High-Performance Computing
With GPUs

What are GPUs ?

- GPU – graphics processing unit

- Originally designed as a graphics processor

- NVIDIA GeForce 256 (1999) – first GPU
    - single-chip processor for mathematically-intensive tasks
    - transforms of vertices and polygons
    - lighting
    - polygon clipping
    - texture mapping
    - polygon rendering

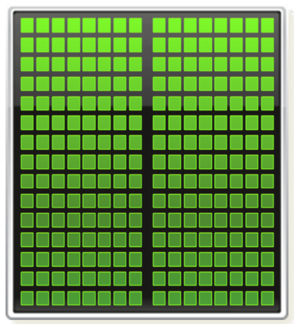
- NVIDIA Geforce 3, ATI Radeon 9700 – early 2000's
    - Now programmable!

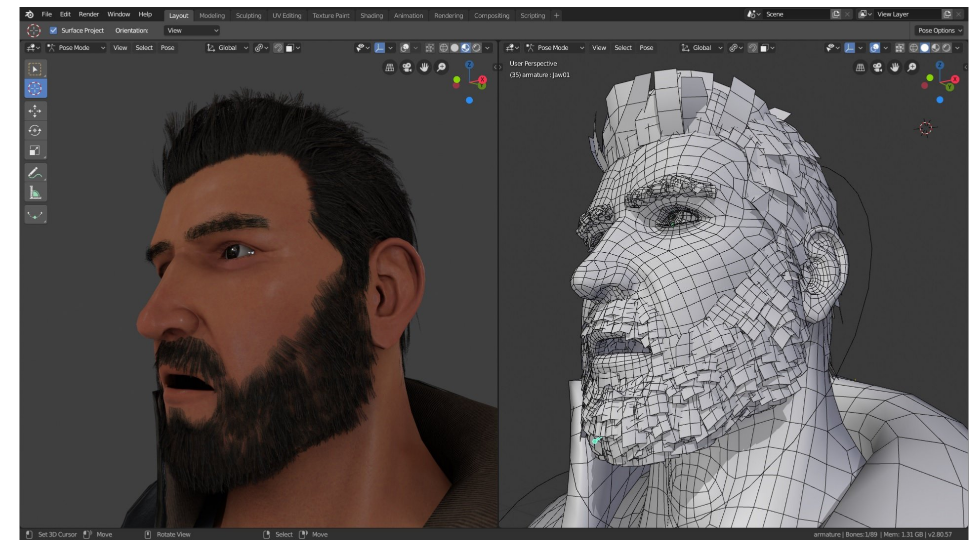
What are GPUs ?

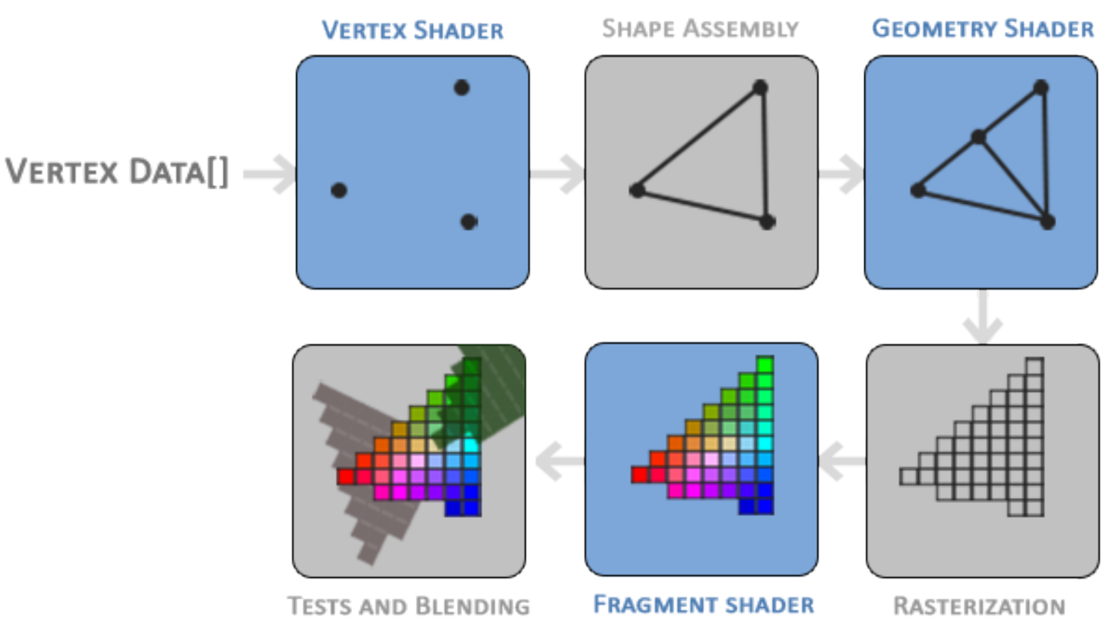
Modern GPUs are present in

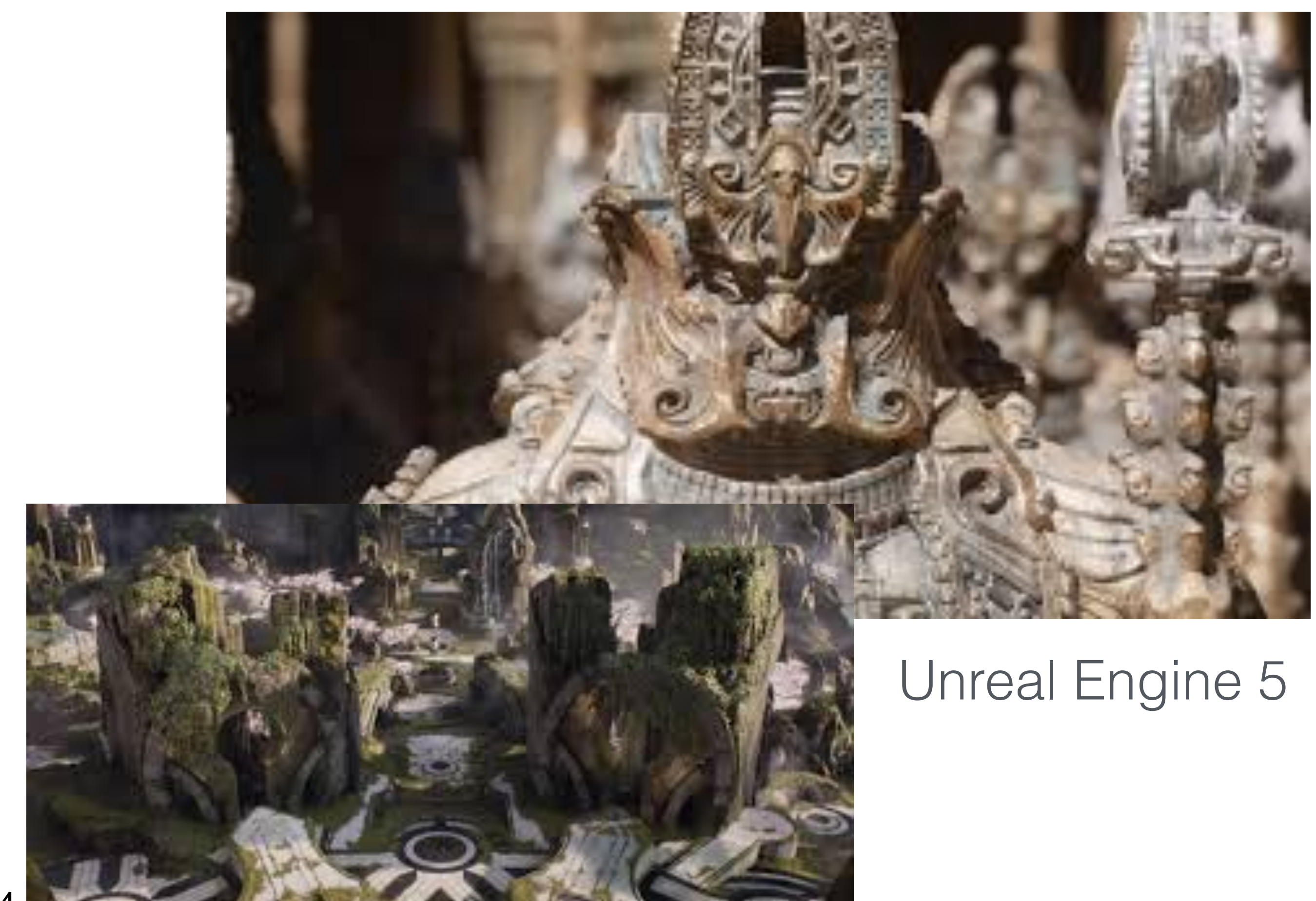
- ✓ Embedded systems
- ✓ Personal Computers
- ✓ Game consoles
- ✓ Mobile Phones
- ✓ Workstations

Historical GPUs workflow









Unreal Engine 5

GPGPU

GPGPU = General Purpose computation using GPU and graphics API in applications other than 3D graphics where GPU accelerates critical path of application

Timeline:
1999-2000 computer scientists from various fields started using GPUs to accelerate a range of scientific applications.
- GPU programming required the use of graphics APIs such as OpenGL and Cg.

2001 – LU factorization implemented using GPUs

2006 - NVIDIA launched CUDA, an API that allows to code algorithms for execution on GeForce GPUs using the C programming language.

2008 - Khronos Group defined the OpenCL programming language. It is supported on AMD, NVIDIA and ARM GPU platforms. OpenCL code can also be compiled to run on CPUs.

2012 - NVIDIA presented and demonstrated OpenACC - a set of directives that greatly simplify parallel programming of heterogeneous systems. Kepler architecture

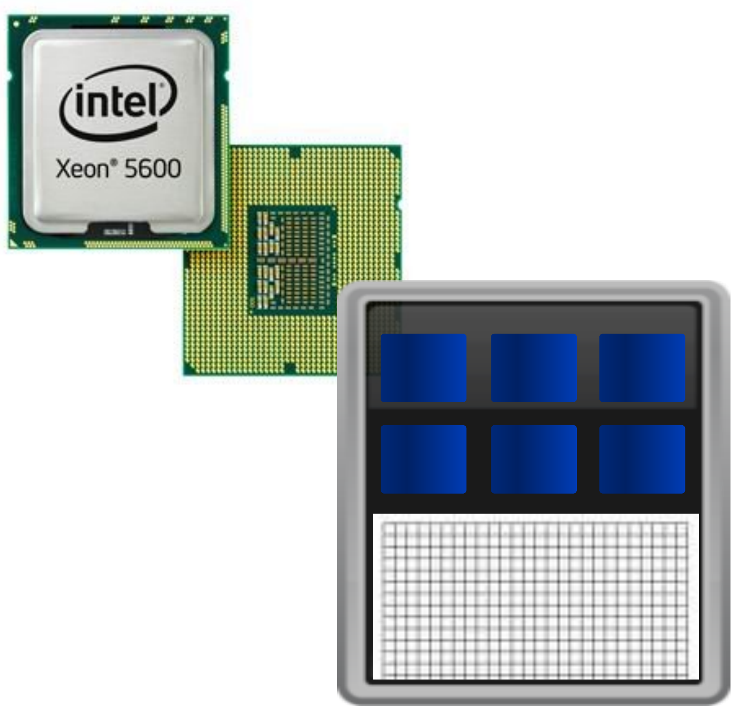
2013 - First mobile processors Tegra

2016 - Pascal Architecture - GPUs become the first tool for AI and deep learning

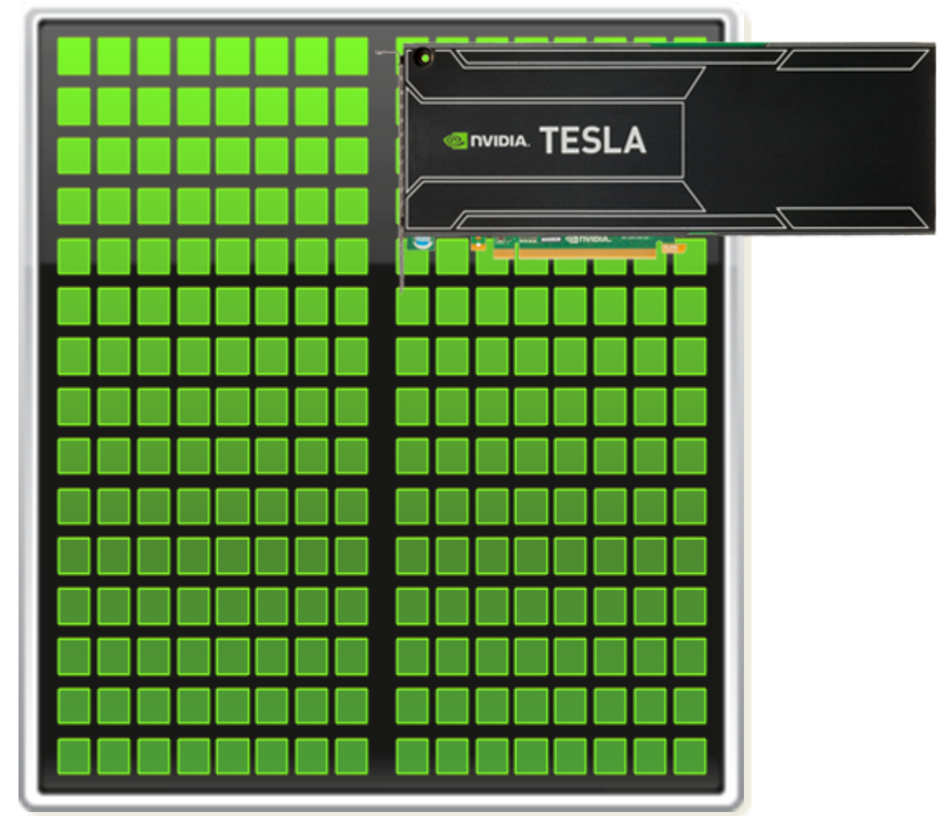
2020 - Nvidia buys Mellanox and ARM

CPU VS GPU

CPU



GPU

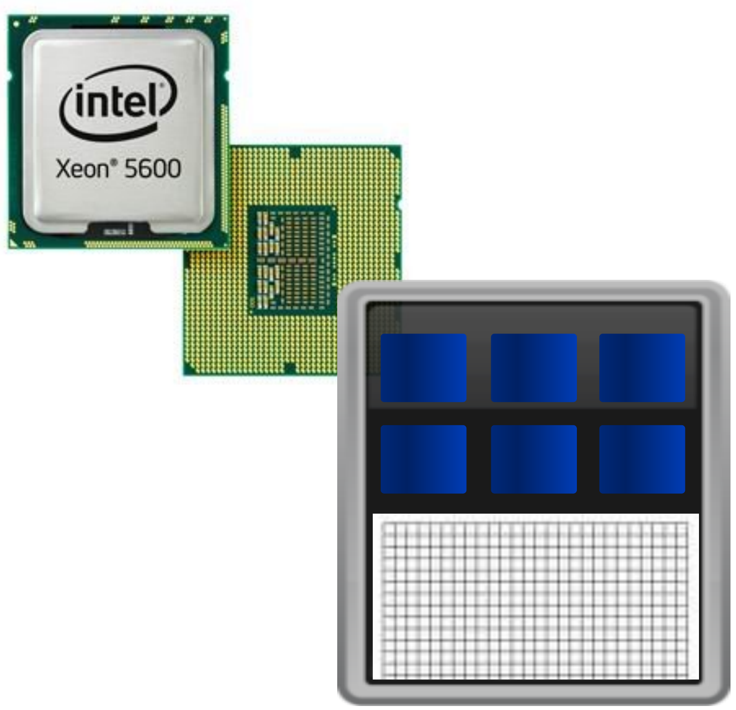


CPUs consist of a few cores optimized for serial processing and general purpose calculations.

GPUs consist of hundreds or thousands of smaller, efficient cores designed for parallel performance. The hardware is designed for specific calculations.

CPU VS GPU

SCC CPU

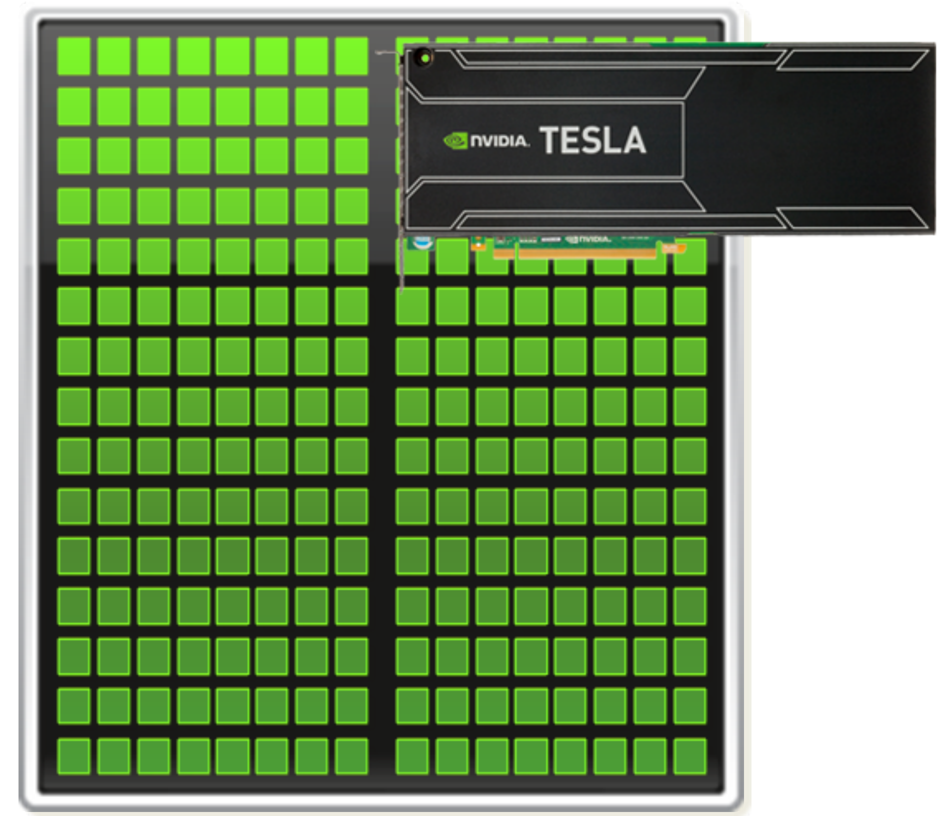


Intel Xeon E5-2680v4:

Clock speed: 2.4 GHz
4 instructions per cycle with AVX2
CPU - 28 cores

2.4 x 4 x 28 =
268.8 Gigaflops double precision

SCC GPU



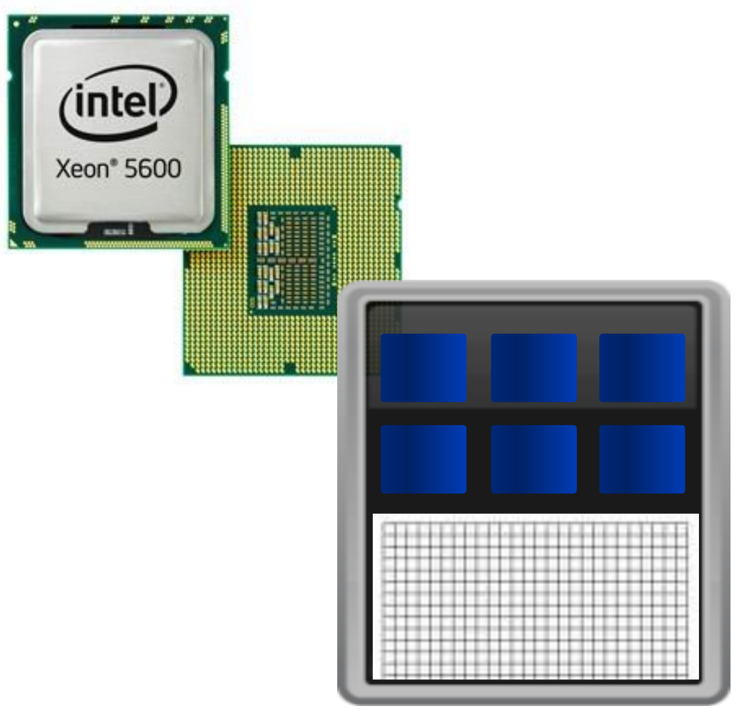
NVIDIA Tesla P100:

Single instruction per cycle
3584 CUDA cores

4.7 Teraflops double precision

CPU VS GPU

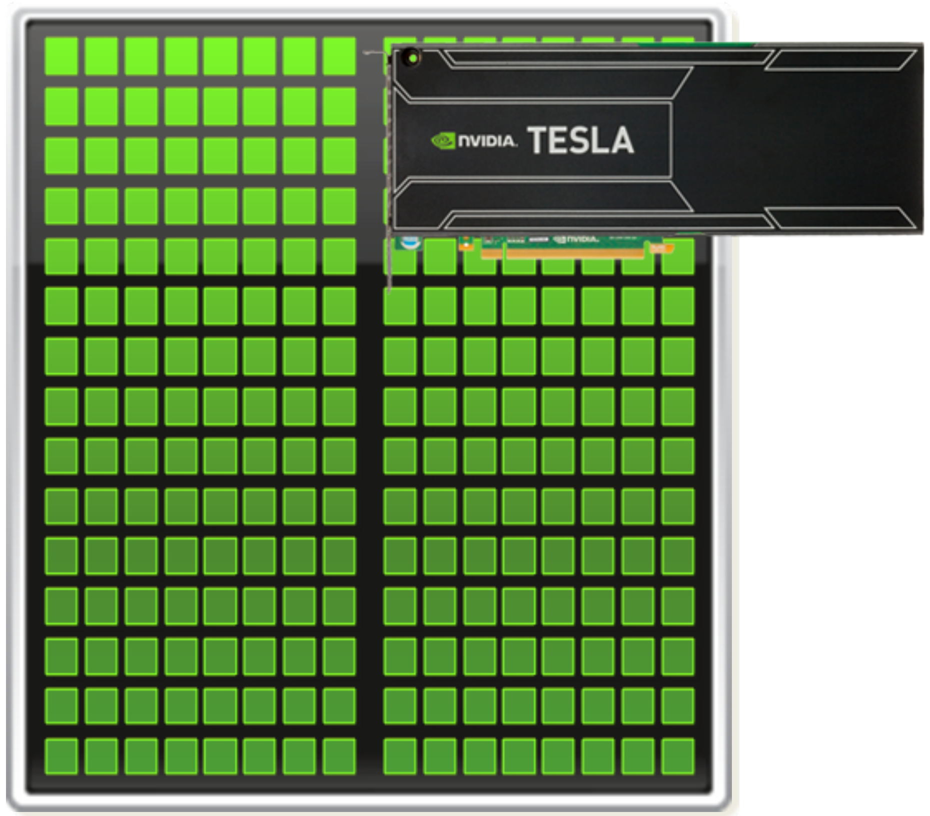
SCC CPU



Intel Xeon E5-2680v4 :

Memory size: 256 GB
Bandwidth: 76.8 GB/sec

SCC GPU



NVIDIA Tesla P100:

Memory size: 12GB total
Bandwidth: 549 GB/sec

Evolution

10x GPU Computing Growth

2008	2015
6,000 Tesla GPUs	**450,000** Tesla GPUs
150K CUDA downloads	**3M** CUDA downloads
77 Supercomputing Teraflops	**54,000** Supercomputing Teraflops
60 University Courses	**800** University Courses
4,000 Academic Papers	**60,000** Academic Papers

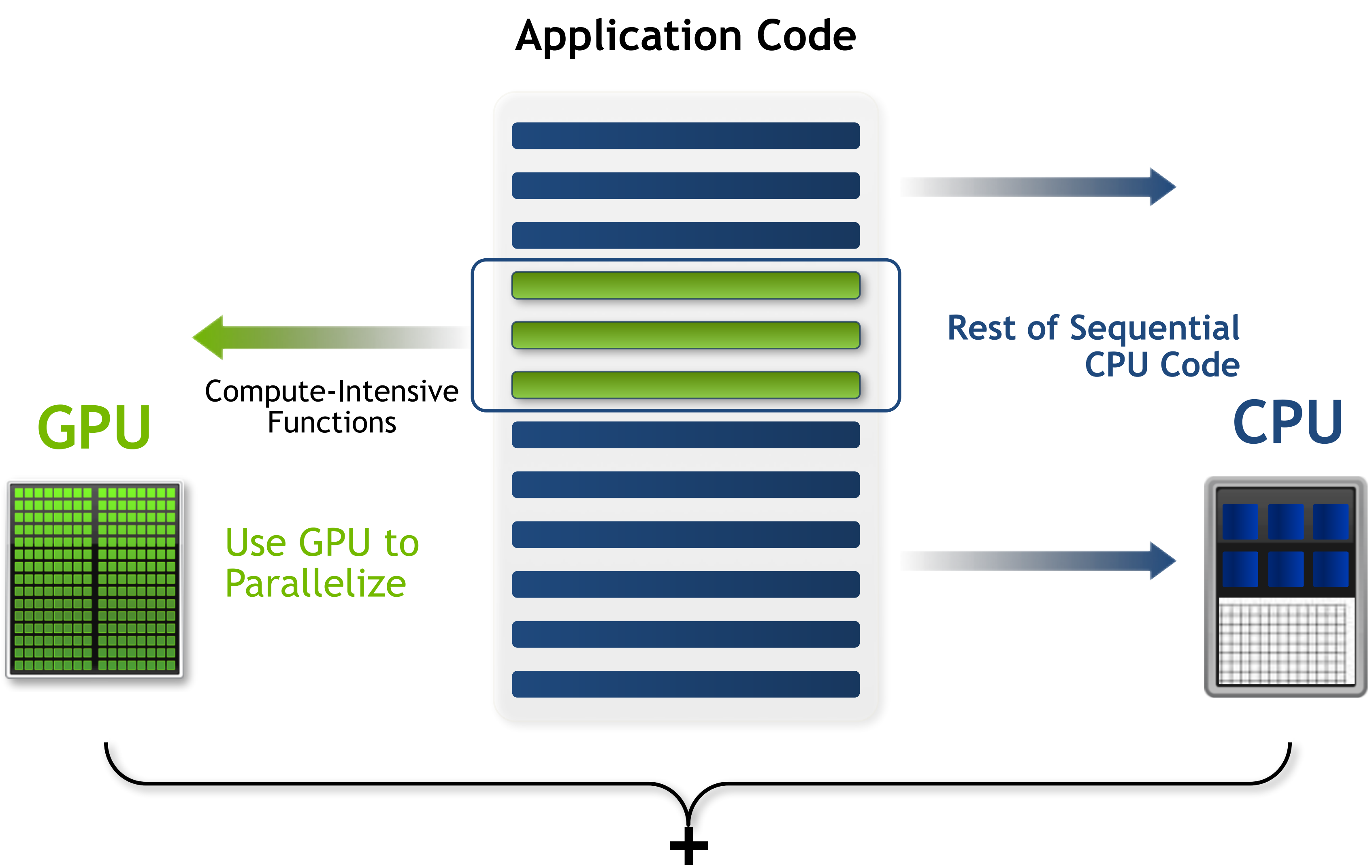
GPU Acceleration

Applications

GPU-accelerated libraries	OpenACC Directives	Programming Languages
Seamless linking to GPU-enabled libraries.	Simple directives for easy GPU-acceleration of new and existing applications	Most powerful and flexible way to design GPU accelerated applications
cuFFT, cuBLAS, Thrust, NPP, IMSL, CULA, cuRAND, etc.	PGI Accelerator	C/C++, Fortran, Python, Java, etc.

CPU + GPU

Minimum Change, Big Speed-up



The GPU ecosystem

C	OpenACC, CUDA
C++	Thrust, CUDA C++
Fortran	OpenACC, CUDA Fortran
Python	PyCUDA, PyOpenCL
Numerical analytics	MATLAB, Mathematica
Machine Learning	Theano, Tensorflow, Caffe, Torch, etc.

Will Execution on a GPU Accelerate My Application?

Yes if:

Computationally intensive—The time spent on computation significantly exceeds the time spent on transferring data to and from GPU memory.

Massively parallel—The computations can be broken down into hundreds or thousands of independent units of work.

Well suited to GPU architectures – some algorithms or implementations will not perform well on the GPU.

CUDA

- Compute Unified Device Architecture
- Developed by NVIDIA in 2007
- Native language to program GPUs,
    - Written in C
    - allows to communicate with the GPU through a dedicated driver
    - Has its own compiler: NVCC

CUDA

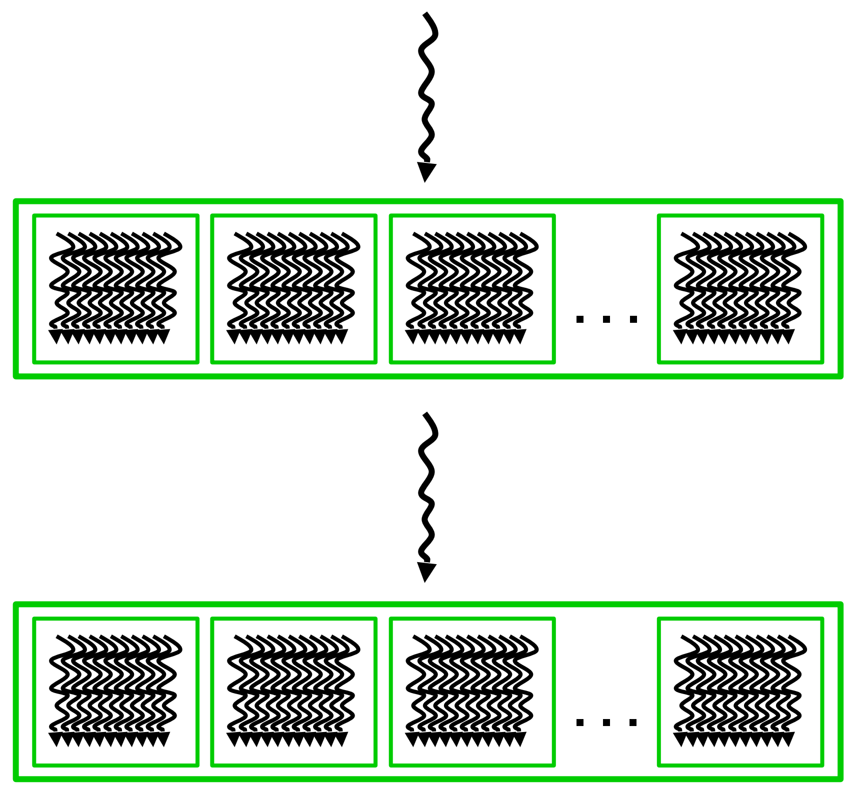
- Integrated host+device app C program
  - Serial or modestly parallel parts in host C code
  - Highly parallel parts in device SPMD kernel C code

Serial Code (host)

Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);

Serial Code (host)

Parallel Kernel (device)
KernelB<<< nBlk, nTid >>>(args);



CUDA devices and threads

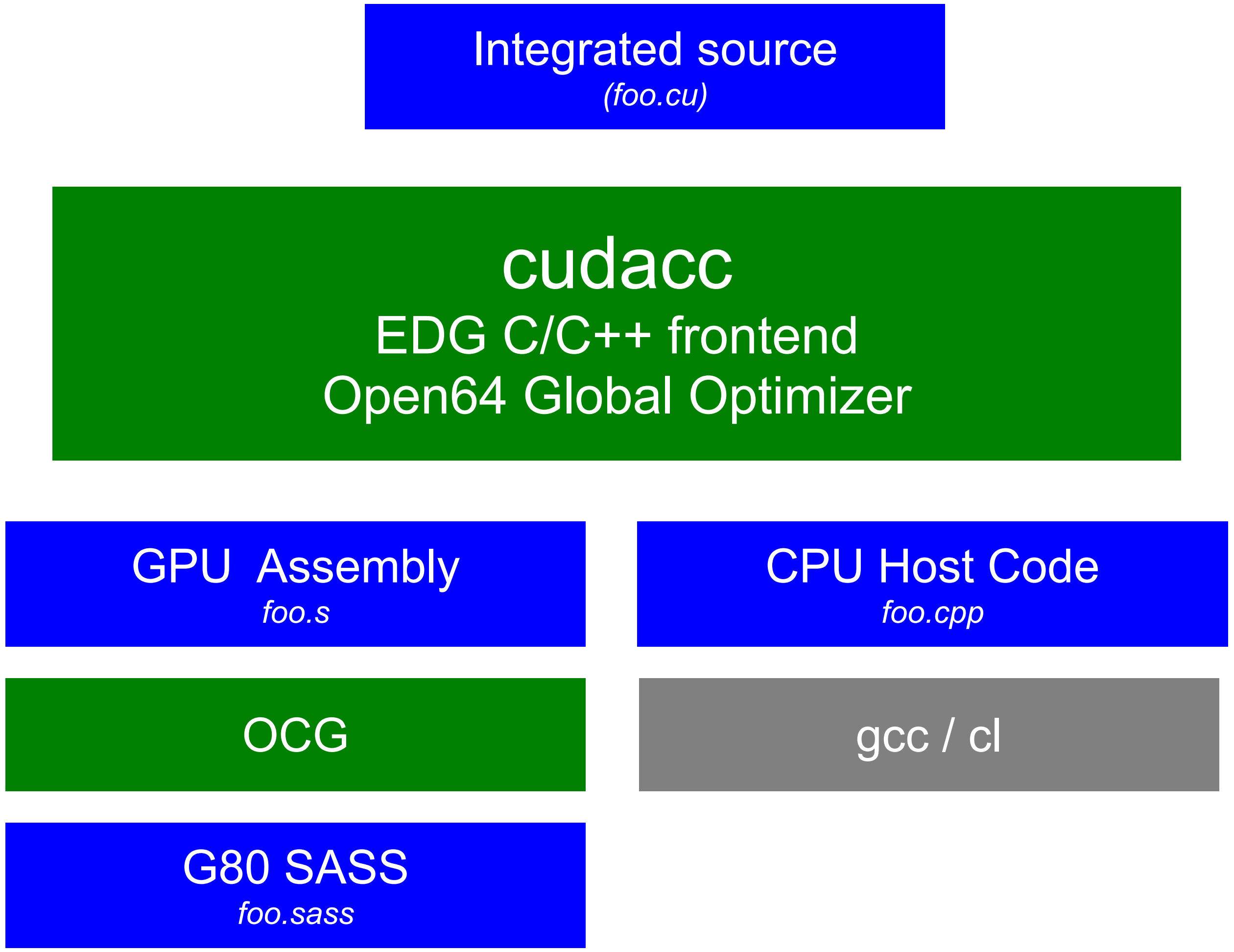
- A compute device
    - Is a coprocessor to the CPU or host
    - Has its own DRAM (device memory)
    - Runs many threads in parallel
    - Is typically a GPU but can also be another type of parallel processing device
- Data-parallel portions of an application are expressed as device **kernels** which run on many threads
- Differences between GPU and CPU threads
    - GPU threads are extremely lightweight
    - Very little creation overhead
    - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

CUDA is an extension of C

- **Declspecs**
  - **global, device, shared, local, constant**
- **Keywords**
  - **threadIdx, blockIdx**
- **Intrinsics**
  - **__syncthreads**
- **Runtime API**
  - **Memory, symbol, execution management**
- **Function launch**

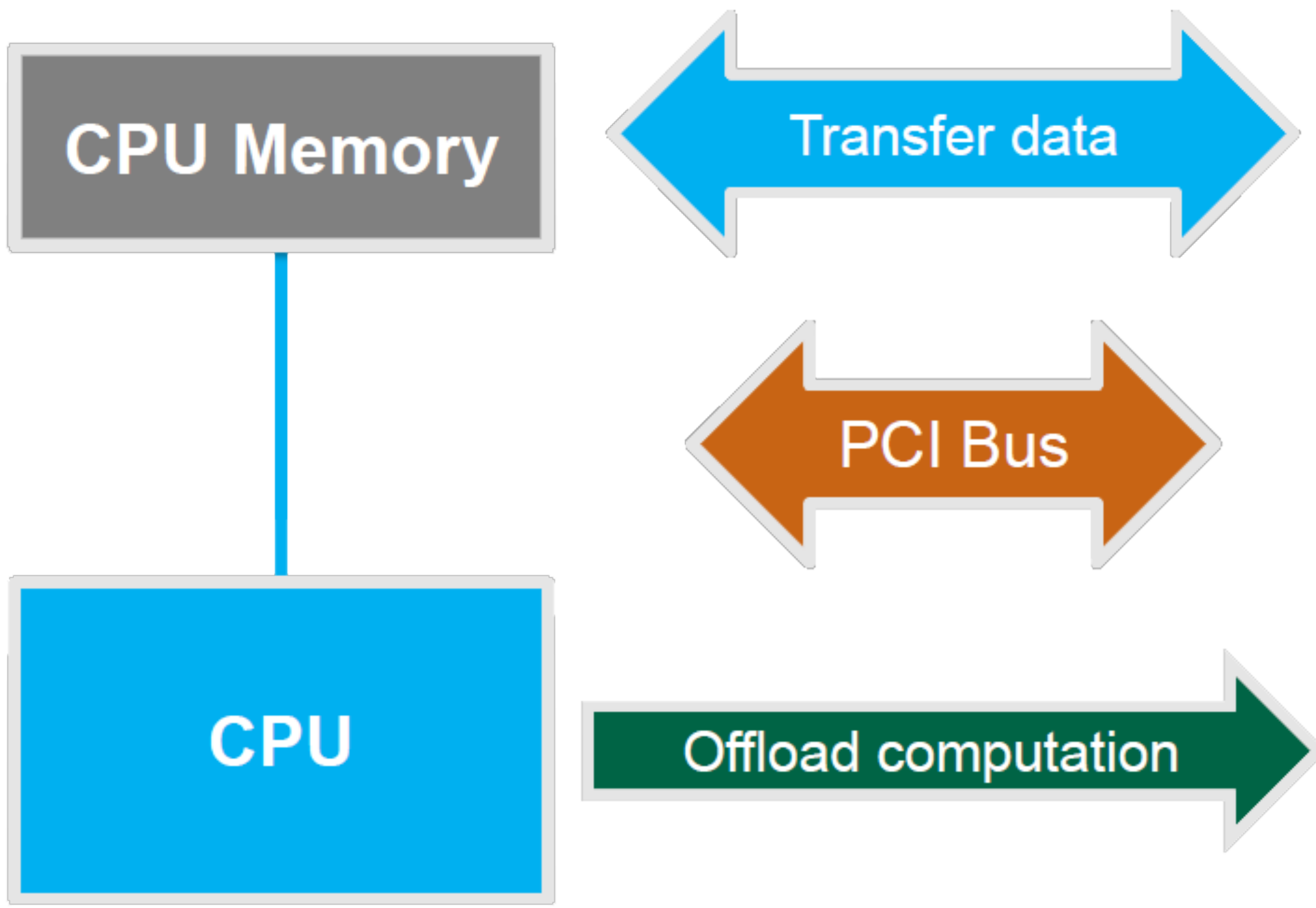
```
__device__ float filter[N];

__global__ void convolve (float *image)  {

  __shared__ float region[M];
  ...

  region[threadIdx] = image[i];

  __syncthreads()
  ...

  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)


// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

CUDA pipeline



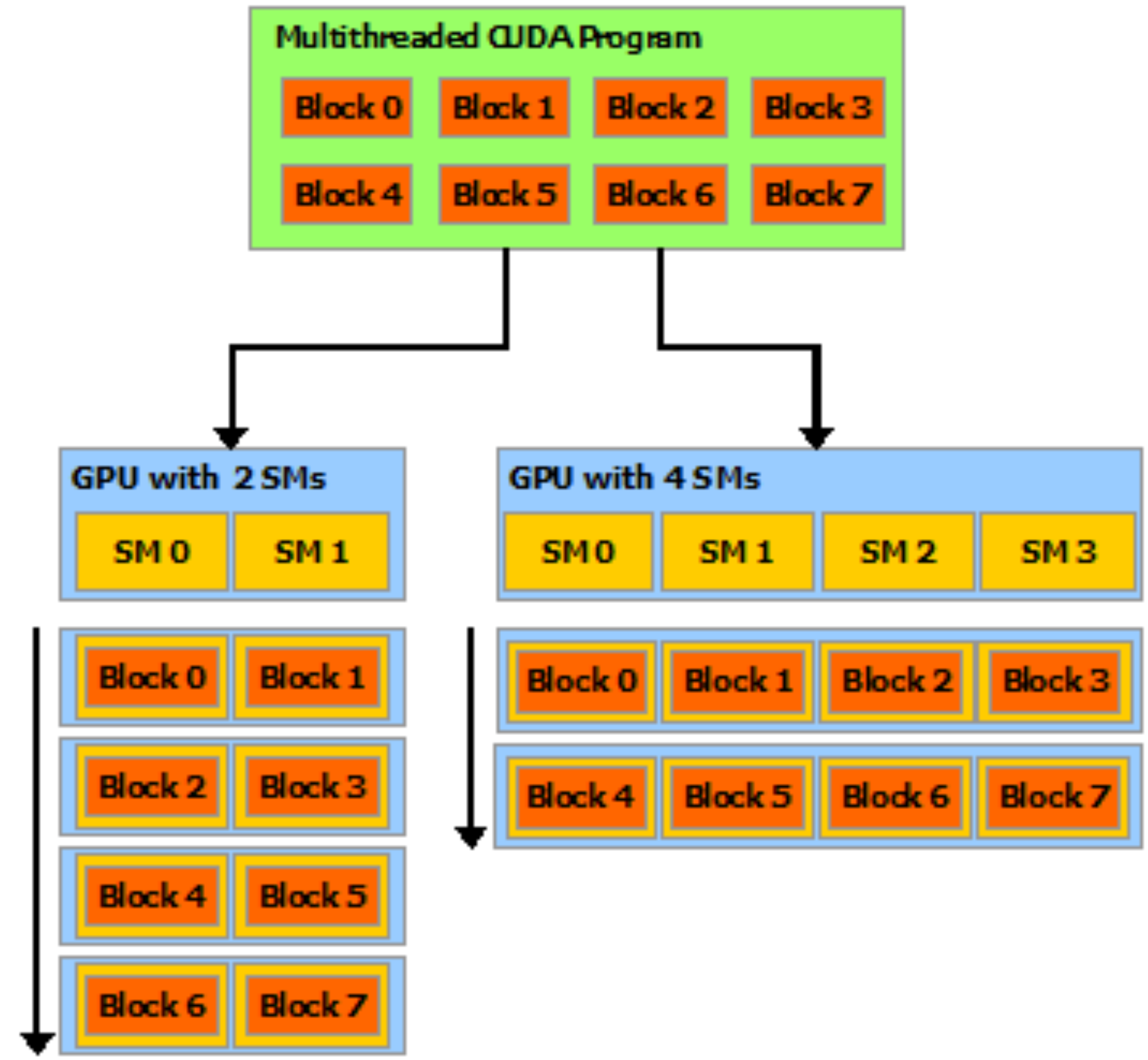
Mark Murphy, "NVIDIA's Experience with Open64,"
www.capsl.udel.edu/conferences/open64/2008/Papers/101.doc

Basic concepts of GPU programming



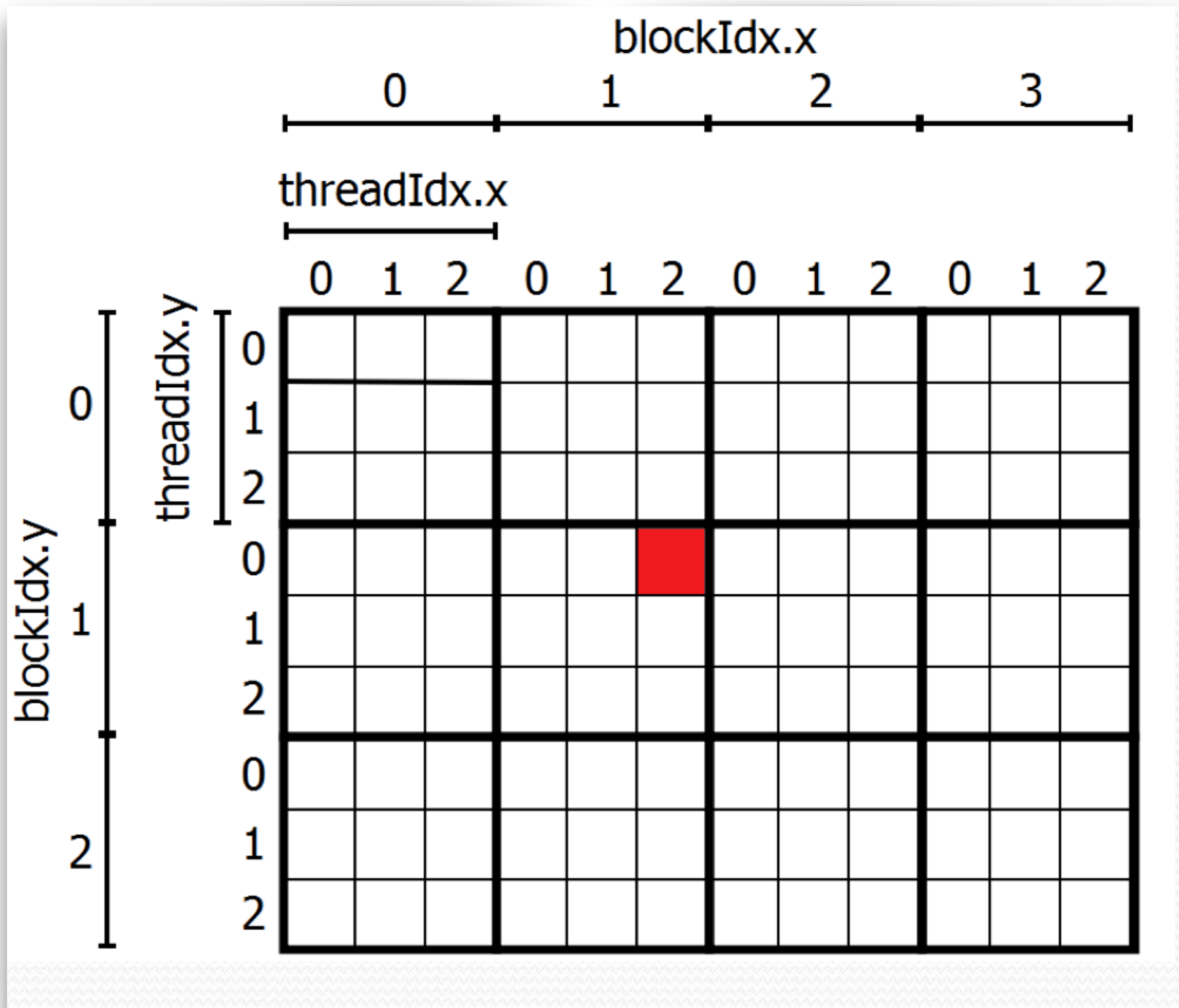
CUDA: **Blocks**, **Threads**, **Grids**, and more!

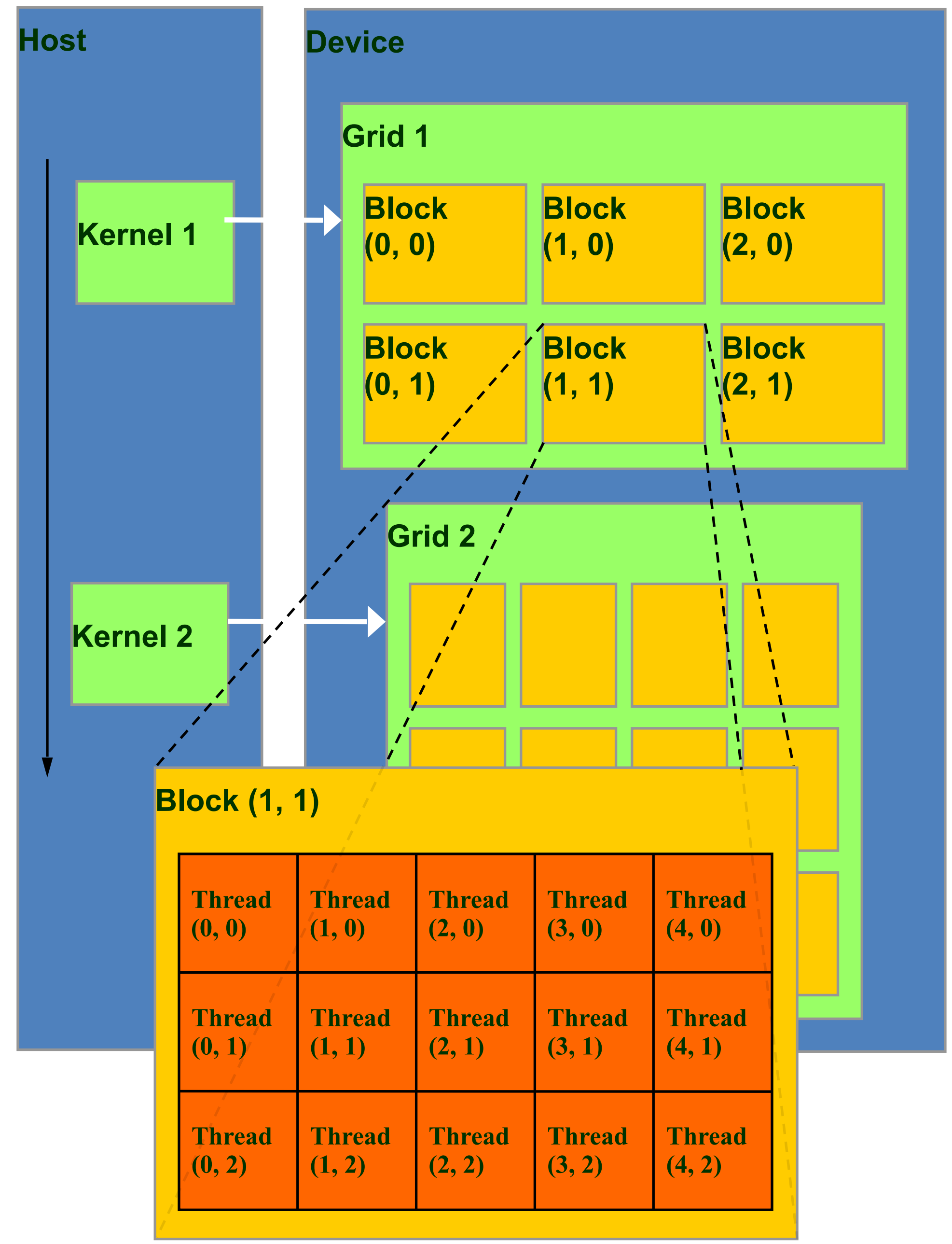
- Threads – Parallelized computations.
- Warp – A group of 32 threads.
- Blocks – Groups of threads arranged in 1, 2, or 3 dimensions assigned to a grid.
- Grids – The set of blocks in the computation, arranged in 1, 2, or 3 dimensions.
- SM – Streaming Multiprocessor. A set of CUDA cores that handle a block or a set of blocks.



http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#scalable-programming-model__automatic-scalability

CUDA Grid


blockIdx.x
0 1 2 3
threadIdx.x
0 1 2 0 1 2 0 1 2 0 1 2
blockIdx.y
0 1 2
threadIdx.y
0 1 2 0 1 2 0 1 2


Host
Device
Kernel 1
Kernel 2
Grid 1
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Grid 2
Block (1, 1)
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (4, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (4, 1)
Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)
Thread (4, 2)

CUDA to PyCUDA: understanding C code

- one way to use CUDA with Python is to embed CUDA C code directly into the Python code.

- Custom C code is how maximum control over the GPU operation is achieved.

- We’ll cover some CUDA C code approaches which also explain how to use the GPU before getting into PyCUDA.

‘Hello, world!’ Example

```
#define NUM_BLOCKS 4
#define BLOCK_WIDTH 8

/* Main function, executed on host (CPU) */
int main( void) {
     /* print message from CPU */
     printf( "Hello Cuda!\n" );
     /* execute function on device (GPU) */
     hello<<<NUM_BLOCKS, BLOCK_WIDTH>>>();
     /* wait until all threads finish their job */
     cudaDeviceSynchronize();
     /* print message from CPU */
     printf( "Welcome back to CPU!\n" );
     return(0);
}
```

Kernel:
A parallel function that runs on the GPU

A more complicated example

- We first try to do **addition of two vectors**

```
/* Main function, executed on host (CPU) */
int main( void) {

    /* 1. allocate memory on GPU */

    /* 2. Copy data from Host to GPU */

    /* 3. Execute GPU kernel */

    /* 4. Copy data from GPU back to Host */

    /* 5. Free GPU memory */

    return(0);
}
`
```

A more complicated example

- We first try to do **addition of two vectors**

```
/* Main function, executed on host (CPU) */
int main( void) {

    /* 1. allocate memory on GPU */

    /* 2. Copy data from Host to GPU */

    /* 3. Execute GPU kernel */

    /* 4. Copy data from GPU back to Host */

    /* 5. Free GPU memory */

    return(0);
}
```

```
/* 1. allocate memory on GPU */

float *d_A = NULL;
if (cudaMalloc((void **)&d_A, size) != cudaSuccess)
   exit(EXIT_FAILURE);

float *d_B = NULL;
cudaMalloc((void **)&d_B, size); /* For clarity we'll not check for err */

float *d_C = NULL;
cudaMalloc((void **)&d_C, size);
```

A more complicated example

- We first try to do **addition of two vectors**

```
/* Main function, executed on host (CPU) */
int main( void) {

    /* 1. allocate memory on GPU */

    /* 2. Copy data from Host to GPU */

    /* 3. Execute GPU kernel */

    /* 4. Copy data from GPU back to Host */

    /* 5. Free GPU memory */

    return(0);
}
```

```
/* 2. Copy data from Host to GPU */

cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);

cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

A more complicated example

- We first try to do **addition of two vectors**

```
/* Main function, executed on host (CPU) */
int main( void) {

    /* 1. allocate memory on GPU */

    /* 2. Copy data from Host to GPU */

    /* 3. Execute GPU kernel */

    /* 4. Copy data from GPU back to Host */

    /* 5. Free GPU memory */

    ret
}
```

```
/* 3. Execute GPU kernel */

/* Calculate number of blocks and threads */
int threadsPerBlock = 256;
int blocksPerGrid =(numElements + threadsPerBlock - 1) / threadsPerBlock;

/* Launch the Vector Add CUDA Kernel */
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);

/* Wait for all the threads to complete */
cudaDeviceSynchronize();
```

A more complicated example

- We first try to do **addition of two vectors**

```
/* Main function, executed on host (CPU) */
int main( void) {

    /* 1. allocate memory on GPU */

    /* 2. Copy data from Host to GPU */

    /* 3. Execute GPU kernel */

    /* 4. Copy data from GPU back to Host */

    /* 5. Free GPU memory */

    return(0);
}
`
```

```
/* 4. Copy data from GPU back to Host */

cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
```

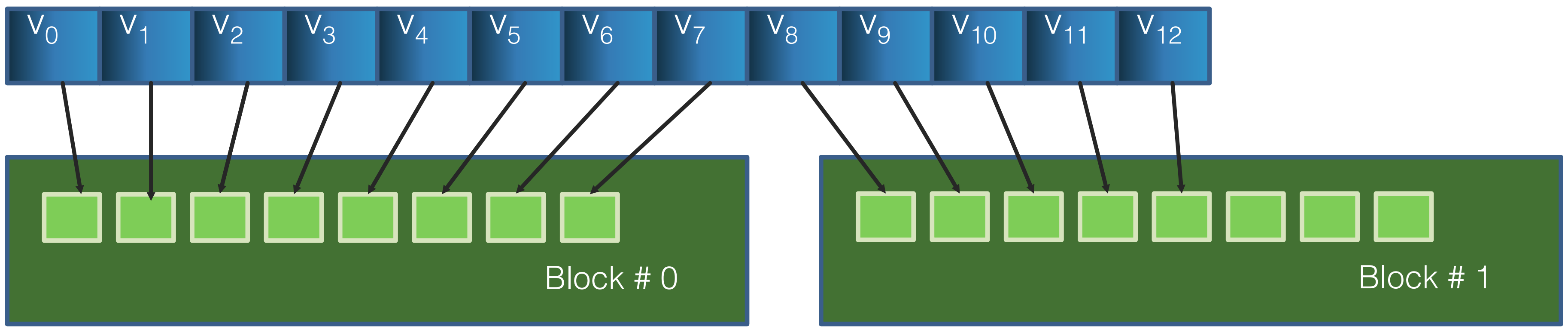
A more complicated example

- We first try to do **addition of two vectors**

```
/* Main function, executed on host (CPU) */
int main( void) {

    /* 1. allocate memory on GPU */

    /* 2. Copy data from Host to GPU */

    /* 3. Execute GPU kernel */

    /* 4. Copy data from GPU back to Host */

    /* 5. Free GPU memory */

    return(0);
}
`
```

```
/* 5. Free GPU memory */

cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
`
```

The ‘addition’ kernel




```
/* CUDA Kernel */
__global__ void vectorAdd( const float *A,
                           const float *B,
                                 float *C,
                                 int numElements) {

      /* Calculate the position in the array */
      int i = blockDim.x * blockIdx.x + threadIdx.x;

      /* Add 2 elements of the array */
      if (i < numElements) C[i] = A[i] + B[i];
}
```

PyCUDA

- Provides access to the CUDA API from Python
- Integrates with numpy, i.e. can automatically handle passing numpy arrays to and from the GPU.
- Full CUDA support.
- CUDA kernels are still C code that is embedded into the Python code.

https://mathema.tician.de/software/pycuda/

Vector element-wise product example

- Python code

```
import numpy

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)
dest = numpy.zeros_like(a)

for i in range(400):
        dest[i] = a[i]*b[i]
print(dest-a*b)
```

Vector element-wise product example

- PyCuda

```
import pycuda.driver as drv
import pycuda.autoinit
import numpy
from pycuda.compiler import SourceModule

mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")

# … the rest of the file continues
```

Initialization:
connection to GPU

Kernel:
Note that the function
is written in C

Compilation:
compile the function
and send code to GPU

Vector element-wise product example

- PyCuda

```
# … continuing the file
a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)
dest = numpy.zeros_like(a)

# mem allocation on gpu side
a_gpu = drv.mem_alloc(a.nbytes)
b_gpu = drv.mem_alloc(b.nbytes)
dest_gpu = drv.mem_alloc(dest.nbytes)

# data transfer to gpu # skipped
drv.memcpy_htod(a_gpu,a)
drv.memcpy_htod(b_gpu,b)

multiply_them(
        dest_gpu, a_gpu, b_gpu,
        block=(400,1,1))

# mem copy from gpu to cpu
drv.memcpy_dtoh(dest,dest_gpu)

print(dest-a*b)
```

Allocate GPU memory

Transfer to GPU memory

Run the kernel

Transfer to CPU memory

Vector element-wise product example

- A more simple way to handle memory transfer
    - automatic handling of the numpy array getting memory allocated on the GPU to store it, passing it to the GPU, and retrieving the result.
    - Slightly more overhead

```
# version 0
#multiply_them(
#        dest_gpu, a_gpu, b_gpu,
#        block=(400,1,1))

# version 1
multiply_them(
        drv.Out(dest), drv.In(a), drv.In(b),
        block=(400,1,1))
```

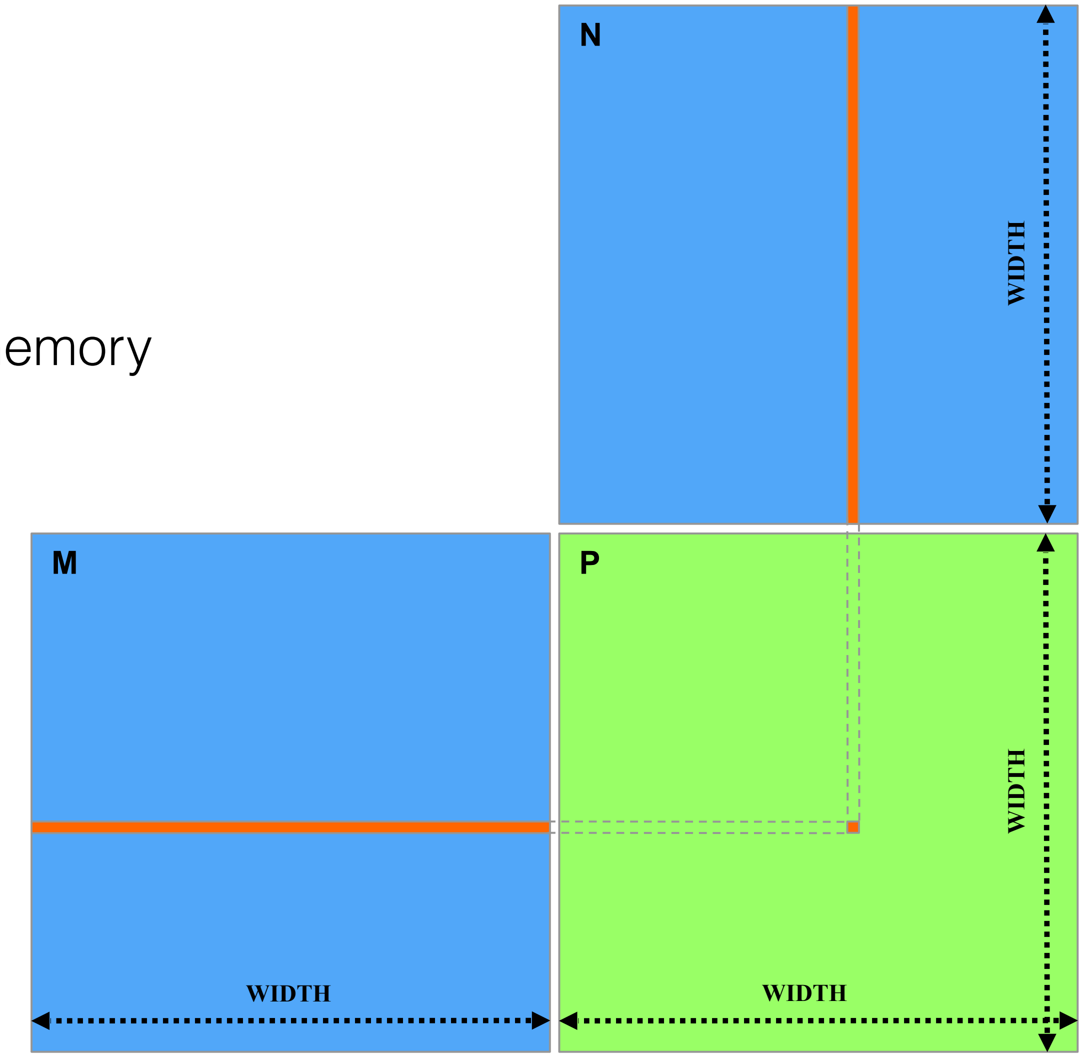
Vector element-wise product example

- A more simple way to handle memory transfer
    - Use the built-in **GPUArray**

```
import pycuda.gpuarray as gpuarray
import pycuda.driver as drv
import pycuda.autoinit
import numpy

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)
a_gpu = gpuarray.to_gpu(a)
b_gpu = gpuarray.to_gpu(b)

dest = (a_gpu*b_gpu).get()

print(dest-a*b)
```

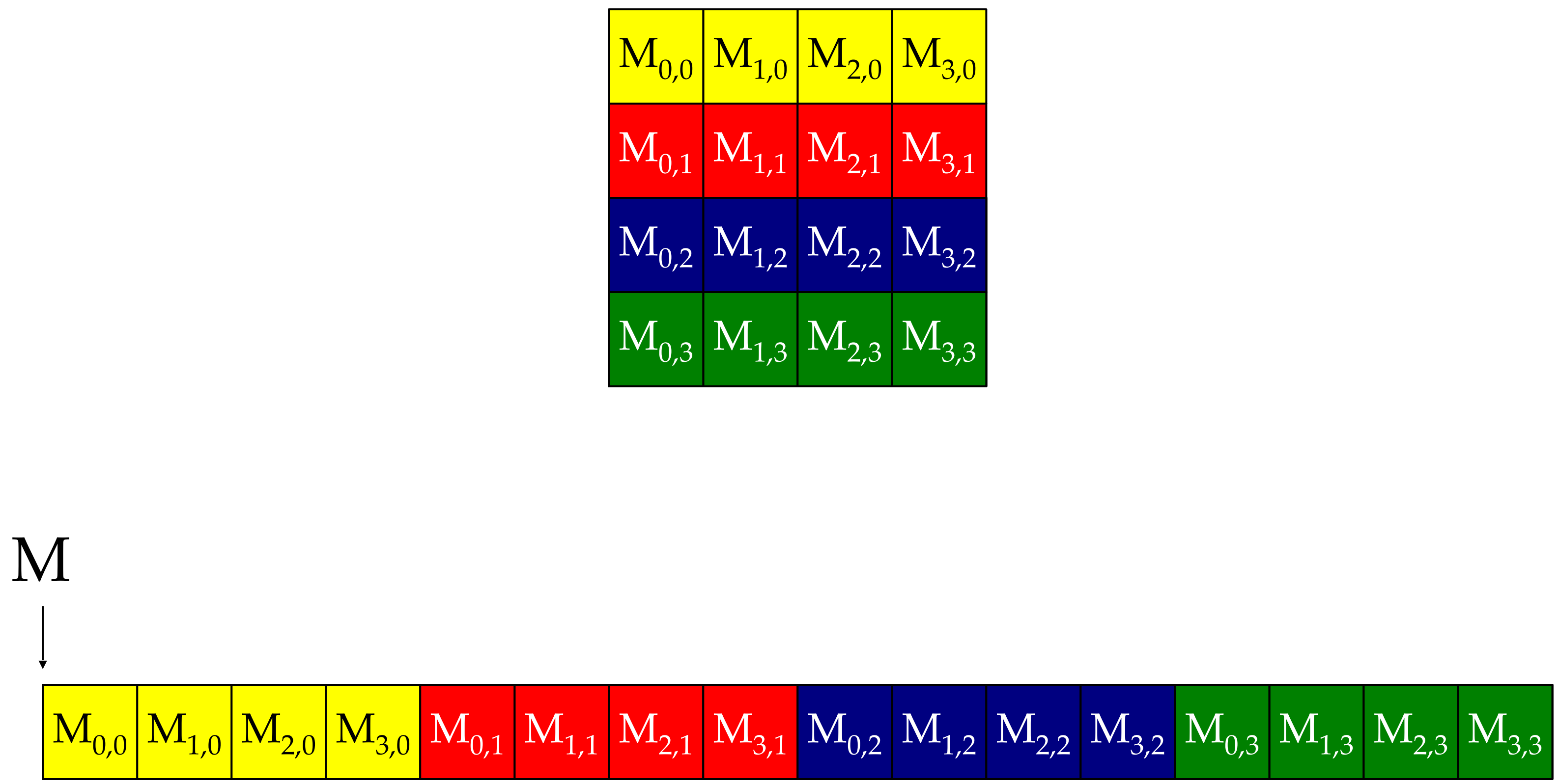
More example: matrix product

- P = M * N of size WIDTH x WIDTH
- Without tiling:
  - One thread calculates one element of P
  - M and N are loaded WIDTH times from global memory



More example: matrix product

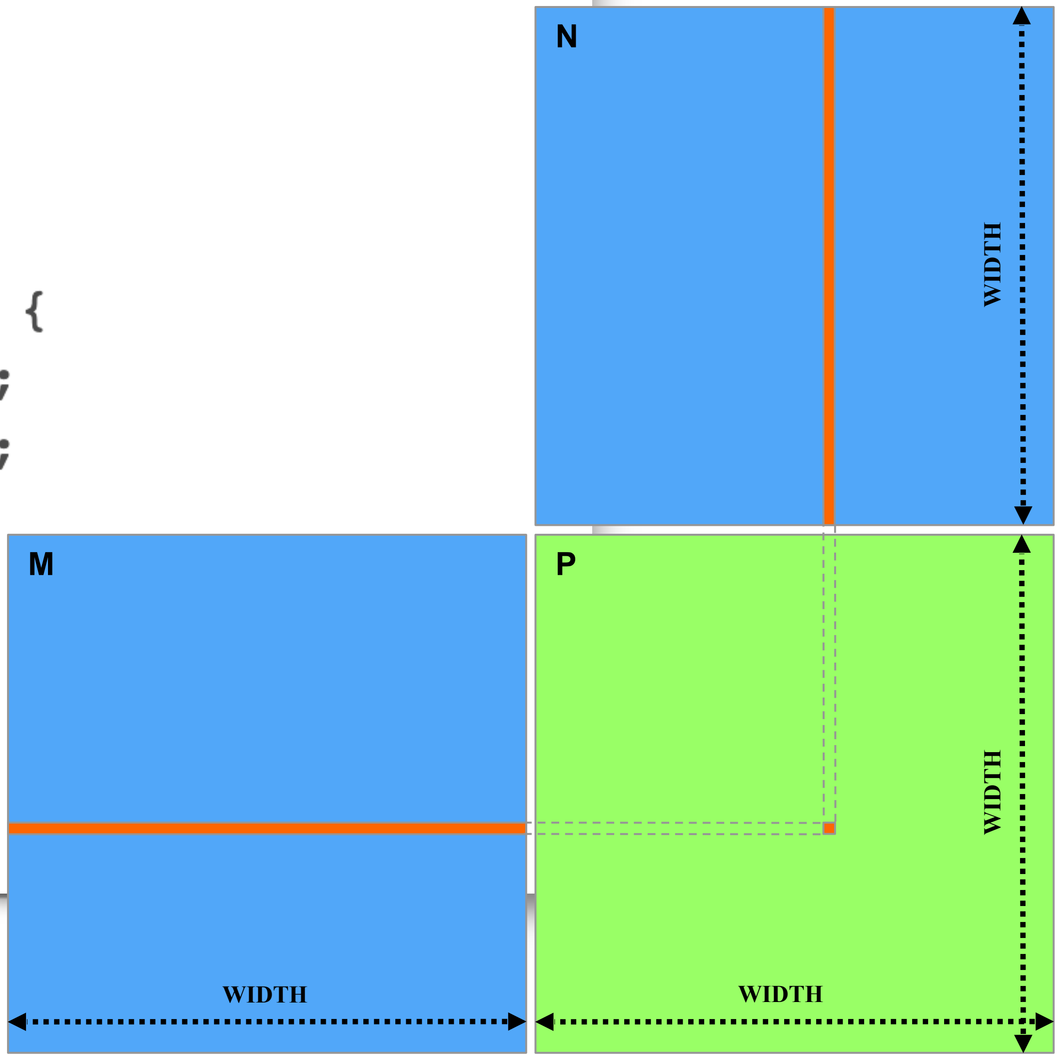
- How to represent a matrix in memory ?
    - Flattening operation



More example: matrix product

- Multiplication in C

```
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            double sum = 0;
            for (int k = 0; k < Width; ++k) {
                double a = M[i * width + k];
                double b = N[k * width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```



More example: matrix product

- Kernel in CUDA

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
  for (int k = 0; k < Width; ++k) {
       float Melement = Md[threadIdx.y*Width+k];
       float Nelement = Nd[k*Width+threadIdx.x];
       Pvalue += Melement * Nelement;
   }

  Pd[threadIdx.y*Width+threadIdx.x] = Pvalue;
}
```