Parallel Programming Using Graphics Processing Units (GPUs)

What is a GPU?

- Specialized hardware for processing graphics
- Available in almost all computers
- Simple and regular design
- Billions of transistors
- Can be integrated with the CPU or stand alone
- Performance: Up to 2,0 Gflops (2011)

  7.0 Tflops (2018)

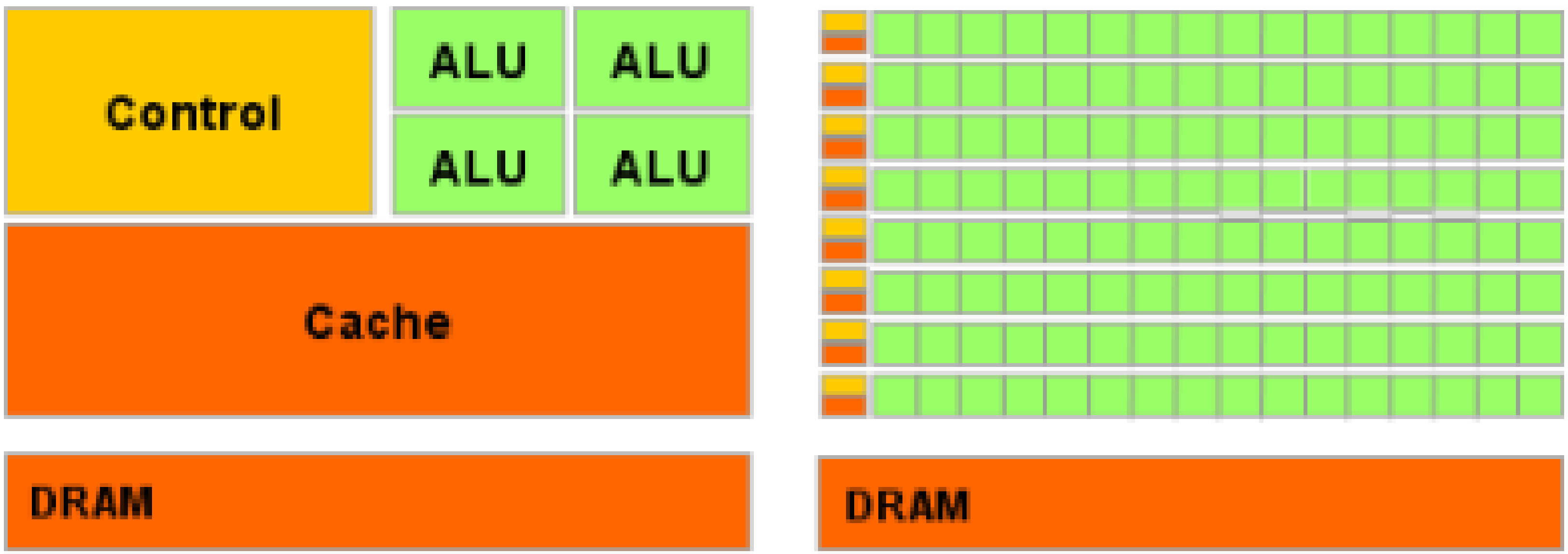
Main producers:

Nvidia, ATI (stand alone)

Intel (integrated)

ARM (mobile, in Trondheim!)

Layout: CPU vs GPU



CPU **GPU**

DRAM: memory

Cache: for data caching

Control: for flow control

ALU (Arithmetic-Logic Unit): for data processing

Mode of Operation: CPU vs GPU

CPU

- General purpose processing (Run Linux, file system, etc.)
- Typically, one thread on one CPU core
- Heavy weight threads: stop one thread to start another
- Threads on different cores can do different things (MIMD).

GPU

- Specialized, only performs compute tasks
- Can run 1000+ threads simultaneously on one GPU processor
- Light weight threads: Hardware support for massive multi-threading.
- Works best when all the threads do the same thing (SIMD).

Nvidia Tesla GPUs

- Designed for general computing
- A separate computing unit
- Up to 7 TFlops double precision
- Used in more than 50% of computers on Top500 list

Programming the GPU

CUDA: Compute Unified Device Architecture (http://www.nvidia.com/cuda)

– Designed by NVIDIA

– Supported on all modern NVIDIA GPUs (notebook GPUs, high-end GPUs, mobile devices)

– Gives high performance

– Forward compatible versions

OpenCL (http://www.khronos.org/opencl)

– Open standard, targeting NVIDIA, AMD/ATI GPUs, Cell, multicore x86, ..

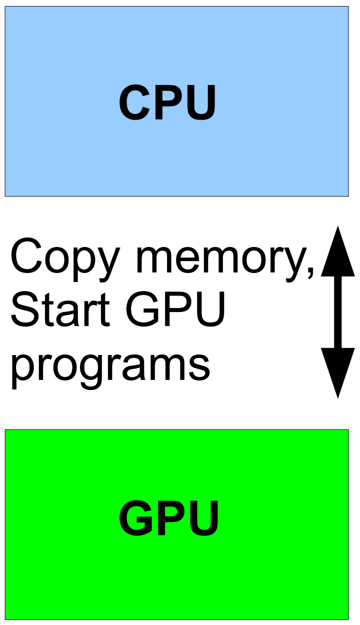
– Gives portable code, can be run on all devices that support the model (CPU, GPU)

– Lower performance than CUDA

– Less mature

CUDA – Compute Unified Device Architecture

Environment enabling programming of GPU h/w by a variety of common (non-graphics specific) high-level languages



Single source for CPU and GPU,

- CPU starts up, use routines to transfer data to and from GPU.
- Language extensions used to control GPU routines (referred to as *kernels*)

Languages

- Host code C, C++, Fortran (PGI)
- GPU code C/C++ with extensions, Fortran with extensions
- `nvcc`: NVIDIA CUDA compiler
- PGI compiler for CUDA Fortran

Program Outline

(1) C program on CPU starts up

(2) Performs initial (local) computation

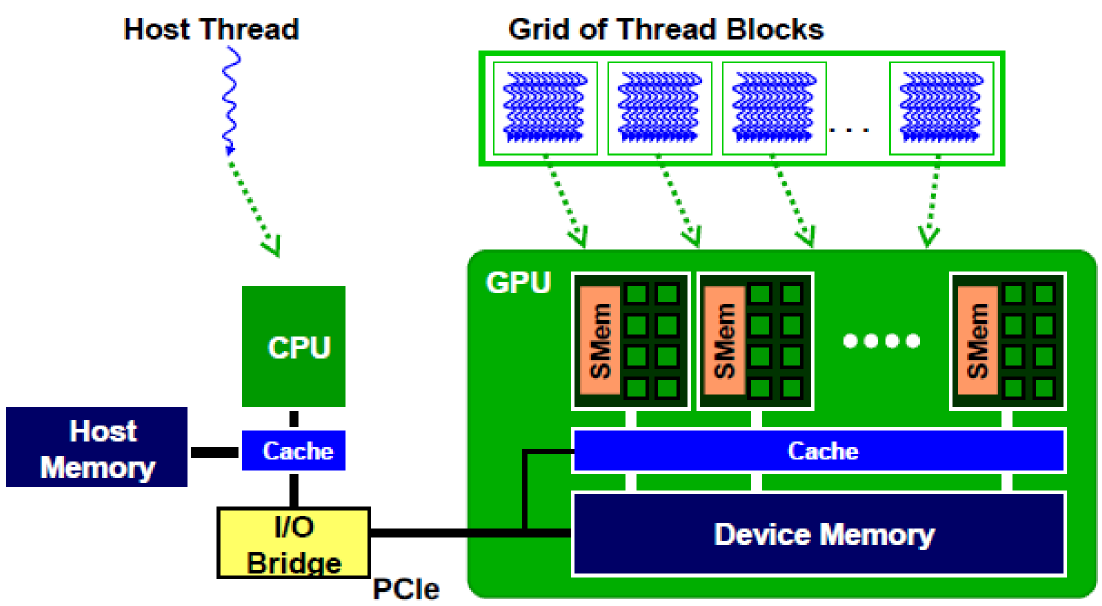
(3) Allocates memory on the GPU

(4) Copies data from CPU memory to GPU memory

(5) Specifies number of blocks and threads per block and starts GPU program (the *kernel*)

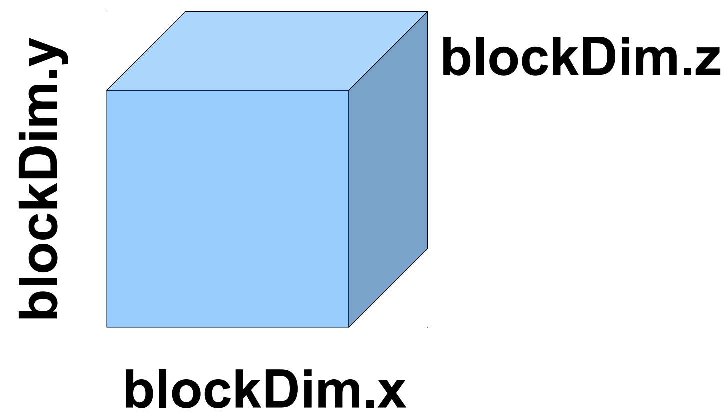
(6) GPU program executes on GPU until completion

(7) CPU copies result from GPU memory to CPU memory for post processing



Organizing threads on the GPU

A thread block:

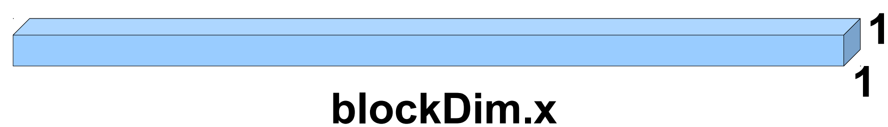


ID of a thread: `threadIdx.x,  threadIdx.y,  threadIdx.z`

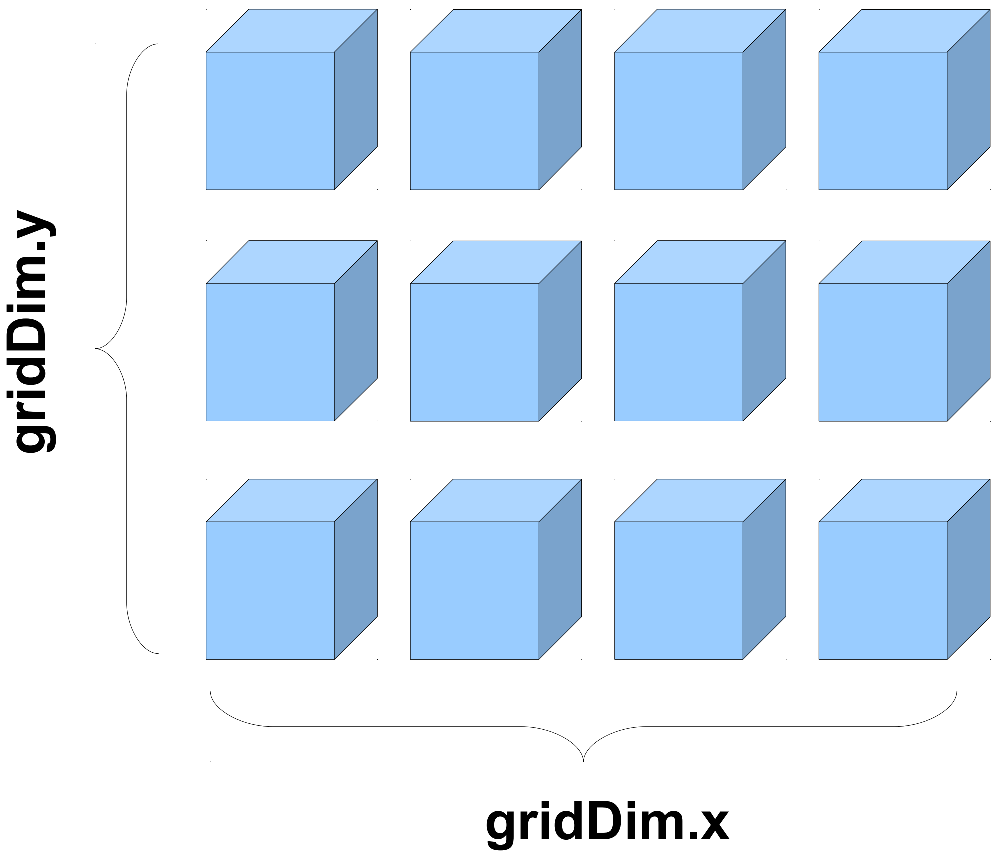
```
myid = threadIdx.z*(blockDim.y*blockDim.x) + threadIdx.y*blockDim.x +
       threadIdx.x
```

If `blockDim.z` = `blockDim.y` = 1 then

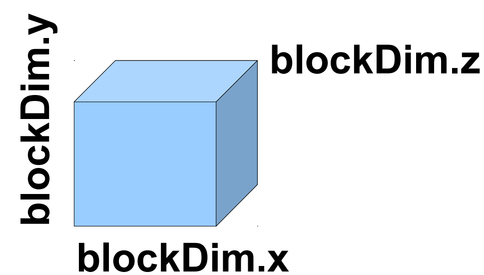
```
myid = threadIdx.x
```



A grid of thread blocks

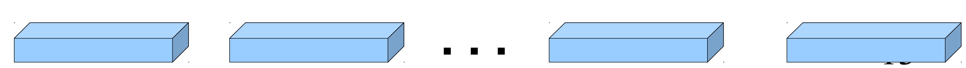


ID of a block:
```
blockIdx.x
blockIdx.y
blockIdx.z
```



If `threadDim.y` = `threadDim.z` = 1 and `gridDim.y` = `gridDim.z` = 1 then

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



Kernel launch parameters

Kernel launch parameters

- Block size: (x, y, z). x*y*z = Maximum of 768 threads total. (Hw dependent)
- Grid size: (x, y, z). Maximum of thousands of threads. (Hw dependent)

`dim3` is a Cuda data type for specifying dimension values.

Can take 1, 2, or 3 arguments:

```
dim3 blocks1D( 5       );  // Only x value, others = 1
dim3 blocks2D( 5, 5    );  // Only x and y, z = 1
dim3 blocks3D( 5, 5, 5 );  // x, y and z
```

Launching a kernel:

```
helloFromGPU<<<gridD, blockD>>>();
```

The GPU Memory Model

Local storage

- Each thread has own local storage
- Mostly registers, thus limited (KB)
- Data lifetime = thread lifetime
- Fast access

Shared memory

- Each thread block has own shared memory
- Accessible only by threads within that block
- Data lifetime = block lifetime
- Small (KB), but fast

Global (device) memory

- Accessible by all threads as well as host (CPU)
- Data lifetime = from allocation to deallocation
- Large (GB) but slower

Basic C Extensions in CUDA

Function modifiers

- `_ _global_ _` : to be called by the host but executed by the GPU.
- `_ _device_ _` : to be called and executed by the GPU only.

Variable modifiers

- `_ _shared_ _` : variable in shared memory.
- `_ _device_ _` : variable in global memory
- Variables in kernels without modifiers are thread local

Device functions

- `_ _syncthreads()` : sync of threads within a block.
  (similar to omp barrier)

Example: Vector Addition

Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Run grid of N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: Vector Addition

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

Host Code

```
int main()
{
    // Run grid of N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Blocks Threads

Host Code for VecAdd (1)

```
// allocate and initialize host (CPU) memory
float *h_A = …,   *h_B = …; *h_C = …(empty)

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

// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice) );
cudaMemcpy( d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice) );
```

Host Code for VecAdd (2)

```
// execute grid of N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);

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

// do something with the result…

// free device (GPU) memory
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

VecAdd continued

What if N is not a multiple of 256?

Number of threads will always be a product of "block size" and "number of blocks"

N = 107, Number of blocks = 11, Number of threads per block = 10

```
int i = threadIdx.x + blockDim.x *  blockIdx.x;
if (i < N)
  C[i] = A[i] + B[i];
```

What if "block size" * "number of blocks" < N?

```
int i = threadIdx.x + blockDim.x *  blockIdx.x;
while (i < N) {
  C[i] = A[i] + B[i];
  i += blockDim.x * gridDim.x;
}
```