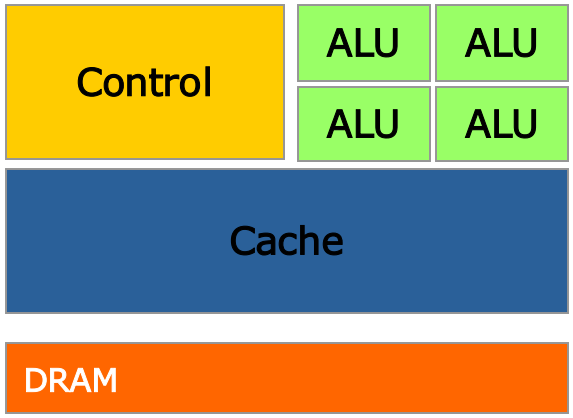
POLITECNICO DI MILANO

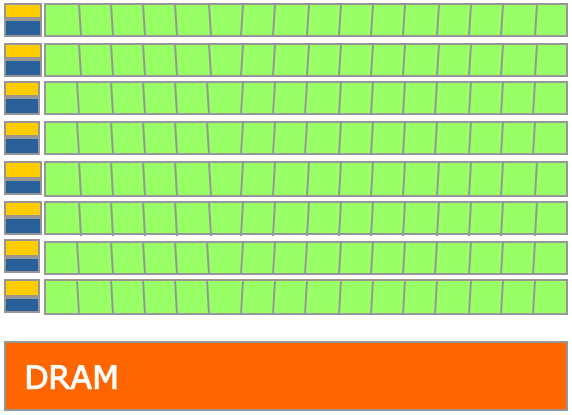
CUDA

Parallel Computing

Serena Curzel
Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria
serena.curzel@polimi.it

CPU vs GPU





CPU

- Powerful ALU
  - Reduced operation latency
- Large hierarchical caches
  - Fast cache access instead of slow memory access
- Sophisticated control
  - Branch prediction
  - Data forwarding

GPU

- Pipelined ALUs
  - Long latency, but high throughput
  - Rely on massive number of processing elements
- Small caches
  - High memory throughput
- Simple control
- Easy context switch

GPGPU computing

Massive Data Parallelism

Instruction Level Parallelism

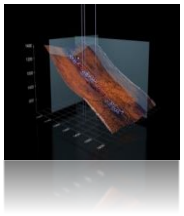
Graphics

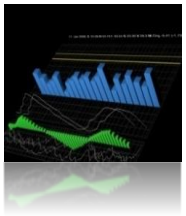
GPU
(Parallel Computing)

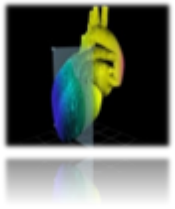
CPU
(Sequential Computing)

Data Fits in Cache

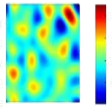
Larger Data Sets


Oil & Gas


Finance


Medical


Biophysics


Numerics


Audio

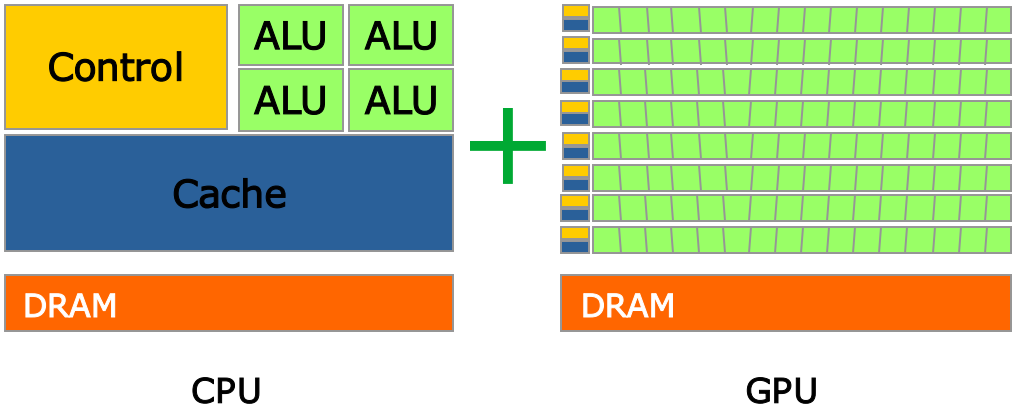

Video


Imaging

GPGPU computing

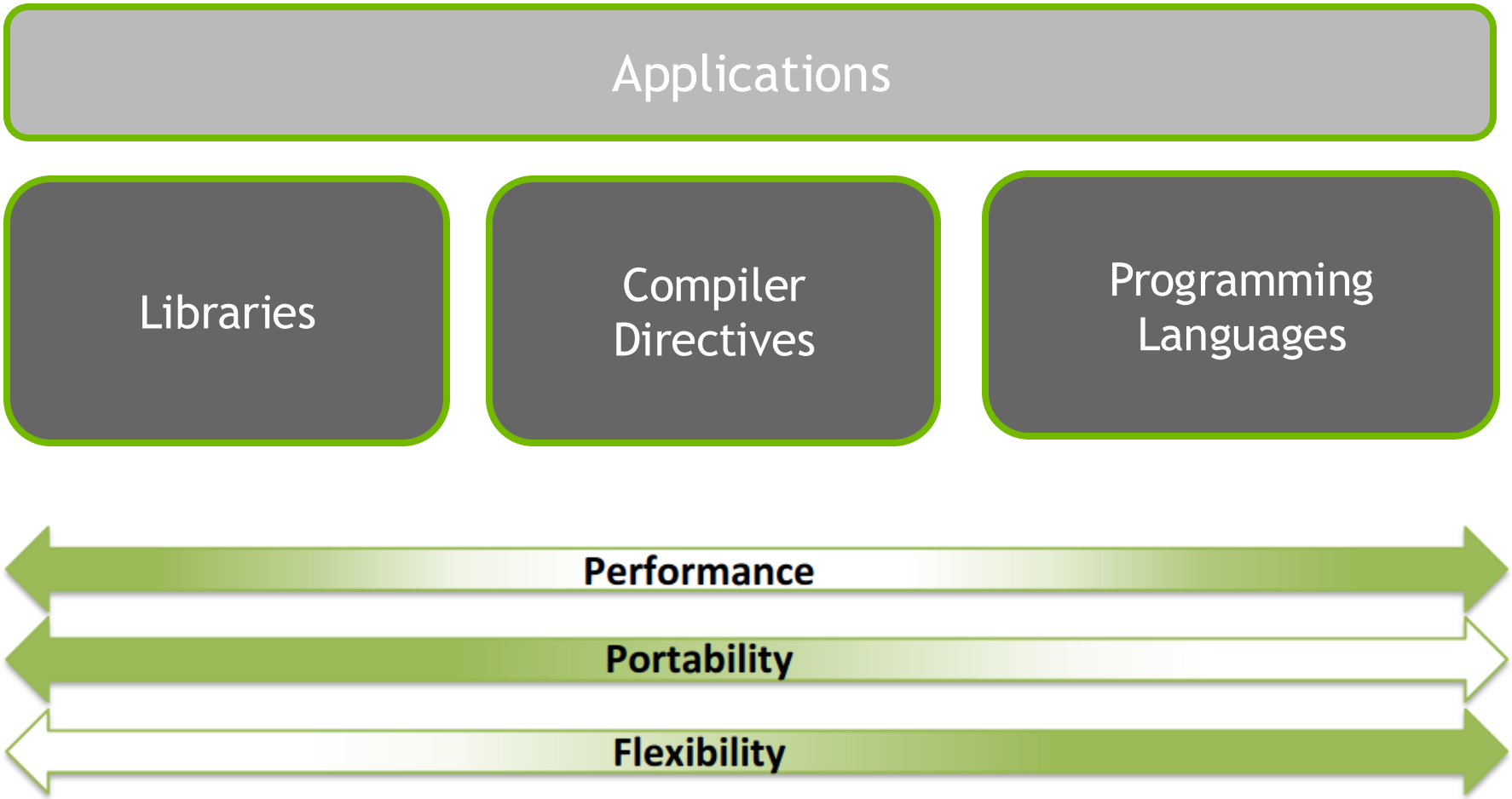
- The GPU (*device*) serves as a coprocessor for the CPU (*host*)
  - CPU and GPU are separate devices, with separate memory space addresses
  - The GPU has its own high-bandwidth memory
  - CPU and GPU should work together for maximum performance

- Low-latency access to cached data
- Out-of-order and speculative execution
- Serial or event-driven tasks



- High-throughput computation
- Hide memory latency
- Data-parallel tasks

GPGPU programming paradigms

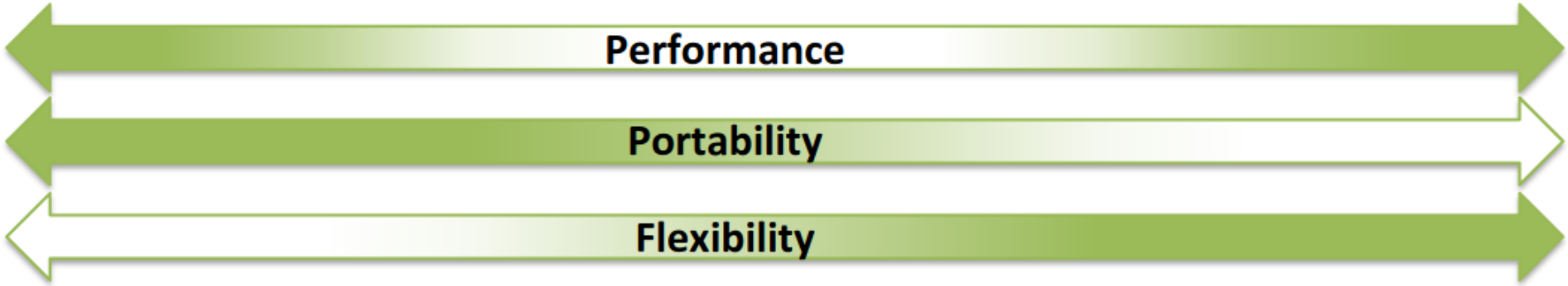


Libraries

Applications

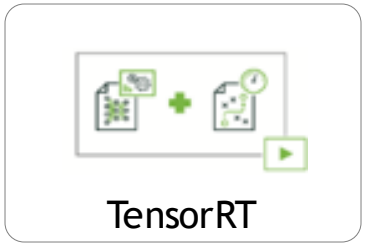
Libraries

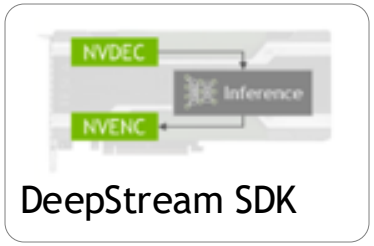
- Easy to use
- Most performance
- “Drop-in”
- High quality



Libraries

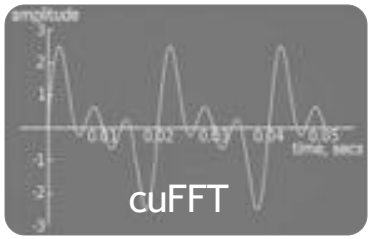
DEEP LEARNING



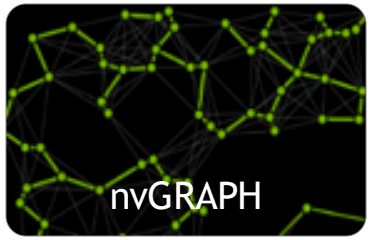


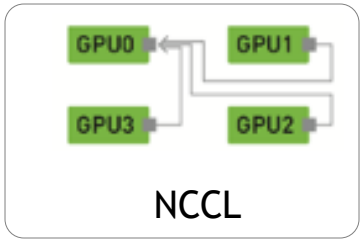
LINEAR ALGEBRA

SIGNAL, IMAGE, VIDEO



PARALLEL ALGORITHMS



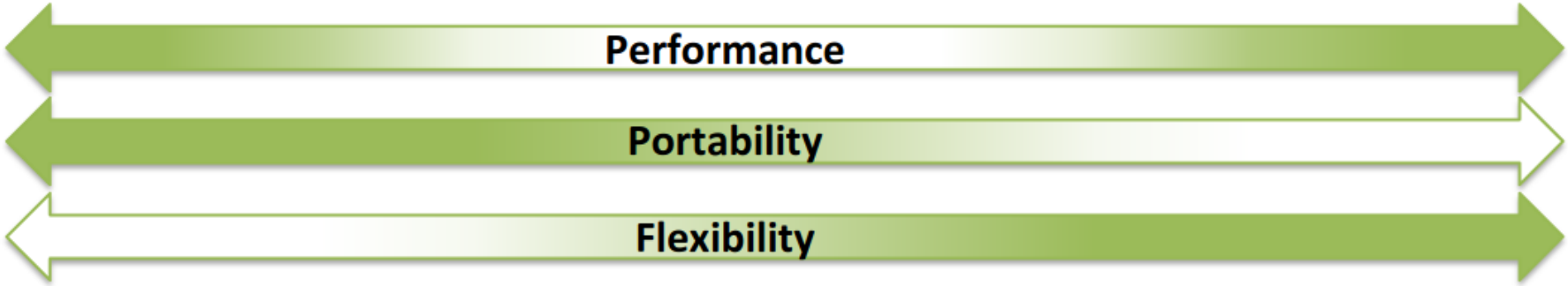


Compiler directives

Applications

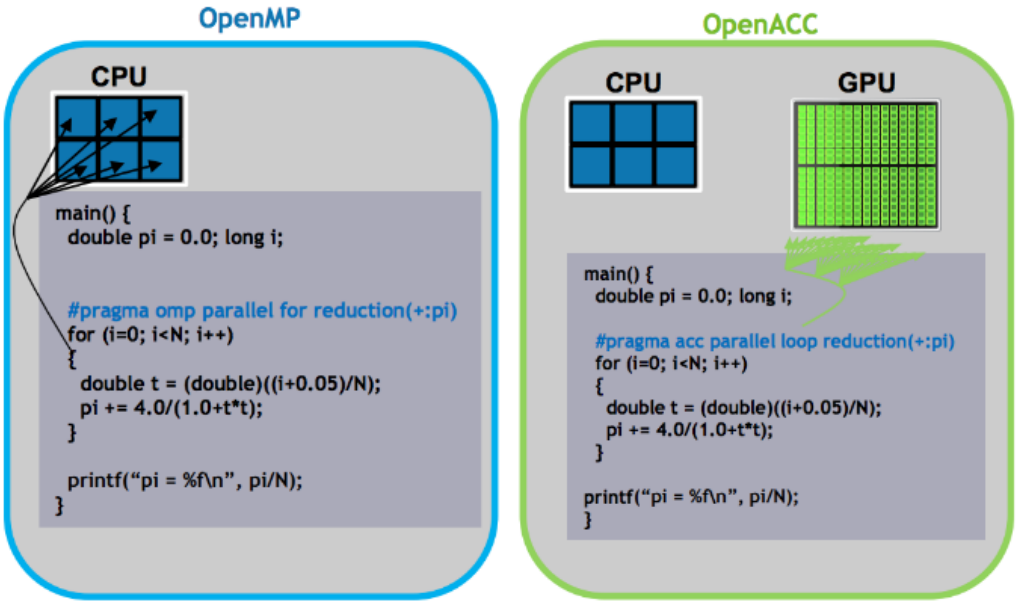
Compiler Directives

- Easy to use
- Portable code
- Uncertain performance



Compiler directives

- OpenACC example (very similar to OpenMP):



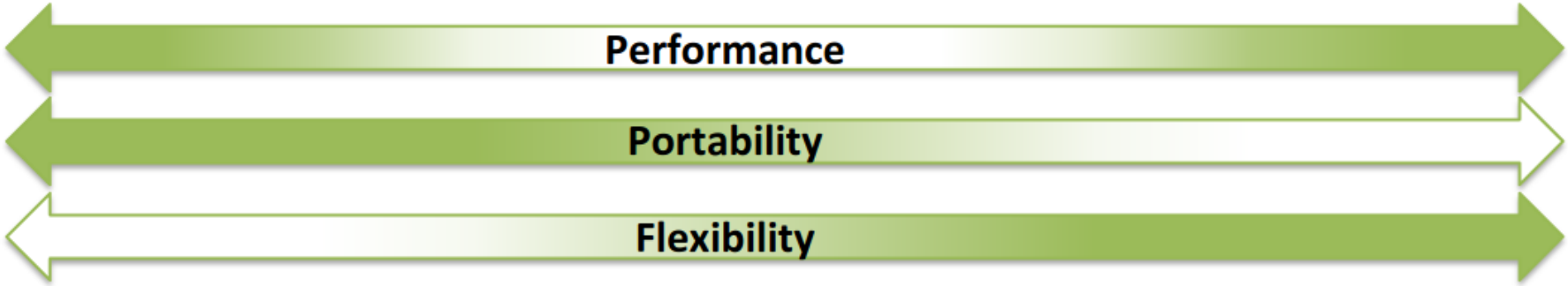

```
#pragma acc data copy(A), create(Anew)
while ( error > tol && iter < iter_max ) {
  error=0.0;

#pragma acc kernels loop gang(32), vector(16)
  for( int j = 1; j < n-1; j++) {
#pragma acc loop gang(16), vector(32)
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);
      error = max(error, abs(Anew[j][i] - A[j][i]);

    }
  }

#pragma acc kernels loop gang(16), vector(32)
  for( int j = 1; j < n-1; j++) {
#pragma acc loop
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

Programming Languages

Applications

- Most performance
- Most flexibility
- Verbose

Programming Languages

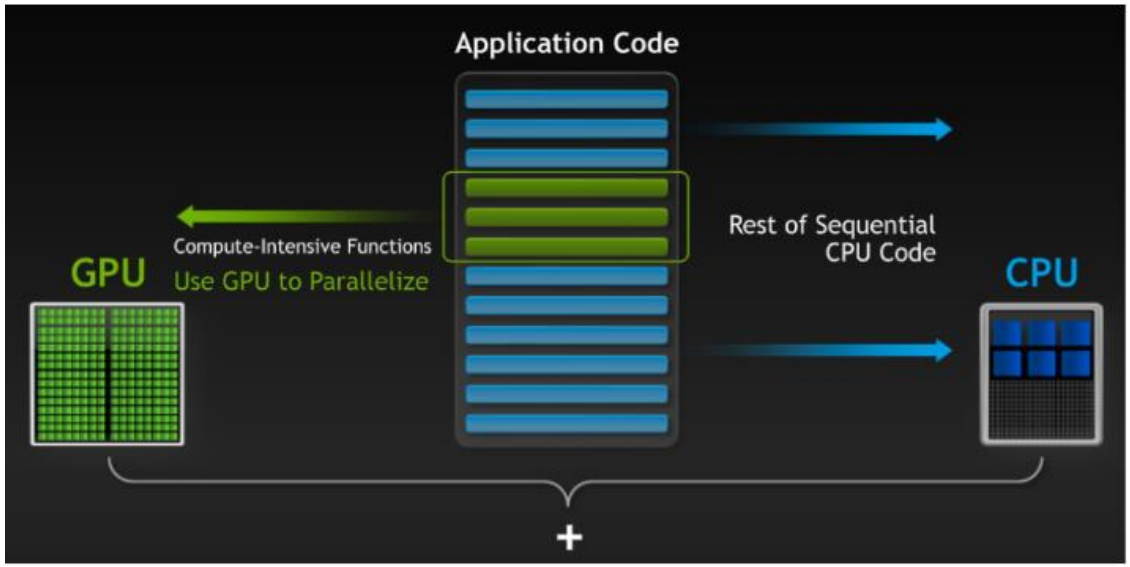


Programming Languages

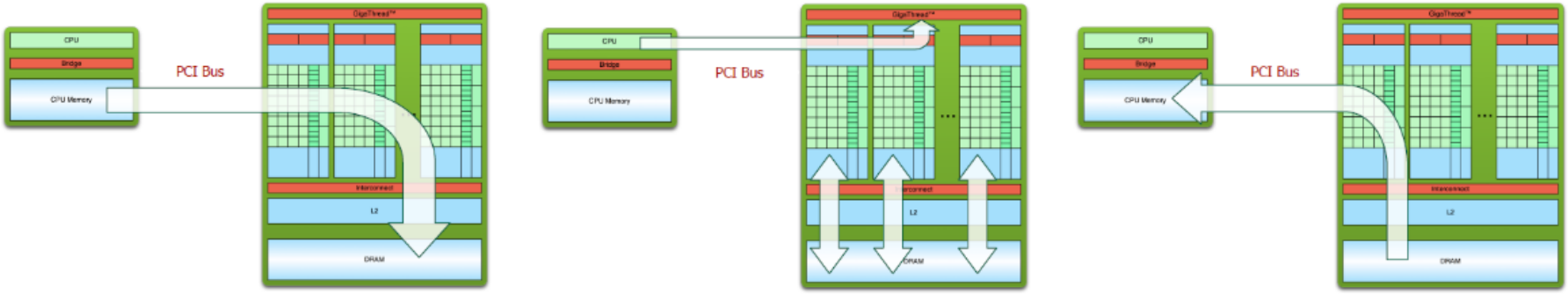
Numerical analytics	MATLAB, Mathematica, LabVIEW
Python	PyCUDA, Copperhead, Numba
Fortran	CUDA Fortran
C	CUDA C, OpenCL
C++	CUDA C++, OpenCL, SYCL, OneAPI
C#	Hybridizer

GPGPU programming model

- Serial parts of a program run on the CPU (host)
- *Computation-intensive* and *data-parallel* parts are offloaded to the GPU (device)
- Data is moved between device memory and host memory when needed



GPGPU programming model



1) Copy input data from CPU memory to GPU memory

2) Load GPU program and execute, caching data on chip for performance

3) Copy results from GPU memory to CPU memory

GPGPU programming model

- Data movement between CPU and GPU is the main bottleneck
  - Low bandwidth with respect to internal CPU and GPU since it exploits PCI Express (12-14GB/s)
  - Relatively high latency
  - *Data transfer can take more than the actual computation*
- This is especially a problem when porting CPU applications to GPGPU
  - Ignoring or not treating data movement seriously destroys GPU performance benefits
  - Some programming solutions may hide/automate data transfer

CUDA basics

- Compute Unified Device Architecture
- Parallel computing architecture and programming model that expose the computational horsepower of NVIDIA GPUs
- Has its own a C/C++/Fortran compiler
- Enables GPGPU computing with minimal effort:
  - Write a program for one thread
  - Instantiate it on many parallel threads
  - Low learning curve, no knowledge of graphics is required
- The programming model evolves to reflect the underlying hardware architecture

CUDA basics

- A function which runs on a GPU is called kernel
  - When a kernel is launched on a GPU thousands of threads will execute its code
  - The programmer decides the number of threads
  - Each thread acts on different data elements independently (SIMD/SPMD/SIMT parallelism)

```
void vsum(int* a, int* b, int* c){
  int i;
  for (i=0;i<N;i++){
    c[i]=a[i]+b[i];
  }
}

void main(){
   int va[N], vb[N], vc[N];
   ...
   vsum(va,vb,vc);
   ...
}
```

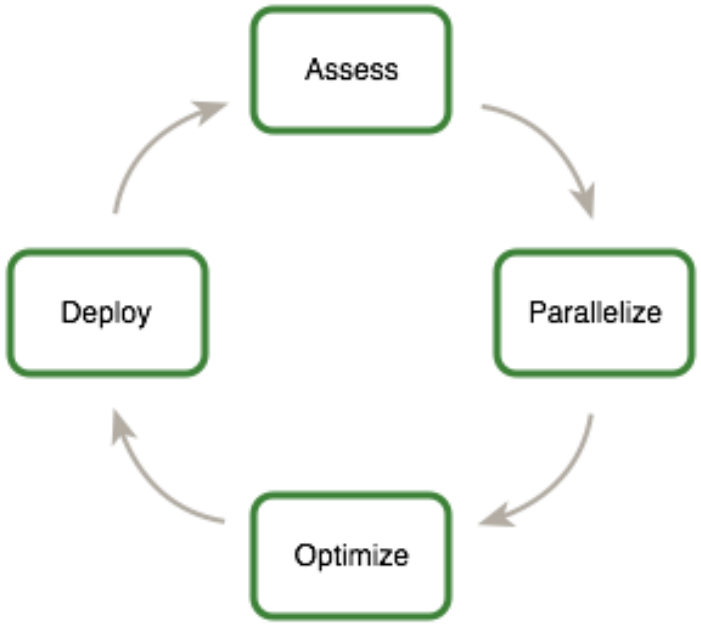
C program

```
__global__
void vsum(int* a, int* b, int* c){
  int i = ... // get unique thread ID
  c[i]=a[i]+b[i];
}

void main(){
   int va[N], vb[N], vc[N];
   ...
   vsum<<<1, 1>>>(va,vb,vc);
   ...
}
```

CUDA C program

GPGPU best practices

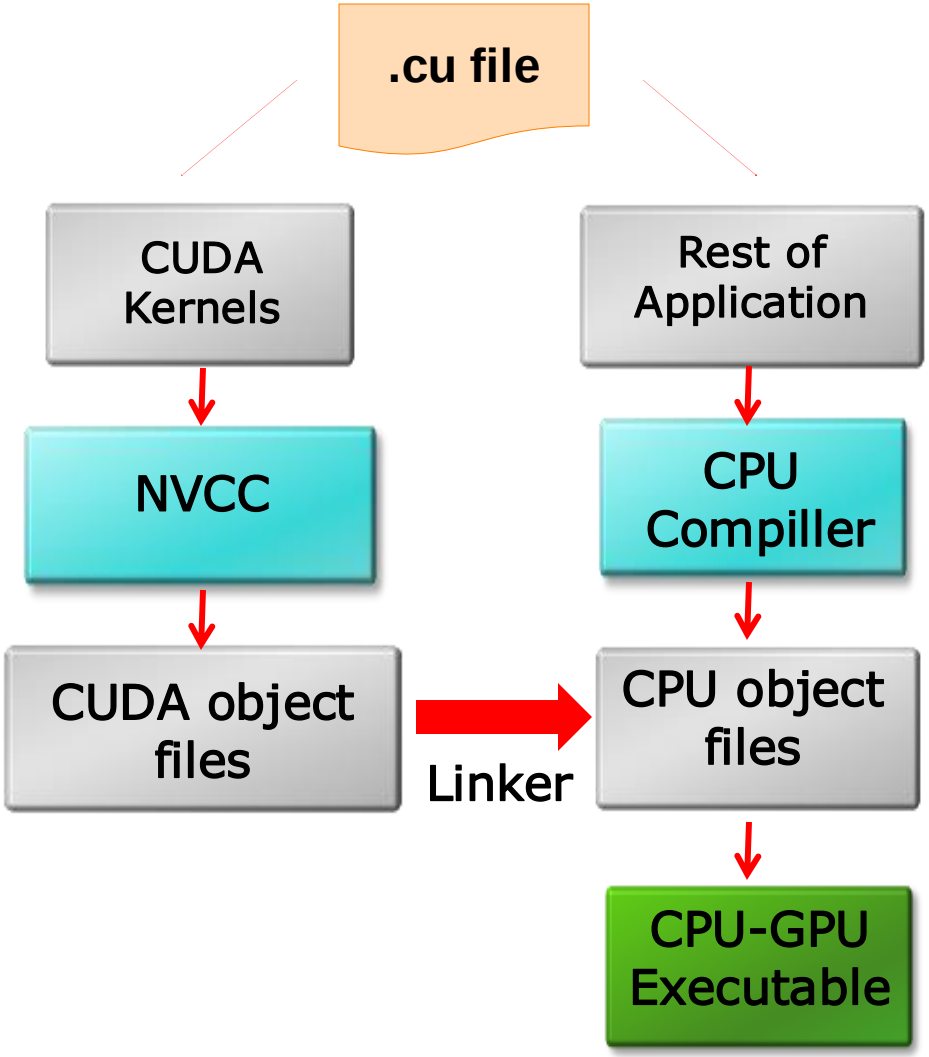
- NVIDIA guide: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/
- Suggested cyclical process to accelerate an application with NVIDIA GPUs:
  - Assess
  - Parallelize
  - Optimize
  - Deploy



GPGPU best practices

- Assess: locate bottlenecks and estimate how much the application can benefit from parallelization
- Parallelize: apply libraries, compiler directives or CUDA
- Optimize: apply optimizations and measure performance improvements
- Deploy: compare with performance estimations and move to production

Compilation



Volta virtual architecture

```
nvcc -arch=sm_70 -o hello-gpu
01-hello/01-hello-gpu.cu -run
```

Compile and execute

Compilation

- Useful compilation flags:
  - -x: treat all input files as .cu files
  - -Xcompiler: pass a host compiler flag that is not supported by NVCC
  - -g: include host debugging symbols
  - -G: include device debugging symbols
  - -lineinfo: include line information with symbols

Debugging

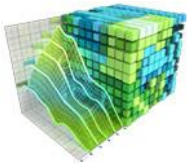
- cuda-memcheck
  http://docs.nvidia.com/cuda/cuda-memcheck
  - Memory leaks
  - Memory errors (OOB, misaligned access, illegal instruction, etc)
  - Race conditions
  - Illegal Barriers
  - Uninitialized Memory
- cuda-gdb http://docs.nvidia.com/cuda/cuda-gdb
  - Extension of GDB

Profiling

NVPROF

```
==20561== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 49.88%  866.69ms    504758  1.7170us  1.5840us  2.8160us  void th
int, thrust::detail::device_generate_functor<thrust::detail::fill_
 25.33%  440.05ms    252602  1.7410us  1.5360us  2.3680us  void th
t, thrust::detail::device_generate_functor<thrust::detail::fill_fu
 17.07%  296.60ms       200  1.4830ms  1.2840ms  1.7253ms  kerComp
  2.98%  51.819ms       200  259.09us  246.97us  264.83us  kerMake
  1.16%  20.173ms       501  40.265us     928ns  17.677ms  [CUDA m
  0.93%  16.198ms       200  80.991us  71.840us  90.751us  kerColl
  0.73%  12.636ms       400  31.589us  14.720us  50.432us  [CUDA m
  0.69%  12.075ms       200  60.376us  59.680us  62.304us  kerNorm
  0.63%  10.993ms       200  54.963us  52.608us  58.208us  kerMake
  0.32%  5.5524ms       200  27.761us  22.559us  33.152us  [CUDA m
  0.12%  2.1342ms         1  2.1342ms  2.1342ms  2.1342ms  void th
```

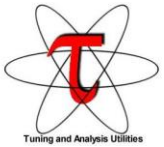
NVVP



NSIGHT

NVIDIA Provided

TAU



VampirTrace

3rd Party

Nsight command-line profiler

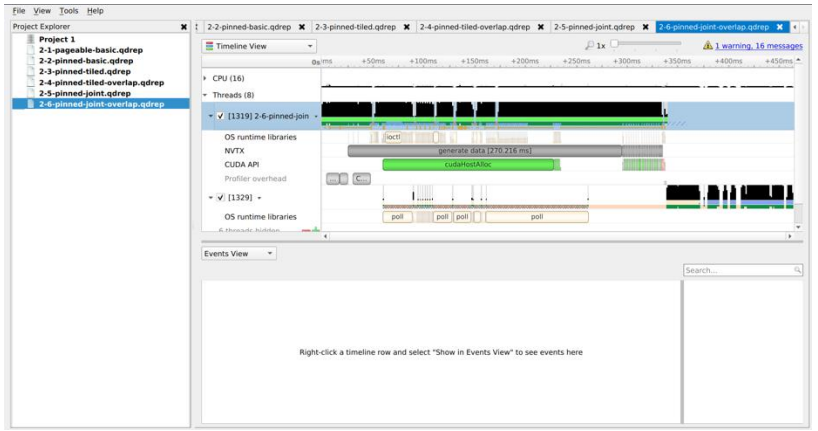
- Nsight Systems' **nsys** tool can used to profile an accelerated application
- nsys launches a CUDA application and reports information about
  - GPU activity
  - CUDA API calls
  - Memory activity

```
CUDA API Statistics:

 Time(%)  Total Time (ns)  Num Calls    Average      Minimum     Maximum           Name
 -------  ---------------  ---------  ------------  ----------  ----------  ---------------------
    90.9       2351076956          1  2351076956.0  2351076956  2351076956  cudaDeviceSynchronize
     8.3        215286867          3    71762289.0       18524   215210818  cudaMallocManaged
     0.7         18876773          3     6292257.7     5580127     7479496  cudaFree
     0.0            52646          1       52646.0       52646       52646  cudaLaunchKernel


CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances    Average      Minimum     Maximum                       Name
 -------  ---------------  ---------  ------------  ----------  ----------
-----------------------------------------
   100.0       2351062696          1  2351062696.0  2351062696  2351062696  addVectorsInto(float*, float*,
int)


CUDA Memory Operation Statistics (by time):

 Time(%)  Total Time (ns)  Operations  Average  Minimum  Maximum             Operation
 -------  ---------------  ----------  -------  -------  -------  ---------------------------------
    76.6         68333344        2304  29658.6     1887   181408  [CUDA Unified Memory memcpy HtoD]
    23.4         20904640         768  27219.6     1119   159776  [CUDA Unified Memory memcpy DtoH]


CUDA Memory Operation Statistics (by size in KiB):

   Total     Operations  Average  Minimum  Maximum              Operation
 ----------  ----------  -------  -------  --------  ---------------------------------
 393216.000        2304  170.667    4.000  1020.000  [CUDA Unified Memory memcpy HtoD]
 131072.000         768  170.667    4.000  1020.000  [CUDA Unified Memory memcpy DtoH]
```

Nsight Systems GUI


```
nsys profile --stats=true -o vector-add-no-prefetch-report ./vector-add-no-prefetch
```

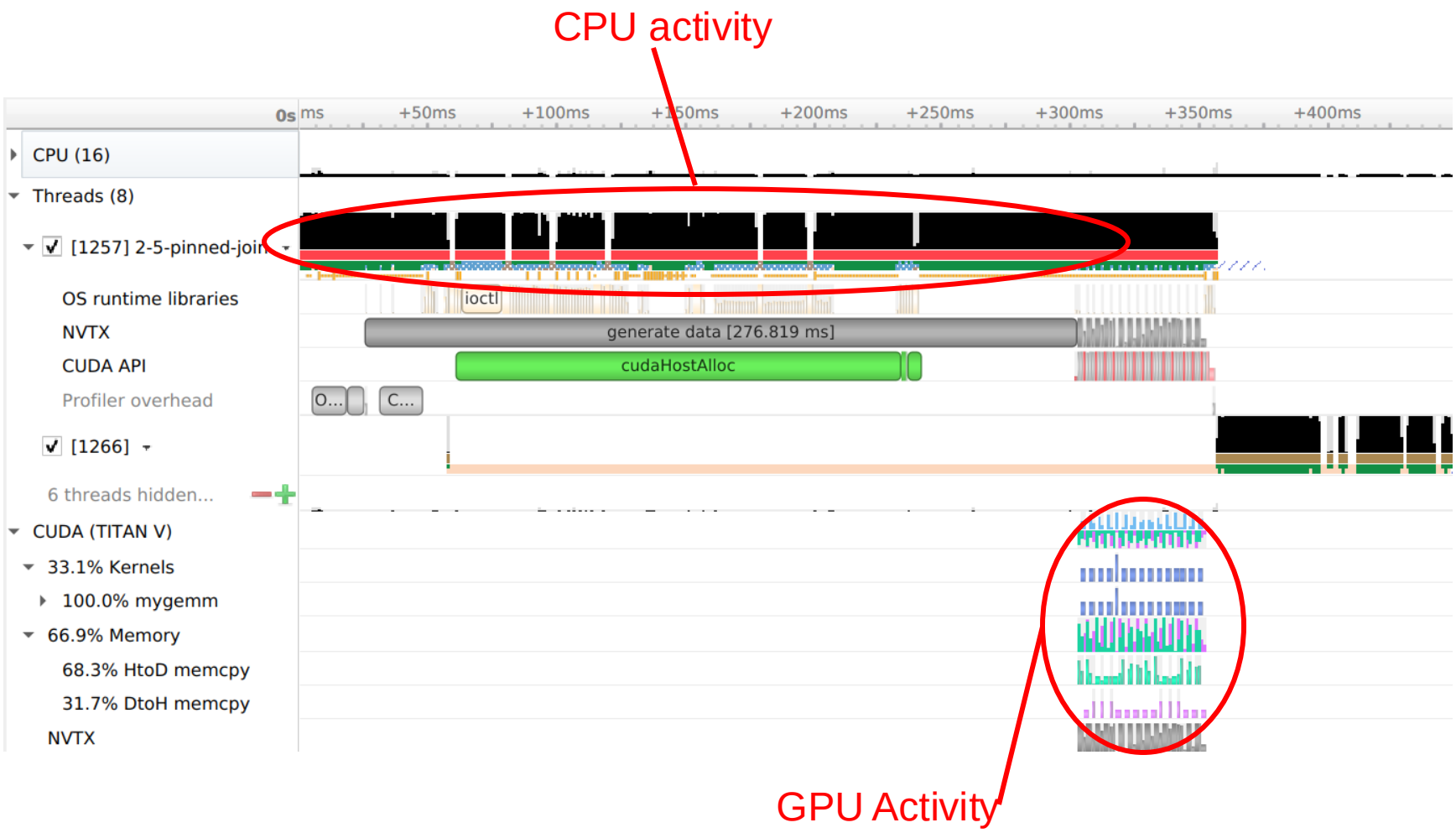
Record application statistics with nsys



Analyze report with Nsight Systems

Nsight Systems

- Timeline wiew:



Nsight Systems

NVIDIA Systems Management Interface

- nvidia-smi supports monitoring and management of GPU devices

```
Mon May 10 14:21:42 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.33.01    Driver Version: 440.33.01    CUDA Version: 10.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla T4            On   | 00000000:00:1E.0 Off |                    0 |
| N/A   25C    P8     9W /  70W |      0MiB / 15109MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Table 5. Comparison of the Pascal Tesla P4 and the Turing Tesla T4

GPU	Tesla P4 (Pascal)	Tesla T4 (Turing)
GPCs	4	5
TPCs	20	20
SMs	20	40
CUDA Cores/SM	128	64
CUDA Cores/GPU	2,560	2,560
Tensor Cores/SM	NA	8
Tensor Cores/GPU	NA	320
RT Cores	NA	40
GPU Base Clock MHz	810	585*
GPU Boost Clock MHz	1,063	1,590
Peak FP32 TFLOPS	5.5	8.1
Peak INT32 TIPS	NA	8.1
Peak FP16 TFLOPS	NA	16.2
Peak FP16 Tensor TFLOPS with FP16 Accumulate*	NA	65
Peak FP16 Tensor TFLOPS with FP32 Accumulate*	NA	65
Peak INT8 Tensor TOPS*	22	130
Peak INT4 Tensor TOPS*	NA	260
Frame Buffer Memory Size and Type	8192 MB GDDR5X	16384 MB GDDR6
Memory Interface	256-bit	256-bit
Memory Clock (Data Rate)	6 Gbps	10 Gbps
Memory Bandwidth (GB/sec)	192	320
ROPs	64	64
TDP	75 Watts	70 Watts

CUDA kernel launch


```
void CPUFunction()
{
  printf("This function is defined to run on the CPU.\n");
}

__global__ void GPUFunction()
{
  printf("This function is defined to run on the GPU.\n");
}

int main()
{
  CPUFunction();

  GPUFunction<<<1, 1>>>();
  cudaDeviceSynchronize();
}
```

.cu file

Function qualifiers


```
__global__ void GPUFunction()
{
  printf("This function is defined to run on the GPU.\n");
}
```

	Executed on	Callable from
`__device__ type DeviceFunc()`	Device	Device
`__global__ void KernelFunc()`	Device	Host
`__host__ type HostFunc()`	Host	Host

- (GPU = device, CPU = host)

Kernel configuration


```
int main()
{
  CPUFunction();

  GPUFunction<<<1, 1>>>();
  cudaDeviceSynchronize();
}
```

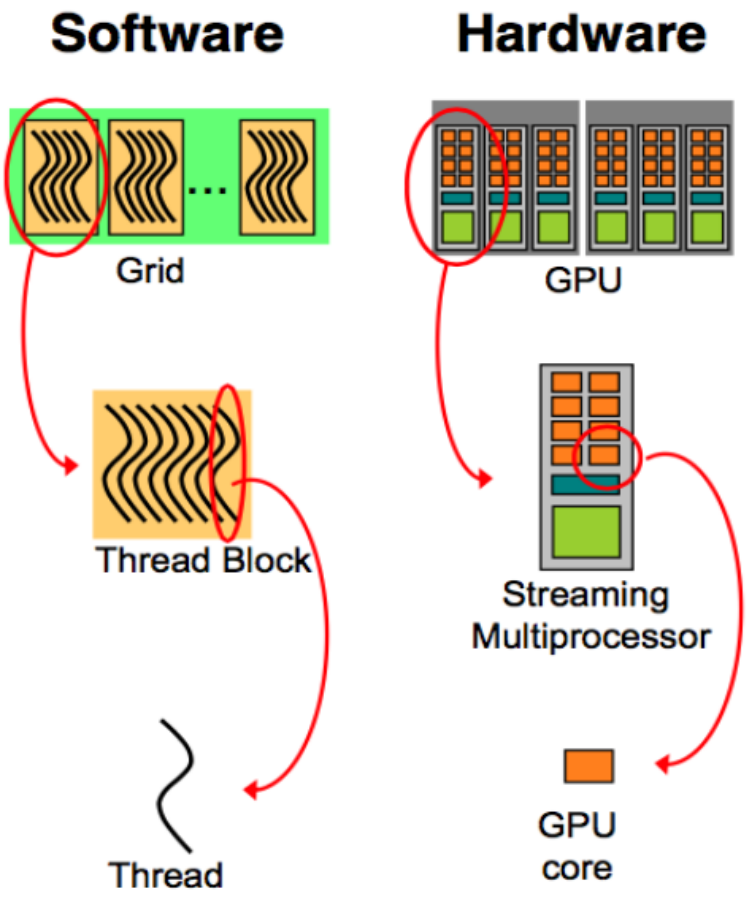
- Execution configuration:
  - Number of blocks
  - Number of threads in each block
- 1D hierarchy → integer numbers
- Higher dimensions → use `dim3` objects

CPU-GPU synchronization


```
int main()
{
  CPUFunction();

  GPUFunction<<<1, 1>>>();
  cudaDeviceSynchronize();
}
```

- Kernel launch is asynchronous
  - CPU can keep working while GPU is executing the kernel
- Synchronization is achieved with CUDA runtime functions

CUDA execution model



- A GPU kernel is invoked:
  - Each thread block is assigned to a SM
  - Threads within a block are divided in warps

Blocks should be aligned to SMs

Threads within a block should be a multiple of the number of threads in a warp

Execution Configuration Mismatches

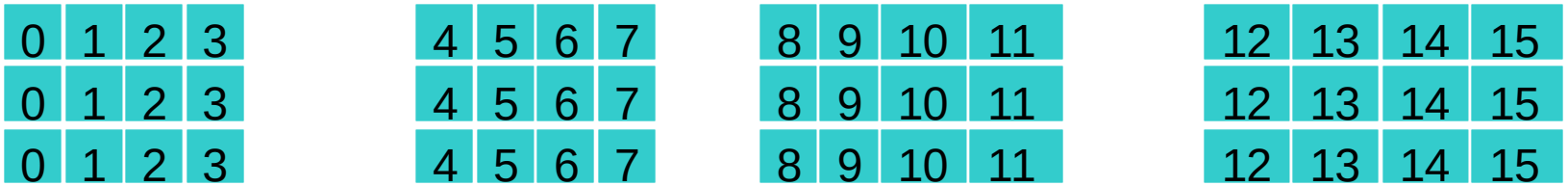
- Blocks that contain a multiple of 32 threads are desirable for performance reasons

GPU	Kepler GK180	Maxwell GM200	Pascal GP100	Volta GV100
Compute Capability	3.5	5.2	6.0	7.0
Threads / Warp	32	32	32	32
Max Warps / SM	64	64	64	64
Max Threads / SM	2048	2048	2048	2048
Max Thread Blocks / SM	16	32	32	32

- If the dataset is not a multiple of 32 (e.g. N=1000):
  - Create more than 1000 threads
  - Pass the size N of the dataset as an argument to the kernel
  - Inside the kernel: check if the thread ID is larger than N, only do work if it is smaller

Execution Configuration Mismatches

- Opposite case: there are *less* threads than values to process (maybe still for performance reasons, or because the dataset is so large that all the threads in the GPU are not enough)
- Use a stride factor corresponding to the number of threads
  - In case of a 1D grid of 1D blocks: `gridDim.x * blockDim.x`



gridDim.x = 4
blockDim.x = 4
stride = 16

```
for(int i = tId; i < N; i += stride)
        // work on a[i]
```

Querying Device Properties

- Different GPUs may have a different number of SMs

```
int deviceId;
cudaGetDevice(&deviceId);
cudaDeviceProp props;
cudaGetDeviceProperties(&props, deviceId);
```

```
struct cudaDeviceProp {
    char name[256];
    size_t totalGlobalMem;
    size_t sharedMemPerBlock;
    int regsPerBlock;
    int warpSize;
    size_t memPitch;
    int maxThreadsPerBlock;
    int maxThreadsDim[3];
    int maxGridSize[3];
    size_t totalConstMem;
    int major;
    int minor;
    int clockRate;
    size_t textureAlignment;
    int deviceOverlap;
    int multiProcessorCount;
    int kernelExecTimeoutEnabled;
    int integrated;
    int canMapHostMemory;
    int computeMode;
    int concurrentKernels;
    int ECCEnabled;
    int pciBusID;
    int pciDeviceID;
    int tccDriver;
}
```

Querying Device Properties

- What are the maximum dimensions of a thread block? `devicePROP.maxThreadsDim[0] x devicePROP.maxThreadsDim[1] x devicePROP.maxThreadsDim[2]`
- What is the maximum number of threads in a block? `devicePROP.maxThreadsPerBlock`
- What is the maximum size of the GPU global memory? `devicePROP.totalGlobalMem`
- What is the warp size? `devicePROP.warpSize`
- How many Streaming Multiprocessors are available? `devicePROP.multiProcessorCount`

Querying Device Properties

- Is an execution configuration of <<<100, 1000>>> suitable on this device?
  - Is 100 a multiple of the number of SMs?
  - Is 1000 a multiple of the number of threads in a warp?
- Suppose you have a CUDA program using multiple blocks, should each block contain 4x4, 8x8, or 30x30 threads?
  - How many threads/SM are allowed?
  - How many blocks/SM are allowed?
  - Check compute capability: `deviceProp.major`

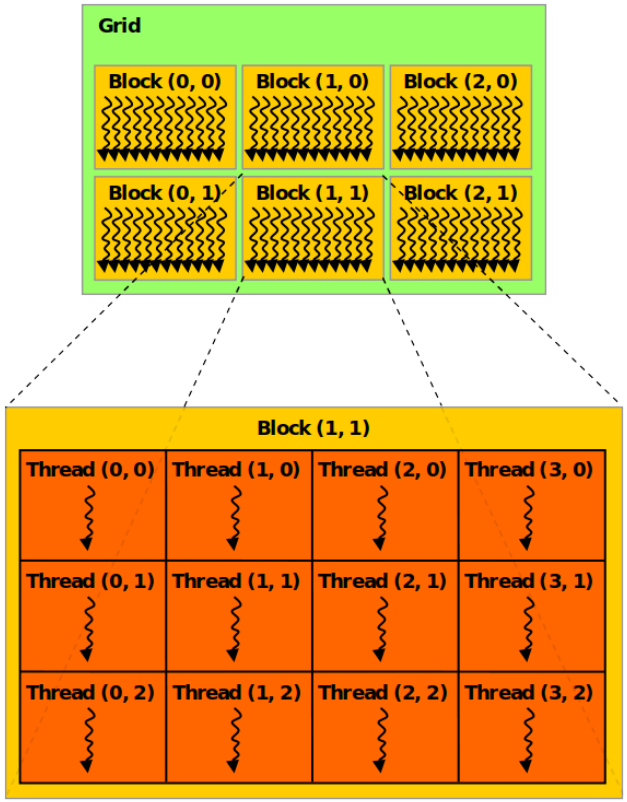
Error Handling

- CUDA functions return a variable of `cudaError_t` type
- Kernels return void, so use the `cudaGetLastError` function
- Asynchronous errors (during kernel execution) are reported as error states of `cudaDeviceSynchronize`

```
cudaError_t err1, err2;
err1 = cudaMallocManaged(&a, N)

SomeKernel<<<1, -1>>>(); // invalid thread number

err2 = cudaGetLastError();
if (err1 != cudaSuccess || err2 != cudaSuccess)
{
  printf("Error 1: %s\n", cudaGetErrorString(err1));
  printf("Error 2: %s\n", cudaGetErrorString(err2));
}
```

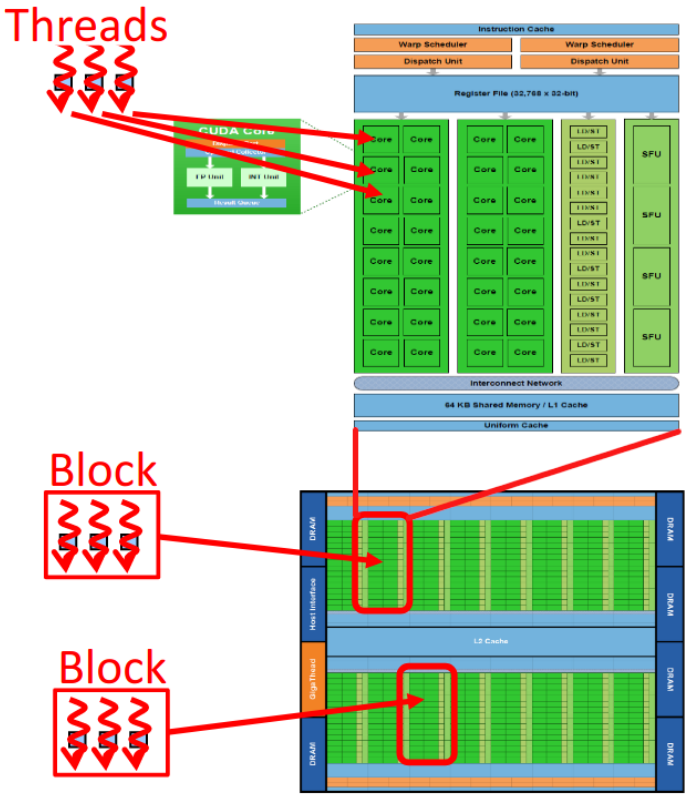
CUDA thread hierarchy

- CUDA thread hierarchy
  - *Threads* are grouped together in *blocks*
  - *Blocks* are grouped together in *grids*
  - Blocks and grids are organized as N-dimensional (up to 3) arrays
- Threads that belong to the same block can cooperate through a shared memory
- IDs are used to identify threads and blocks



CUDA thread hierarchy

- CUDA thread hierarchy matches the underlying NVIDIA architecture
  - Each *core* executes a thread
  - Each *SM* executes one or more blocks
  - A *GPU* executes one or more grids
- Each SM executes threads of the same block in groups of 32 (*warps*)
- If there are more blocks than SMs, execution is interleaved



Coordinating Parallel Threads


```
void increment(int* a, int b, int N){
  for (int i=0; i<N; i++){
    a[i] = a[i] + b;
  }
}

void main(){
   ...
   increment(a, b, N);
   ...
}
```

CPU program

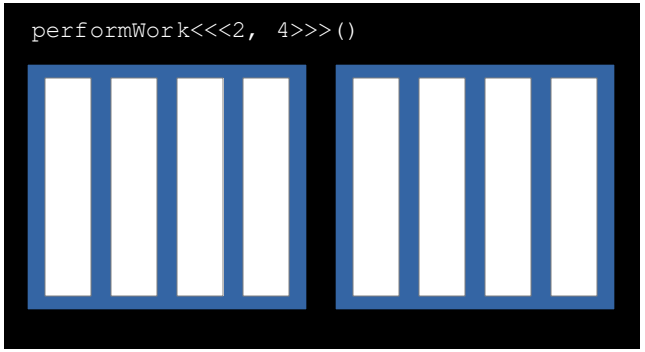
- Max threads per block is typically 1024
  - If we want more threads, we can exploit the CUDA thread hierarchy

in case N is not a multiple of dimBlock

```
__global__
void increment(int* a, int b, int N){
  int i = blockIdx.x * blockDim.x
    + threadIdx.x;
  if (i<N)
    a[i] = a[i] + b;
}

void main(){
   ...
   dim3 dimBlock(blocksize);
   dim3 dimGrid(ceil(N/(float)blocksize));
   increment<<<dimGrid, dimBlock>>>(a,b,N);
   ...
}
```

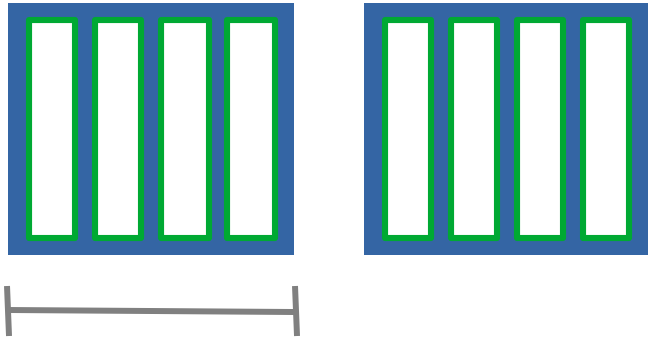
CUDA program

Thread Hierarchy Variables



gridDim.x = 2

blockIdx.x = 0 blockIdx.x = 1



blockDim.x = 4

threadIdx.x = 0 1 2 3

```
globalID = blockIDx.x
* blockDim.x +
threadIdx.x
```

Thread Hierarchy Variables

- N-element vector:

- Let's assume N=16, blocksize=4

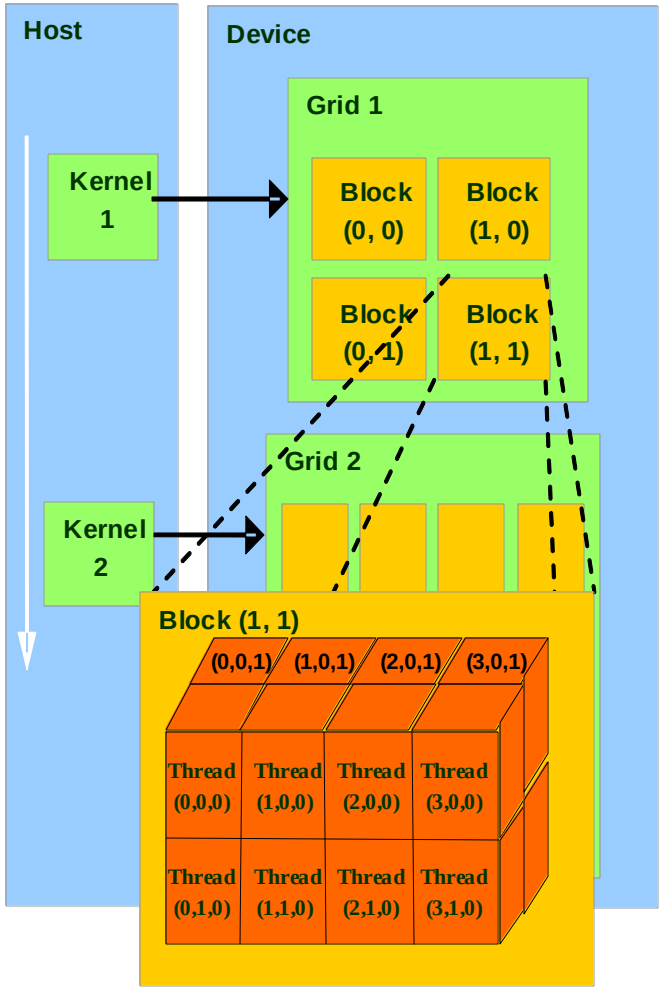
```
dimBlock(blocksize) = 4
dimGrid(ceil(N/(float)blocksize)) = 4
```

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

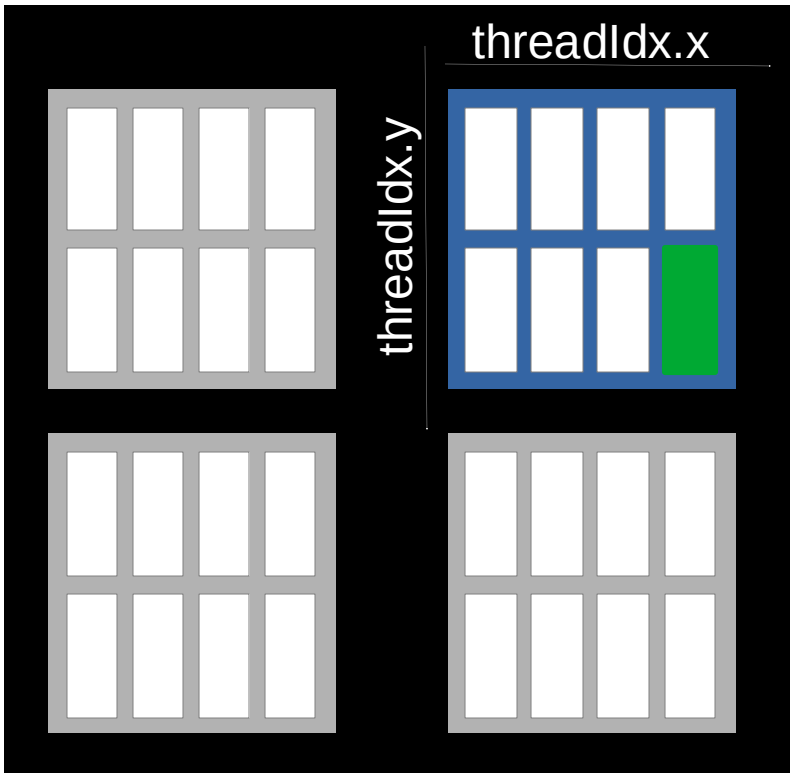
blockIdx.x=0 threadIdx.x=0, 1,2,3	blockIdx.x=1 threadIdx.x=0, 1,2,3	blockIdx.x=2 threadIdx.x=0, 1,2,3	blockIdx.x=3 threadIdx.x=0, 1,2,3
0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15

Thread Hierarchy Variables

- Each thread uses IDs to decide what data to work on
- 2D/3D structures simplify memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes



Thread Hierarchy Variables




```
gridDim.x = 2
gridDim.y = 2
blockDim.x = 4
blockDim.y = 2
```

threadsPerBlock = blockDim.x * blockDim.y = 8

threadNumInBlock = threadIdx.x + blockDim.x * theadIdx.y = 3 + 4*1 = 7

blockNumInGrid = blockIdx.x + gridDim.x * blockIdx.y = 1 + 2*0 = 1

tId = blockNumInGrid * threadsPerBlock + threadNumInBlock = 1*8 + 7 = 15

Thread Hierarchy Variables

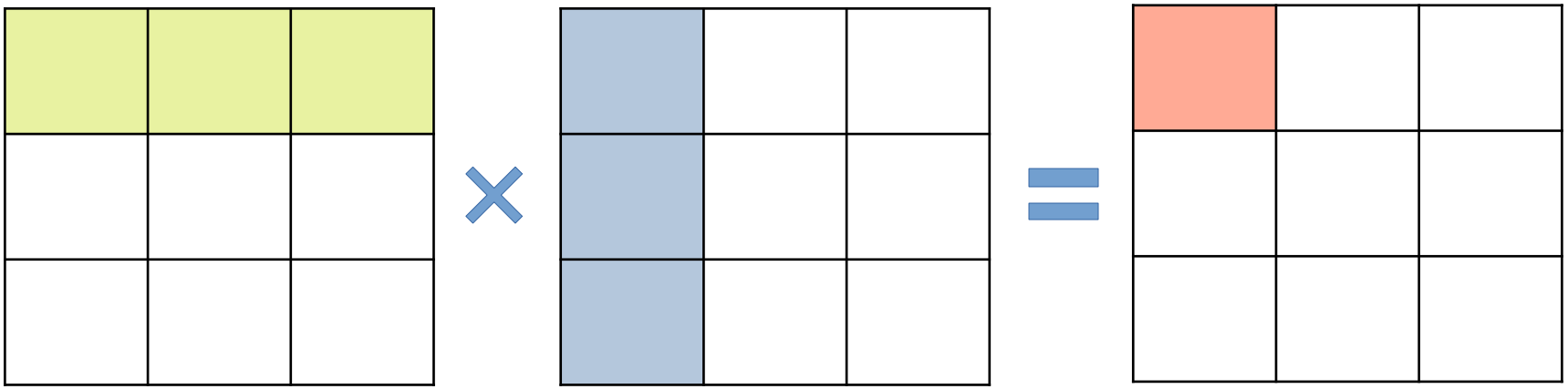
- 6D organization (3D Grid + 3D Blocks)

```
dim3 threads_per_block(16, 16, 1);
dim3 number_of_blocks(16, 16, 1);
someKernel<<<number_of_blocks, threads_per_block>>>();
```

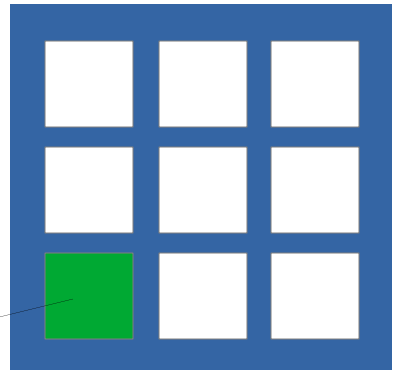
```
gridDim.x = 16
gridDim.y = 16
gridDim.z = 1

0 ≤ blockIdx.x ≤ 15
0 ≤ blockIdx.y ≤ 15
blockIdx.z = 0
```

```
blockDim.x = 16
blockDim.y = 16
blockDim.z = 1

0 ≤ threadIdx.x ≤ 15
0 ≤ threadIdx.y ≤ 15
threadIdx.z = 0
```

Matrix Multiplication



- Each thread needs a row index and a column index
- If the threads are organized in a 2D block, index calculation is easier

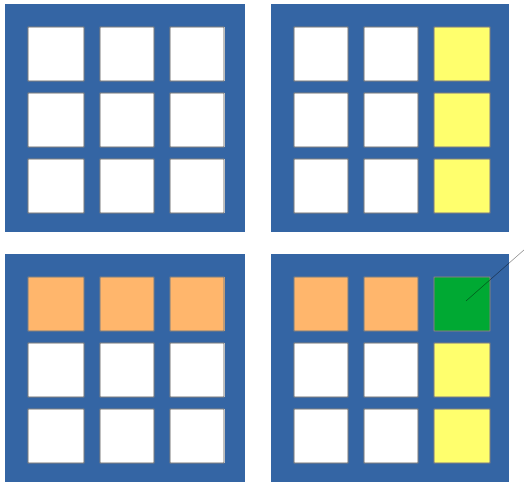


Compute pixel (0,2) from row 2 and column 0

Compute pixel (**threadIdx.x**, **threadIdx.y**) from row **threadIdx.y** and column **threadIdx.x**

Matrix Multiplication

- What if we have 64x64 inputs? We need 4096 threads, while the limit of threads within a block is 1024
- To maintain the *global* 2D structure, we need a 2D grid of 2D blocks



Compute pixel (3,5) from row 3 and column 5

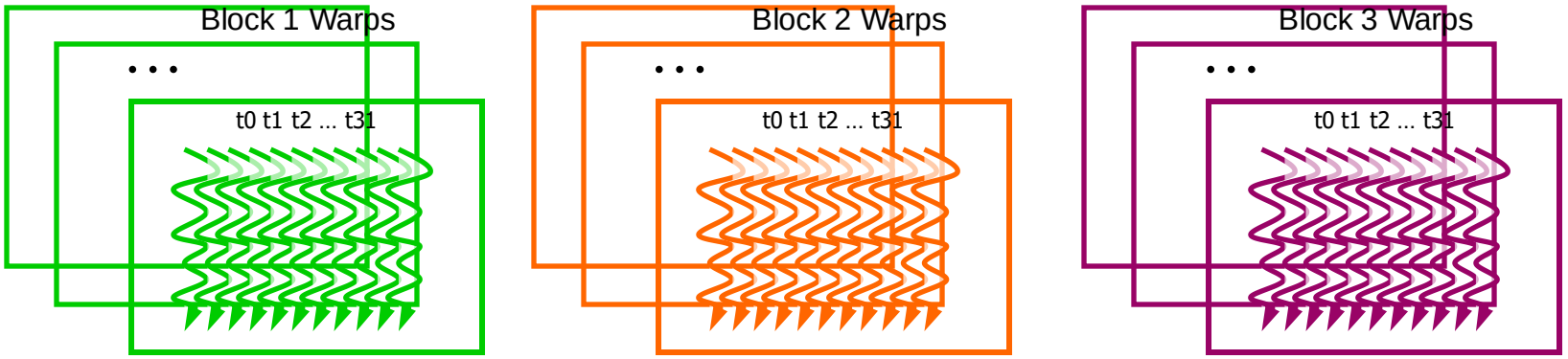
Compute pixel (**globalRow**, **globalColumn**) from row **globalRow** and column **globalColumn**

globalRow = threadIdx.y + blockDim.y * blockIdx.y

globalColumn = threadIdx.x + blockDim.x * blockIdx.x

The matrix is stored as an array, row after row. So the output pixel position will be out(globalColumn + N * globalRow)

Warp scheduling



- Blocks are divided in warps
- Warps are scheduling units on the SM
- All threads in a warp execute the same instruction (SIMD)
  - Control unit for instruction fetch, decode, and control is shared among multiple processing units
  - Control overhead is minimized

Control divergence

- Control divergence occurs when threads in a warp take different control flow paths by making different control decisions
  - If path/else path
  - Different number of loop iterations
- The execution of threads taking different paths are serialized in current GPUs
  - The control paths taken by the threads in a warp are traversed one at a time
  - During the execution of each path, only the threads taking that path will be executed in parallel

Control divergence

- Example with control divergence:

```
if (threadIdx.x > 2) { }
```

  - Decision granularity less than warp size
  - Threads 0, 1 and 2 follow different path than the rest of the threads in the first warp

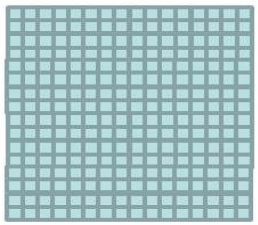
- Example without control divergence:

```
if (blockIdx.x > 2) { }
```

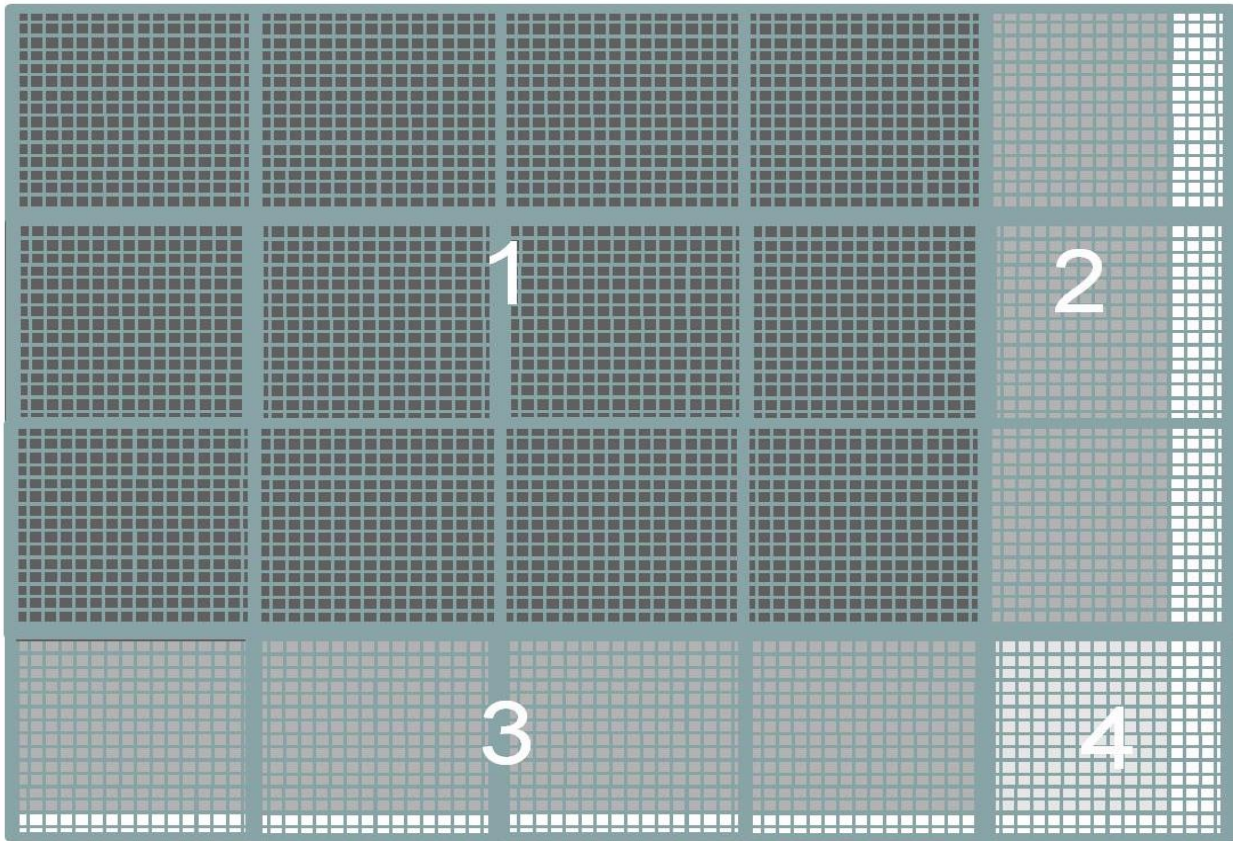
  - Decision granularity is a multiple of block size
  - All threads in any given warp follow the same path

Control divergence

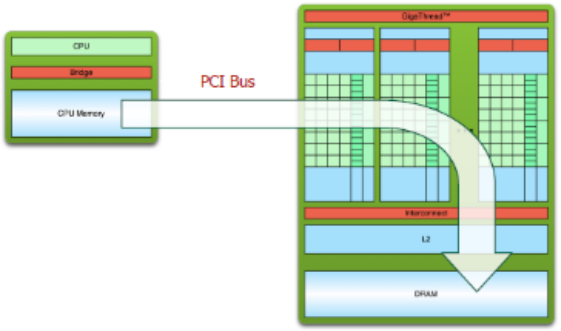
- Covering a 62x76 picture with 16x16 blocks:

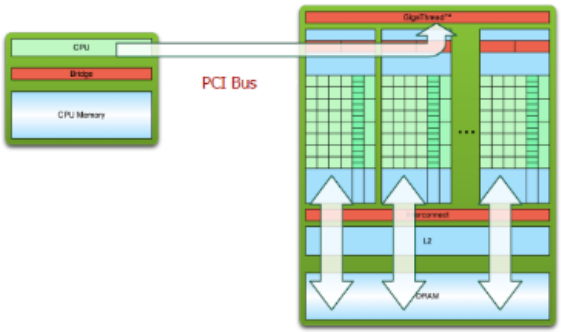


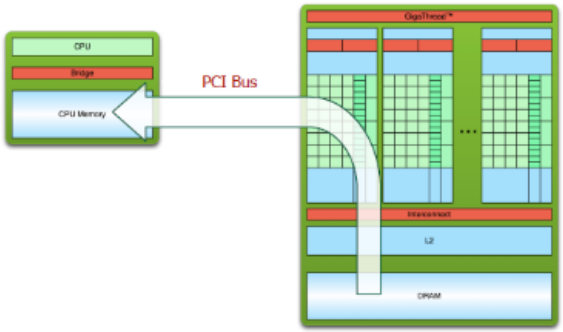
16x16 block



Memory allocation and data transfer







1) Copy input data from CPU memory to GPU memory

- allocate device memory for input and output arguments
- copy input arguments to device memory

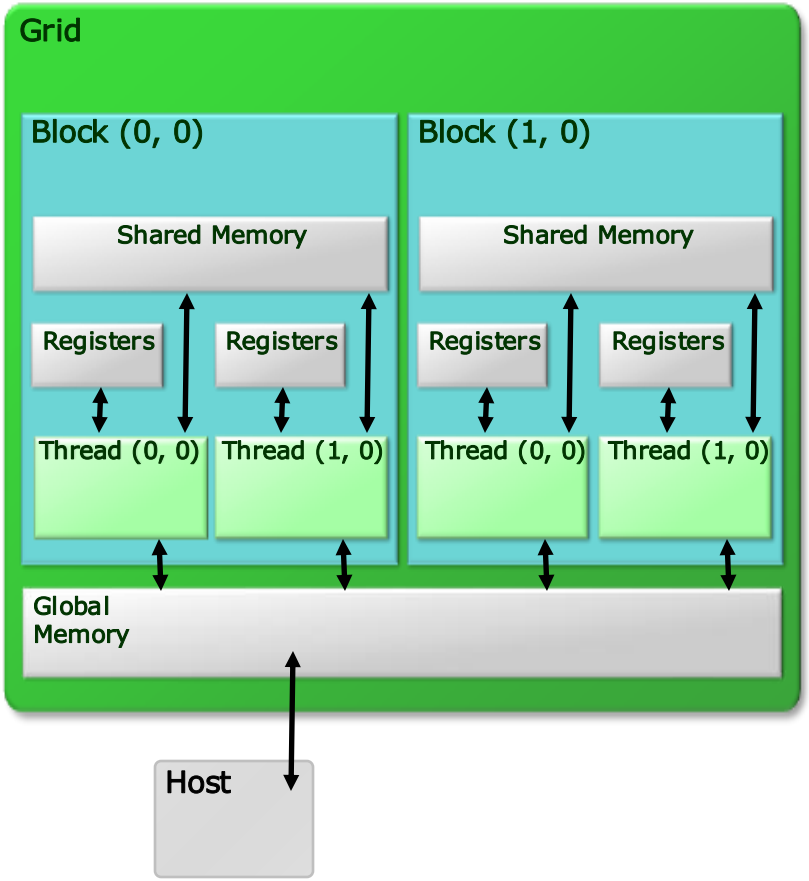
2) Load GPU program and execute, caching data on chip for performance

3) Copy results from GPU memory to CPU memory

- copy output arguments from the device memory
- free device vectors

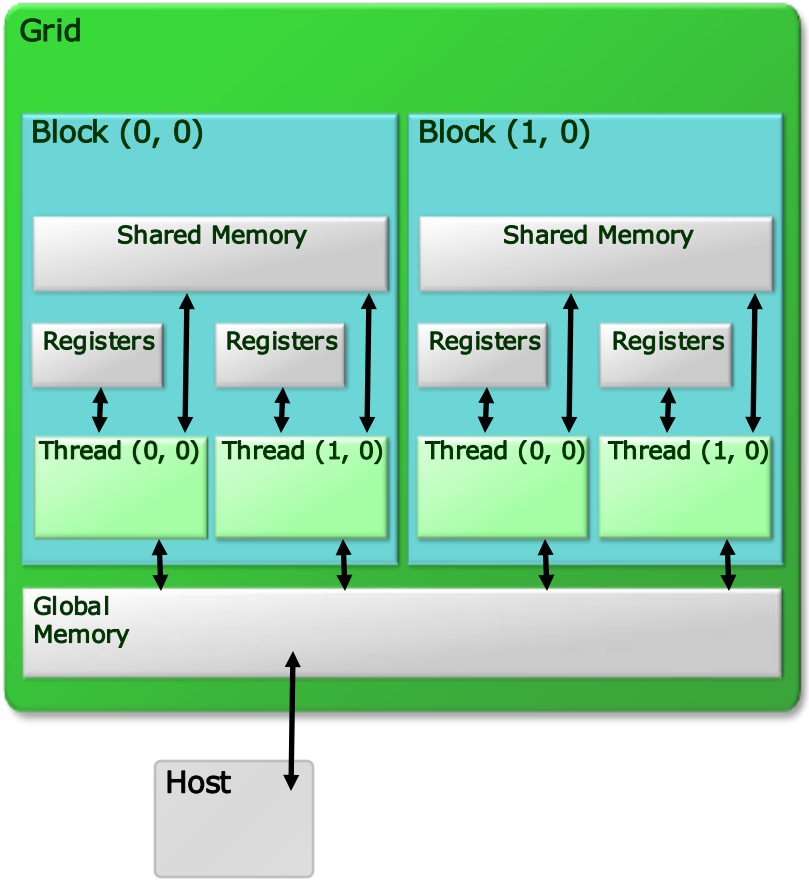
CUDA memory hierarchy

- Three types of memory are available, matching the hardware architecture
  - Per-thread Private Local Memory (Registers)
  - Per-block Shared Memory (Cache)
  - Global Memory (off-chip DRAM)



CUDA memory hierarchy

- Device code can
  - R/W per-thread registers
  - R/W per-block shared memory
  - R/W per-grid global memory
- Host code can
  - Transfer data to/from per-grid global memory



CUDA memory management APIs

- Manual method, for when it is known in advance that data will only be used on the device or on the host

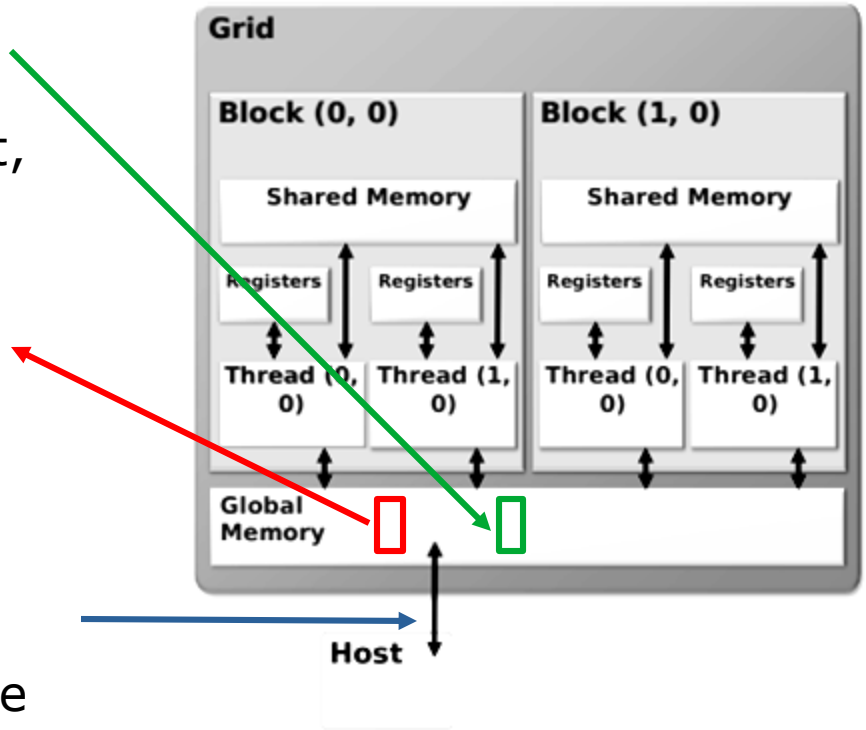
cudaMalloc — Allocate memory to the GPU, not accessible from host code. **No GPU page faults**

cudaMallocHost
cudaFreeHost — Allocate and pin memory on the CPU, allows asynchronous copying to and from the GPU. **Can reduce CPU performance**

cudaMemcpy — Copy (**not transfer!**) data from the CPU to the GPU or viceversa

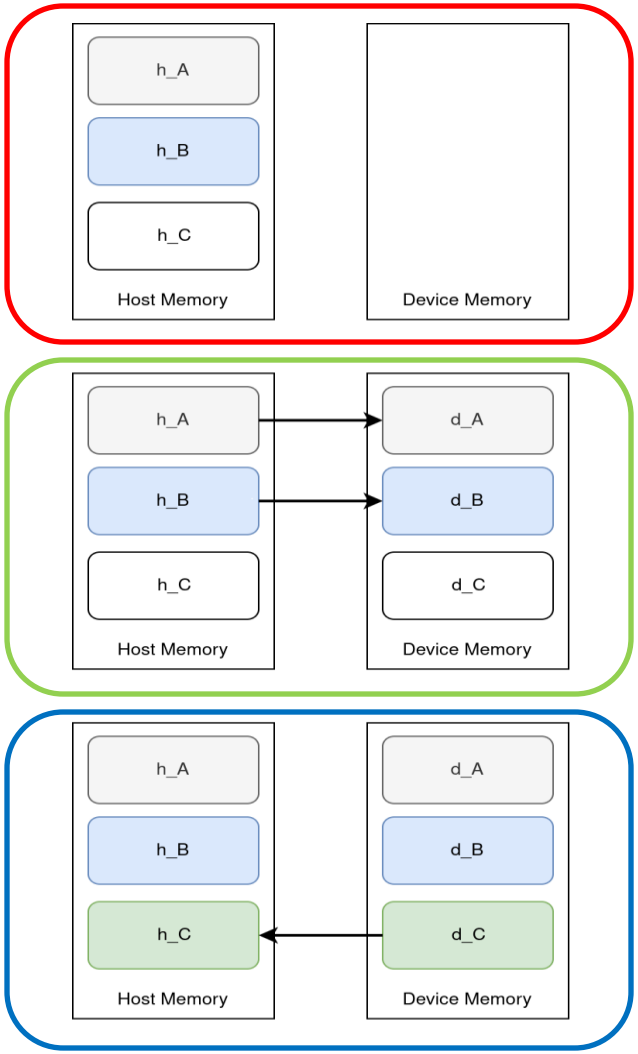
CUDA memory management APIs

- cudaMalloc()
  - Allocates an object in the device global memory
  - Parameters: address of a pointer to the allocated object, size in bytes of the object
- cudaFree()
  - Frees object from the device global memory
  - Parameter: pointer to the object to free
- cudaMemcpy()
  - Memory data transfer
  - Parameters: pointers to source and destination, bytes copied, direction
  - Transfer to device is synchronous



Example: manual memory management

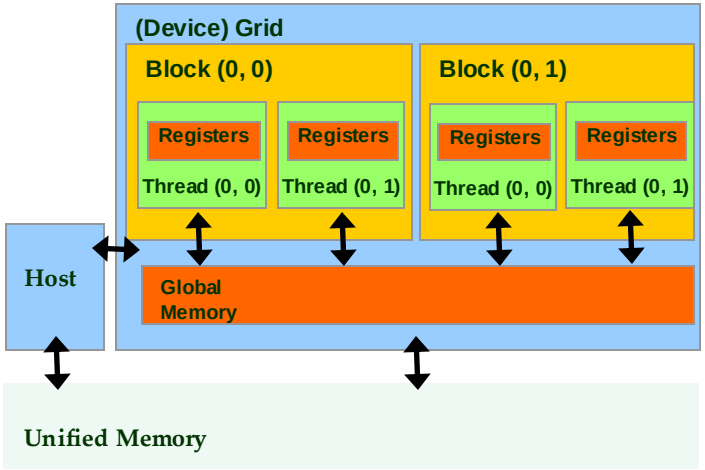

```
… Allocate h_A, h_B, h_C …

void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n * sizeof(float); float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMalloc((void **) &d_B, size);
    cudaMalloc((void **) &d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

     // Kernel invocation code

     cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
     cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}

… Free h_A, h_B, h_C …
```



Unified Memory

- Automated method through CUDA unified memory

```
int N = 2<<20;
size_t size = N * sizeof(int);

int *a;
cudaMallocManaged(&a, size);

// Use 'a' on the CPU and/or on the GPU

cudaFree(a);
```



- `a` can be referenced in both host and device code, CUDA will manage memory transfers transparently
- Single memory space, single copy of data (even across multiple CPUs/GPUs)

Unified Memory

- `cudaMallocManaged` – completely transparent to the user, easiest method to transfer memory between host and device
- Disadvantages:
  - Even if data usage is known in advance, data management will be handled reactively at runtime – **on-demand migration**
  - Data transfer is synchronous with kernel execution
  - Fragmented transfers don't exploit well memory bandwidth
- More knowledge of the Unified Memory and reports from nsys can help improve performance

Unified Memory

- On-demand migration is triggered by **page faults**, and there are applications where this is the most efficient way of handling data transfers:
  - Sparse, unpredictable memory access
  - Multi-GPU systems
- If this is not the case, page faults and on-demand migration may cause a significant overhead

```
CUDA Memory Operation Statistics (by size in KiB):

   Total     Operations  Average  Minimum  Maximum             Operation
----------  ----------  -------  -------  -------  ---------------------------------
131072.000        4671   28.061    4.000  944.000  [CUDA Unified Memory memcpy HtoD]
```

Many operations

Varying transfer size

Asynchronous Memory Prefetching

- Use asynchronous memory prefetching to prevent page faults and move data in larger batches

```
int deviceId;
cudaGetDevice(&deviceId);
cudaMemPrefetchAsync(pointerToSomeUMData, size, deviceId);
cudaMemPrefetchAsync(pointerToSomeUMData, size, cudaCpuDeviceId);
```

Host to device – use GPU ID

Device to host – use CUDA built-in variable

Asynchronous Memory Prefetching

- Without asynchronous memory prefetching:

```
CUDA Memory Operation Statistics (by size in KiB):

   Total     Operations  Average  Minimum  Maximum             Operation
----------  ----------  -------  -------  -------  --------------------------------
131072.000        4671   28.061    4.000  944.000  [CUDA Unified Memory memcpy HtoD]
```

Many operations

Varying transfer size

- And data transfer is part of kernel execution time!
- With asynchronous memory prefetching:

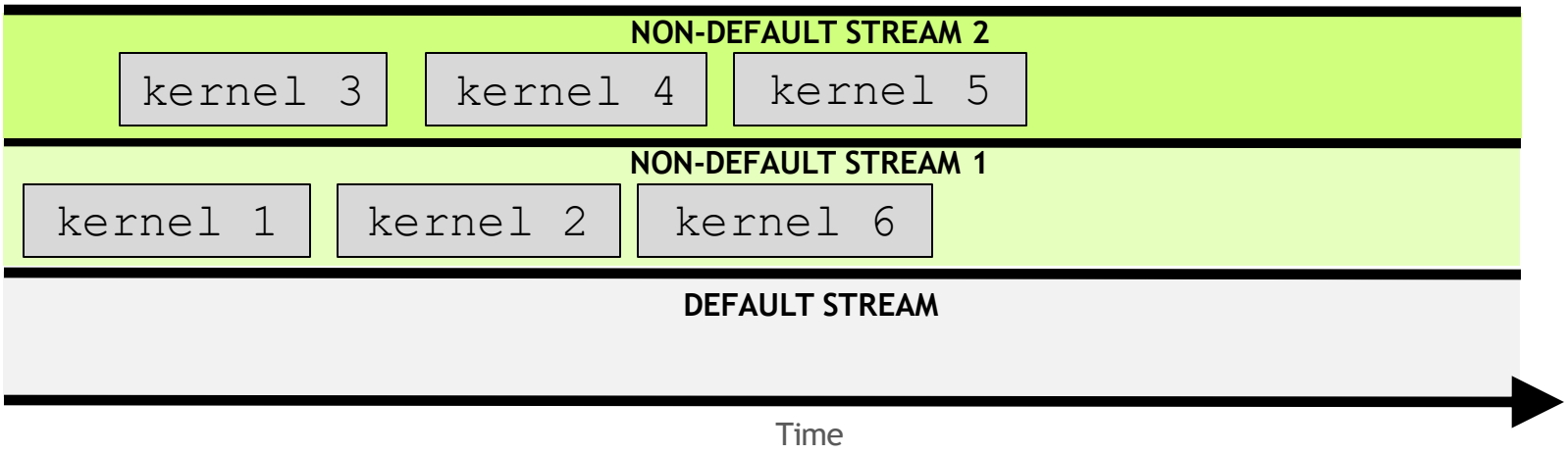
```
CUDA Memory Operation Statistics (by size in KiB):

   Total     Operations  Average   Minimum   Maximum             Operation
----------  ----------  --------  --------  --------  --------------------------------
393216.000         192  2048.000  2048.000  2048.000  [CUDA Unified Memory memcpy HtoD]
```

Fewer operations

Regular transfer size

Concurrent CUDA Streams

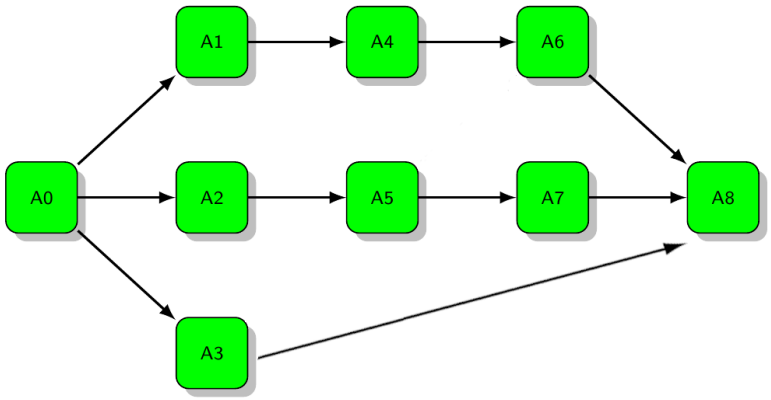
- A stream is a series of commands that execute in order
- Up to now, all CUDA code has been executed within the *default* stream



- Using different streams unlocks a new layer of parallelism

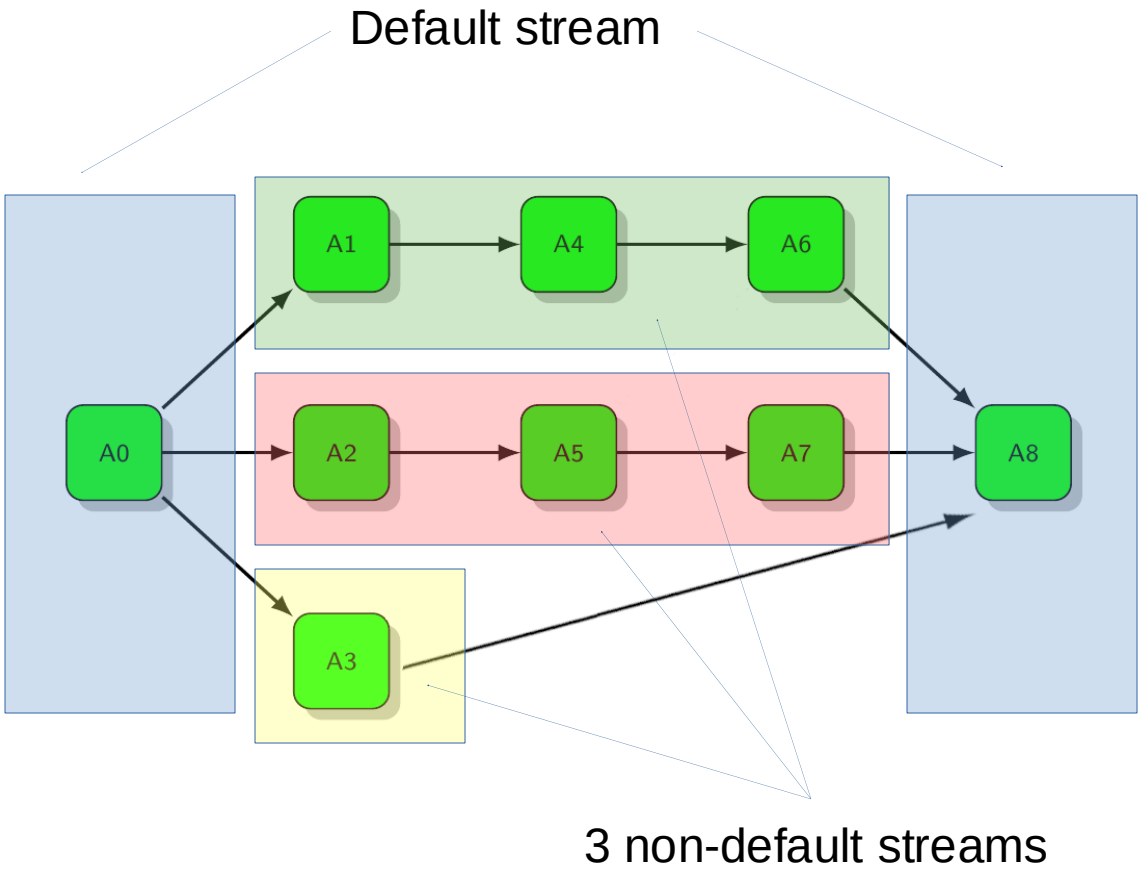
Concurrent CUDA Streams

- Operations within a given stream occur in order
- Operations in different non-default streams are not guaranteed to operate in any specific order relative to each other
- The default stream is blocking
  - It will wait for all other streams to complete before running
  - It will block other streams from running until it completes



Concurrent CUDA Streams

- Example: DAG-like applications with concurrent tasks



Concurrent CUDA Streams

- Creating, utilizing, and destroying non-default CUDA streams:

```
cudaStream_t stream;
cudaStreamCreate(&stream);
kernel<<<num_blocks, threads_per_block, 0, stream>>>();
cudaStreamDestroy(stream);
```

Extra shared memory per block, this is an advanced topic that will not be covered here

Streams and Device Memory Management

- Manual memory management prevents on-demand migration
- Non-default streams and manual memory management allow overlapping data transfer and computation

cudaMemcpyAsync — Asynchronously copy memory. Can be launched in a non-default stream

```
for (int i = 0; i < numberOfSegments; ++i)
{
  ...
  cudaMemcpyAsync(&device_array[segmentOffset], &host_array[segmentOffset], segmentSize,
     cudaMemcpyHostToDevice, stream[i]);
  kernel<<<number_of_blocks, threads_per_block, 0, stream[i]>>>(&device_array[segmentOffset], segmentN);
  ...
}
```

Create stream with data transfer and computation for each segment of work

Exercises

https://drive.google.com/file/d/1VtnUObU3pZg5T5QJlX1XeTm7c0GwJUqD/view?usp=share_link **Exercises 1-7**

Credits & References

- NVIDIA Deep Learning Institute, "Accelerated Computing" Teaching Kit
- More self-learning material from NVIDIA can be found at: https://developer.nvidia.com/cuda-education
- D.B. Kirk and W.W. Hwu, "Programming Massively Parallel Processors. A hands-on approach", 3rd edition, Morgan Kaufmann 2017