel* with identifier **__global__** and type **void**

```
__global__ void Kernel_name(args)
```

- We need to specify the size of the thread block grid we have chosen (`gridDim`) and how many threads there are in each block (`threadsInBlock`) at kernel invocation using the syntax

```
kernel_name<<<gridDim, threadsInBlock>>>(args)

gridDim:        int, dim3(i,j) or dim3(i,j,k)
threadsInBlock: int, dim3(i,j) or dim3(i,j,k)
```

- Recommendation: `threadsInBlock` = integer$*32$

Åbo Akademi University

CUDA Subroutine

```
// GPU subroutine, identified by __device__
__device__ float AddElement(float a, float b)
{
float c;
c = a+b;
return(c);
}
// GPU kernel code for each thread to add two arrays of length N, C = A+B
// we have 1D-dimensional thread blocks
// kernels have the identifier __global__, type is void
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
int i = blockDim.x * blockIdx.x + threadIdx.x;
if (i < N) C[i] = AddElement(A[i], B[i]);
}

// CPU Host code
...
```

Kernel Specification

- Each thread receives four parameters of type dim3 to calculate its ***global thread ID***:

```
dimGrid  :     dimGrid.x,   dimGrid.y,   dimGrid.z
blockIdx :     blockIdx.x,  blockIdx.y,  blockIdx.z
blockDim :     blockDim.x,  blockDim.y,  blockDim.z
threadIdx:     threadIdx.x, threadIdx.y, threadIdx.z
```

```
0 ≤ blockIdx.x < gridDim.x
0 ≤ blockIdx.y < gridDim.y
0 ≤ blockIdx.z < gridDim.z

0 ≤ threadIdx.x < blockDim.x
0 ≤ threadIdx.y < blockDim.y
0 ≤ threadIdx.z < blockDim.z
```

Kernel Specification

- Based on these parameters we can calculate a global thread index in the kernel. For a 1D grid of 1D blocks

```
mythreadx = blockDim.x * blockIdx.x + threadIdx.x
```

- Sometimes we'd like to use a thread index based on a 3D grid of blocks with 3D thread blocks because of our data layout in memory. Then

```
int blockId = blockIdx.x
            + blockIdx.y * gridDim.x
            + gridDim.x * gridDim.y * blockIdx.z;

int threadId = blockId*(blockDim.x*blockDim.y*blockDim.z)
         + (threadIdx.z *(blockDim.x * blockDim.y))
         + (threadIdx.y * blockDim.x)
         + threadIdx.x;
```

Compilation and Running

- Compile your CUDA program on dione

```
nvcc -O3 -arch=sm_70 code.cu -lm
```

- `nvcc` preprocesses the source code .cu into C/C++ and then your native compiler (eg. gcc) compiles and links. Here the result will be in `a.out` .
- `-arch` specifies the compute capability of the GPU that we are compiling for.
- Nice to know: `-arch=compute_70 -code=sm_70 -ptx` generates ptx-code from your source code
  - Text file source.cu.ptx, looks like assembly
  - Will be just-in-time compiled by the GPU driver when run on a specific GPU architecture

Compilation and Running

- On your own machine, run normally, for instance timing the execution

```
time ./a.out args
```

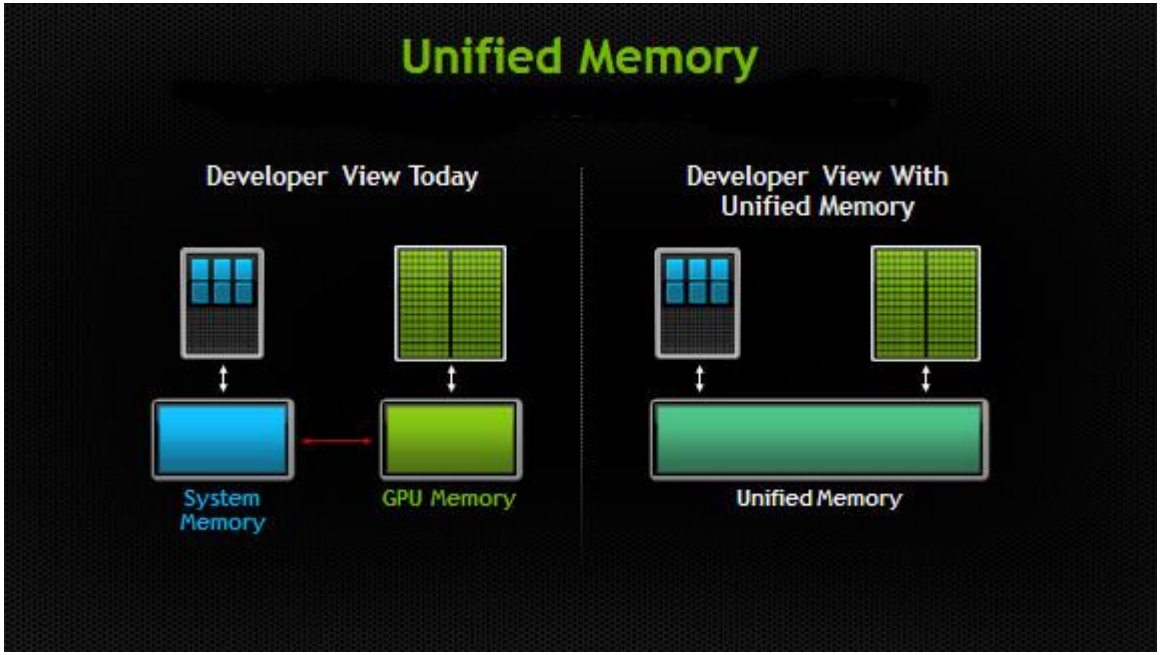
- On dione,

```
srun -p gpu -t 1:00:00 --mem=10G time ./a.out args
```

- What happens to the graphical user interface on your machine?
  - Which program will use the GPU: the OS or your program? Answer: both!
  - A laptop can have two graphics adapters. Perhaps one can be used for GUI and the other for running CUDA programs?

Memory Hierarchy in GPUs

- GPUs have their own memories. Data is (explicitly) transferred from and to CPUs.
- We can also use only one pointer: Unified memory



Unified Memory: Array Addition

```
// Device code to add two arrays A,B of length N, C = A+B
__global__ void VecAdd(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];
}

// CPU Host code
int main(int argc, char *argv[])
{
int N = ...;
size_t arraybytes = N * sizeof(float);
// Allocate input vectors A, B and output vector C
// Allocate using unified pointers
float* A; cudaMallocManaged((void **)&A, arraybytes);
float* B; cudaMallocManaged((void **)&B, arraybytes);
float* C; cudaMallocManaged((void **)&C, arraybytes);
// No explicit copying of arrays from host memory to device memory
// Initialize input arrays
...
```

Array Addition

```
// Invoke kernel
int threadsInBlock = 256;
int blocksInGrid = (N + threadsInBlock - 1) / threadsInBlock;

VecAdd<<<blocksInGrid, threadsInBlock>>>(A, B, C, N);
cudaDeviceSynchronize();
// No explicit copying of result from device memory to host memory
// C contains the result, can be used by host
…
// Free memory
cudaFree(A); cudaFree(B); cudaFree(C);
}
```

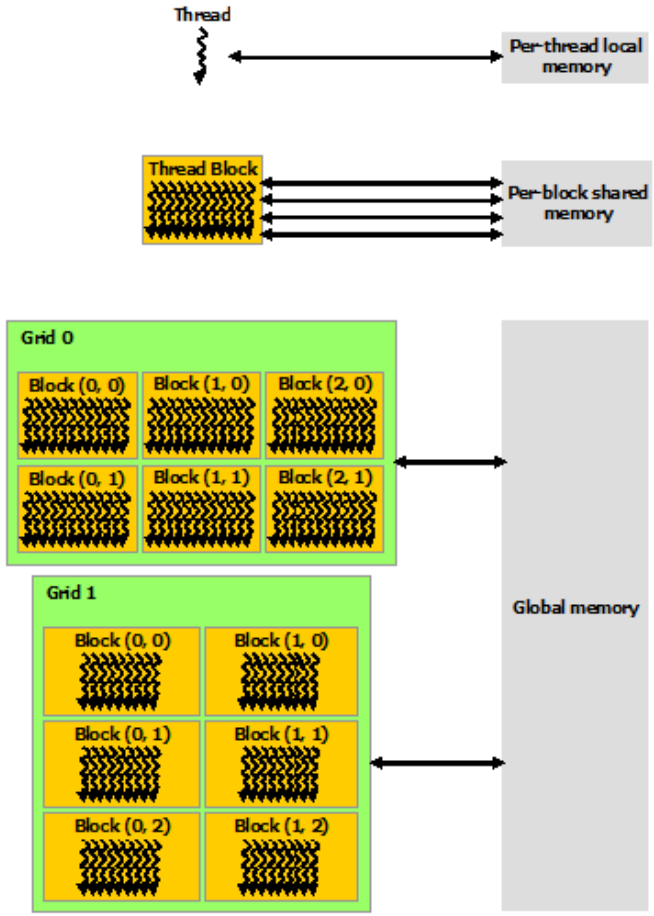
Åbo Akademi University

Memory Hierarchy on GPUs

- GPUs have a memory hierarchy with different access latencies and privileges: global, shared and register
- Some machines use error correction code ECC
  - ECC can detect and correct 1 bit errors
  - Used by most supercomputer centers
- **Global memory**, highest latency
  - Accessible to all threads and host by **`cudaMemcpy`**
- **Shared memory**, low latency
  - Accessible only to *threads within the same thread block*
- **Register** file for local thread variables, low latency

Åbo Akademi University

Memory Hierarchy in GPUs



Åbo Akademi University

Thread Synchronization

- A thread has its own register variables which other threads cannot access. A thread can access shared memory within its thread block, and global memory.
- In CUDA we have no control over thread block execution order (no block synchronization), hence we MUST design for *arbitrary thread block execution order*.
- If two threads never access each other's data then we don't have to pay any special attention to thread synchronization.

Åbo Akademi University

Thread Synchronization

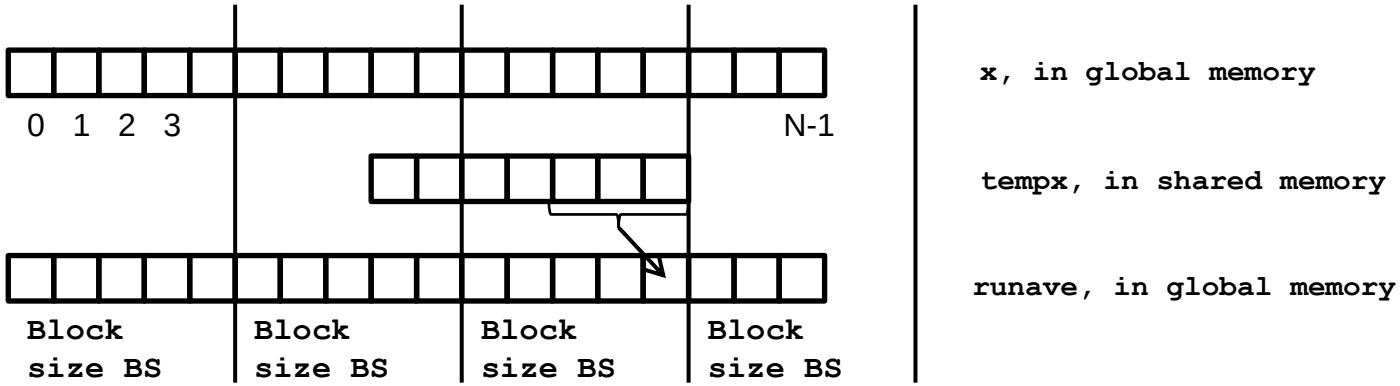
- Threads *within the same thread block* **can** be synchronized: `__syncthreads();`
- Threads from different thread blocks **cannot** be synchronized.
- Why would we like to synchronize threads?
  - We often design programs where the execution of the threads depends on data having been read into shared memory or already calculated by other threads
  - As we have no control over which threads are, have been or will be executed even within a thread block, we must synchronize the threads (= issue a barrier) making sure that all threads in the thread block have reached this point before program execution continues.

Thread Synchronization

- Syntax for thread synchronization: `__syncthreads()`
- Simple case: Calculate running averages
- Input: array **`x`** of floats, length N
- Output: array **`runave`** of N floats such that

```
runave[0] = x[0]
runave[1] = (x[0]+x[1])/2.0
```

```
if (i>1 && i<N) runave[i] = (x[i]+x[i-1]+x[i-2])/3.0
```



Thread Synchronization

- Design:
  - Each thread calculates one **`runave`** value.
  - # of threads = N.
  - Thread block size BS = 256.
  - # of thread blocks: (N+BS−1)/BS, integer division
- Implementation
  - A thread block copies BS+2 values from array **`x`** in global memory into shared memory array

```
__shared__ float tempx[BS+2]
```

  - Then we synchronize the threads: `__syncthreads();`
  - Now each thread calculates its runave value from **`tempx`** and stores the value in **`runave`** in global memory.

Åbo Akademi University

Thread Block Synchonization

- Suppose you have designed a kernel that performs a calculation and writes its result into an array in global memory on the GPU. After this you'd like to perform a reduction, that is, calculate the sum of the values in this array.
- If the number of thread blocks is greater than 1, will adding __syncthreads(); after the calculation code but before the reduction code in the kernel, be correct?
- How can we synchronize between thread blocks?

Block Synchonization

- Answer: run two kernels
  - first calculate the result array
  - then run a reduction kernel

```
VecCalc<<<blockGrid, threadsinBlock>>>(d_A, d_B, d_C, N);
//if ( cudaDeviceSynchronize()!= cudaSuccess ) return(-1);
VecReduce<<<blockGrid, threadsinBlock>>>(d_C, N);
```

  - Kernel synchronization is automatic for non-managed memory but also possible by using cudaDeviceSynchronize() which does not return before VecCalc has finished.
  - VecCalc finishes only after all thread blocks have run.

Åbo Akademi University

Atomic Operations

- If you are in luck, your program consists of N totally independent threads running in any order writing their results into individual bytes in global memory!
- In many cases this is not possible. Threads will "interact" by trying to change the same data: this is called **data race**!
  - We may for instance collect the results from each thread to `thread_result` and then add up the results into a global variable `global_result`.
  - The kernel code executed by each thread

    ```
    global_result += thread_result;
    ```

    will not work correctly. Why?

Åbo Akademi University

Atomic Operations

- We need atomic operations: while one thread reads `global_result`, adds `thread_result` to its local copy of the value of `global_result` and then writes back `global_result` to global memory, all other threads accessing `global_result` **must** wait. Essentially, we are serializing the thread execution.
- Note: Data races are not specific to global memory operations, they may also occur within shared memory used by a thread block.
- Check from the documentation that the datatype you need for the atomic operation is supported (earlier GPUs did not support atomic operations with double precision floating point numbers).

Åbo Akademi University

Atomic Operations

- Atomic add functions
  - `int atomicAdd(int* address, int val);`
  - `unsigned int atomicAdd(unsigned int* address, unsigned int val);`
  - `unsigned long long int atomicAdd(unsigned long long int* address, unsigned long long int val);`
  - `float atomicAdd(float* address, float val);`
  - `double atomicAdd(double* address, double val);`
- `int atomicSub(int *addr, int val);`
- `int atomicMin(int *addr, int val);`
- `int atomicMax(int *addr, int val);`

Atomic Operations

- Atomic increment/decrement functions contain wrap around values `val`
- `atomicInc(int *addr, int val)`
  - reads the 32-bit word **old** located at the address **addr** from global or shared memory, computes

$$(\textbf{old} >= \textbf{val})\ ?\ 0 : (\text{old}+1)$$

    and stores the result back to memory at the same address. The function returns old.
- `atomicDec(int *addr, int val)`

$$((\textbf{old} == 0) \mid (\textbf{old} > \textbf{val}))\ ?\ \textbf{val} : (\textbf{old}-1)$$

Atomic Operations

- The correct code for the kernel code

```
global_result += thread_result;
```

  is then

```
atomicAdd(&global_result,thread_result);
```

- This ensures that each thread will add in turn to the global variable. The order in which the additions are performed (which thread is adding) is in practice random. Therefore, for floating point values you might get ever so slightly different results each time you run the code!

Åbo Akademi University

Reference Guide

- When in doubt, consult the documentation

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

- Can be found by searching the Internet for "CUDA Programming"