GPU Programming

(in Cuda)

Julian Gutierrez

NUCAR

Session 2

Introduction to CUDA

C for CUDA

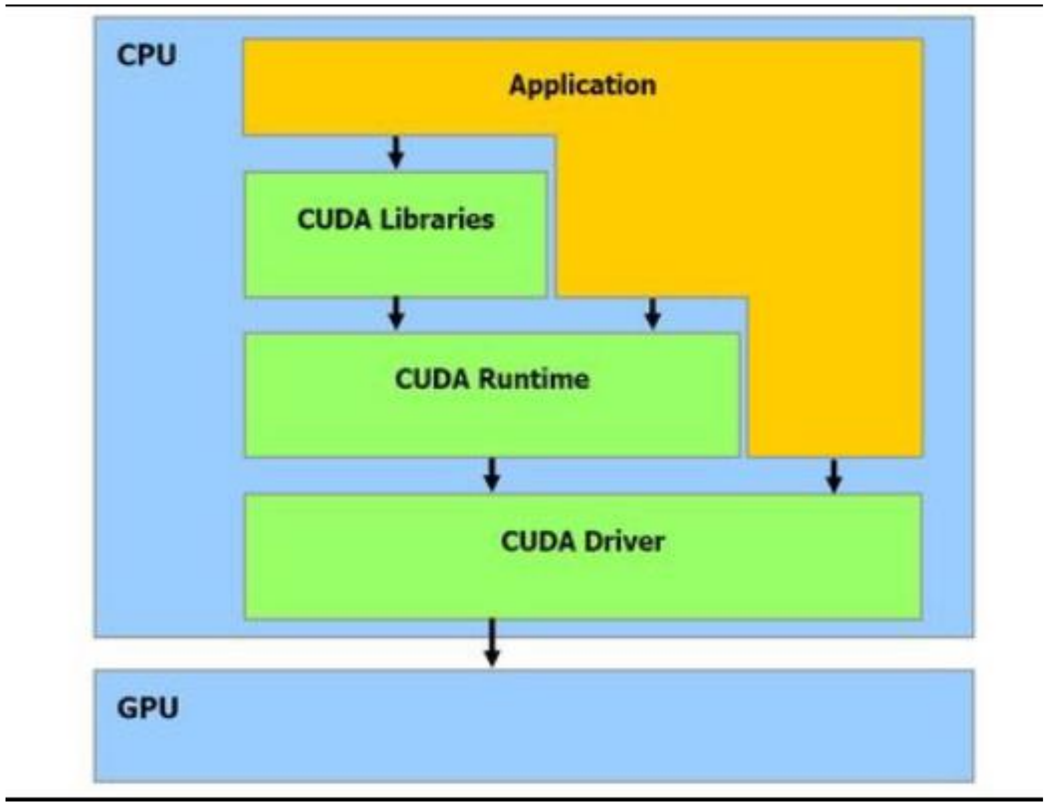
- The language that started the GPGPU excitement
- CUDA only runs on NVIDIA GPUs
- Highest performance programming framework for NVIDIA GPUs
- Learning curve similar to threaded C programming
  - Large performance gains require mapping program to specific underlying architecture

CUDA: A platform for heterogeneous Computing

- CUDA general purpose parallel platform
- CUDA is a standard ANSI C-like language
- CUDA API:
  - Manages devices
  - Memory
  - Synchronization
  - Etc.

CUDA: A platform for heterogeneous Computing

- CUDA API offers two levels:
  - CUDA Driver API. Low level API, offers better control over the GPU.
  - CUDA Runtime API. Higher level; implemented on top of the driver API.
    - We will use this API.



CUDA: A platform for heterogeneous Computing

C Code

```c
for (int i=0;i < MAXi;i++)
   for(int j=0;j< MAXj;j++){

       ...code that uses i and j....
   }
```

CUDA Code

```c
dim3 blocks(MAXj, 1);
dim3 grids(MAXi, 1);

kernel<<<grids, blocks, 1>>>()

__global__ kernel()
{
   int i = blockIdx.x;
   int j = threadIdx.x;

   ...code that uses i and j....

}
```

Threads

CUDA: A platform for heterogeneous Computing

- Concepts
  - Host
    - CPU.
    - Executes the main function, any other CPU related jobs.
  - Device
    - GPU
    - Executes the kernel functions.

CUDA: A platform for heterogeneous Computing

- CUDA Kernels: data-parallel function
  - A kernel is a function callable from the host and executed on the CUDA device
  - It runs “simultaneously” by many threads in parallel.
- CUDA compiler: nvcc
  - Separates the device code from the host code during compilation process.

CUDA Programming Model

- We will learn:
  - Hierarchy structure of threads.
  - Hierarchy structure of memory.
- Point of view:
  - Domain point of view. How to solve the problem using parallel programming (structure).
  - Logic point of view. How to use the threads and the calculation to obtain the correct result.
  - Hardware point of view. How to use the hardware to deliver a high performance implementation.

CUDA Programming Model

- **Kernel Code**
  - key component
  - Runs on the GPU
  - The sequential code executed by each thread

- Flow of a CUDA program:
  1. Copy data from CPU to GPU (cudaMemcpyHostToDevice)
  2. Invoke Kernels to operate over the GPU data (asynchronous call)
  3. Copy data back from GPU to CPU (cudaMemcpyDeviceToHost)

CUDA Programming Model

- Copy data from CPU to GPU
  - Allocate space

```
cudaError_t cudaMalloc ( void** devPtr, size_t size )
```

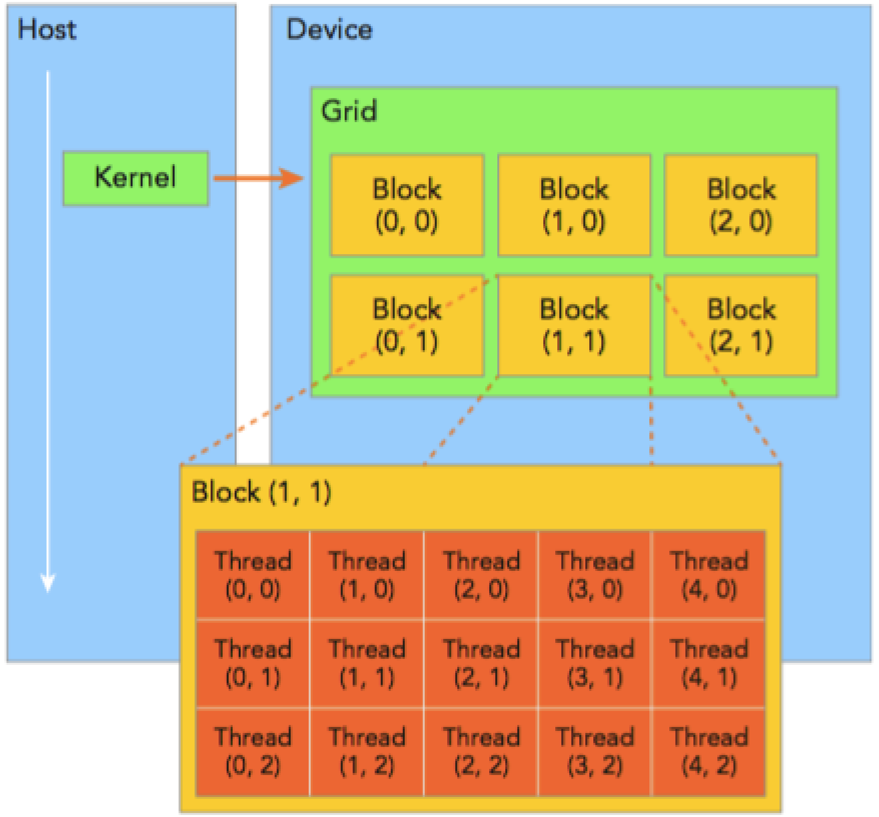
  - Transfer data

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

cudaMemcpyHostToDevice
cudaMemcpyDeviceToHost

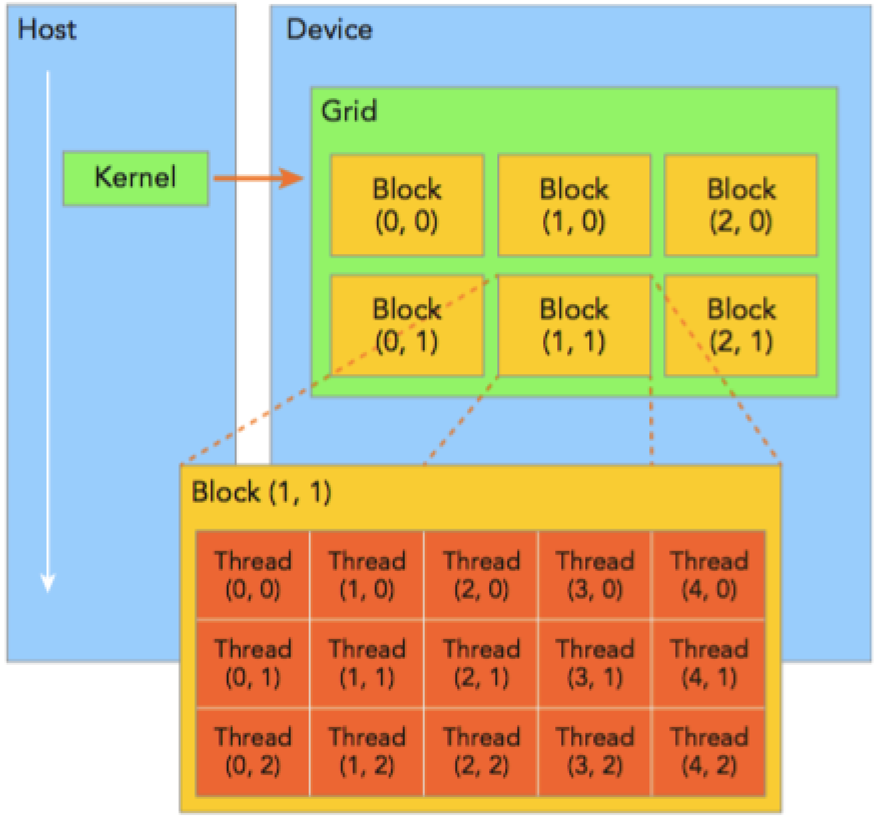
CUDA Programming Model

- Invoke Kernels to operate over the GPU data (asynchronous call)
- Think about adding 2 vectors together:
  - Ai + Bi = Ci
  - How do you divide the number of threads?
  - What changes if you have a 2D array instead?
  - What changes if the solution requires synchronization between certain threads?



CUDA Programming Model

- Invoke Kernels to operate over the GPU data (asynchronous call)
- Organizing threads
  - Threads Per Block (threadBlock)
    - threadIdx.x, threadIdx.y, threadIdx.z
    - blockDim.x, blockDim.y, blockDim.z (measured in threads)
  - Blocks in a Grid (grid)
    - blockIdx.x, blockIdx.y, blockIdx.z
    - gridDim.x, gridDim.y, gridDim.z (measured in blocks)



CUDA Programming Model

- How to define the threadBlocks and gridSize?
  - Consider the nature of the problem
  - Consider the nature of the GPU architecture
  - Use dim3

```
dim3 block(3);
dim3 grid((nElem+block.x-1)/block.x);
```

GridSize is rounded up of multiple of blocks, this based on the parallel architecture of the GPU

CUDA Kernels

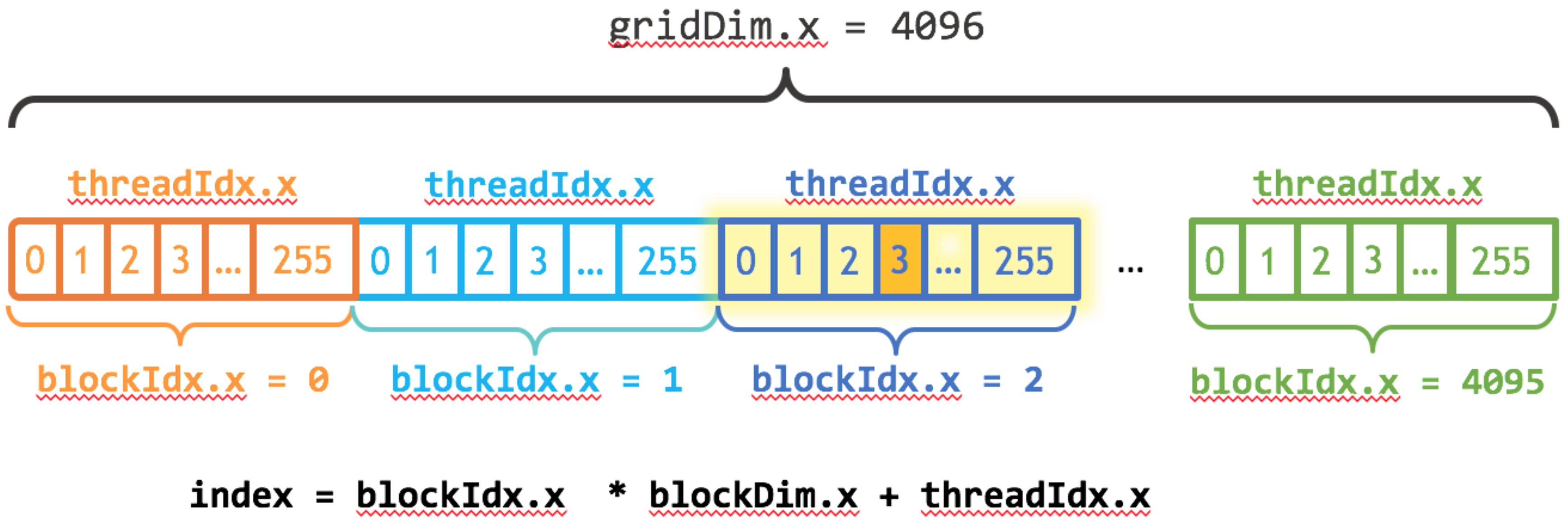
- Invocation of the CUDA kernel from host will have triple-angle-brackets:

```
kernel_name <<<grid, block>>>(argument list);
```

- grid*block is the total number of threads launched for the kernel.

CUDA Kernels

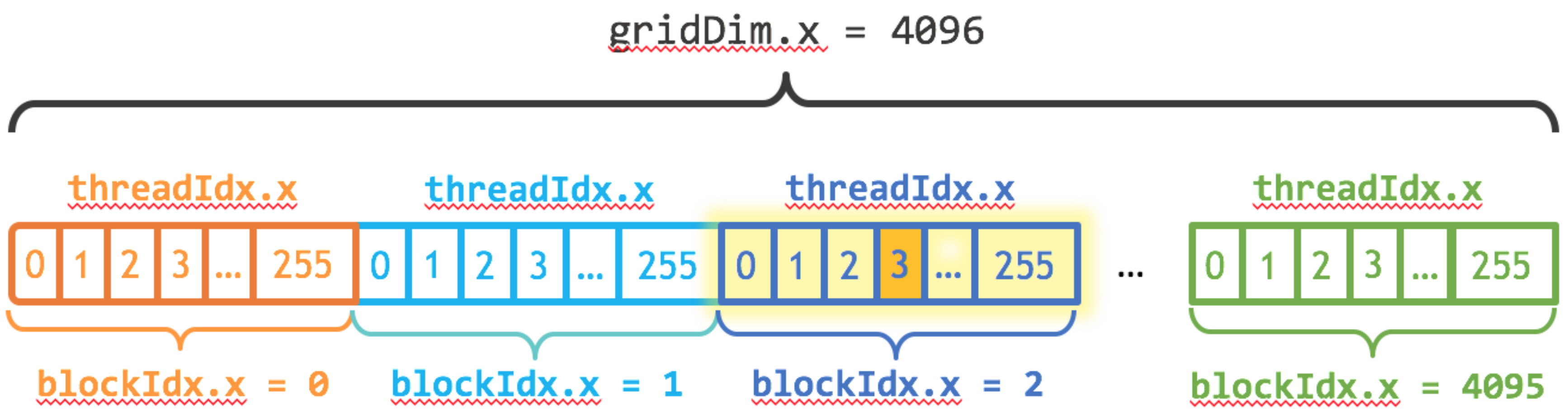
```
kernel_name<<<4096, 256>>>(argument list);
```



CUDA Kernels

How many threads in total?

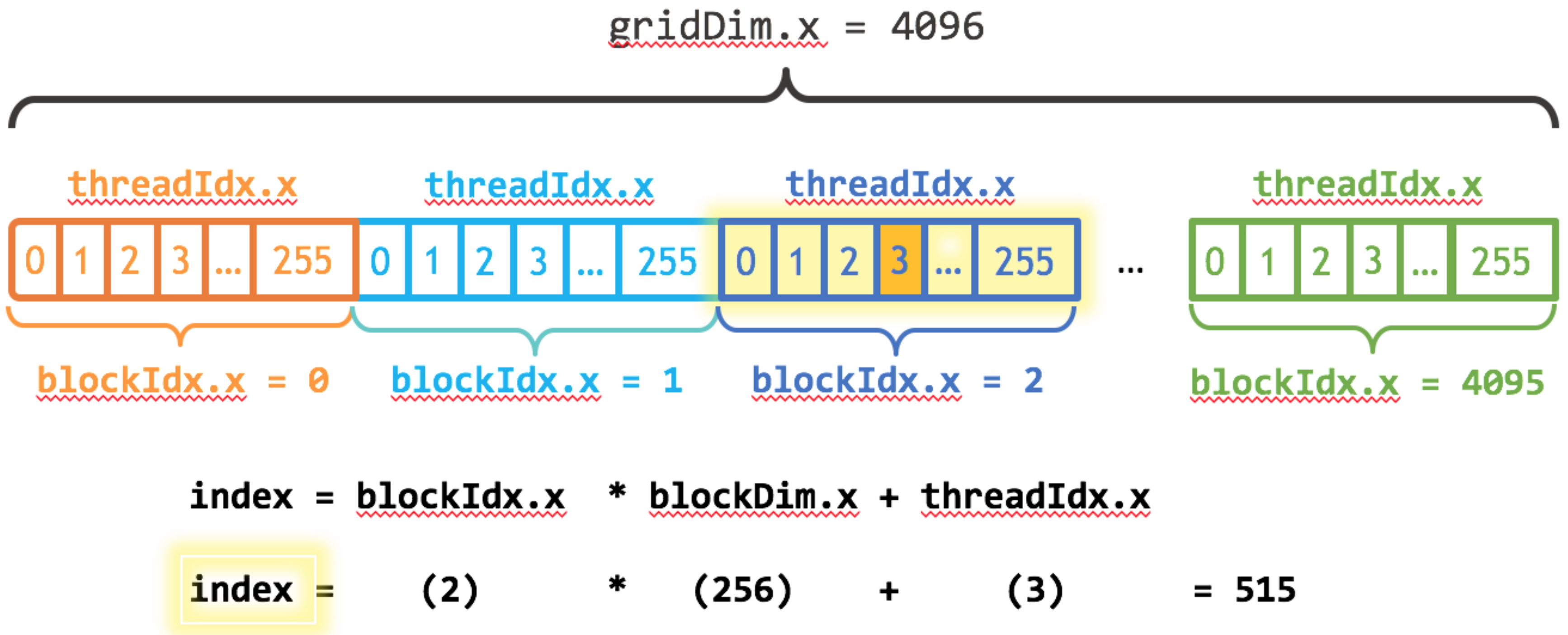
```
kernel_name<<<4096, 256>>>(argument list);
```



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

CUDA Kernels

```
kernel_name<<<4096, 256>>>(argument list);
```



CUDA Kernels

- Writing kernels:
  - Global qualifier (Executed on the device, Callable from the host only)

    ```
    __global__ void kernel_name(argument list);
    ```

  - Device qualifier (Executed on the device Callable from the device only)

    ```
    __device__ called from the device.
    ```

  - Some restrictions:
    - Use only device memory (pointers to GPU RAM)
    - return void
    - no support for variable number of parameters
    - It has asynchronous behavior.

CUDA Example

- Matrix Multiply

```
int main(int argC, char** argV)
{
        float *a, *b, *c, *test;
        //Setting matrix parameters.
        int row = ROW;
        int col = COL;
        int k = COL;
        //Setting host memory space.
        a = (float *) malloc(row*k*sizeof(float));
        b = (float *) malloc(k*col*sizeof(float));
        c = (float *) malloc(row*col*sizeof(float));
        test = (float *) malloc(row*col*sizeof(float));

       //Initializing [A] and [B] with random values from 1 to 10.
        for(int i=0; i<row; i++){
                for(int j=0; j<k; j++){
                        a[i*k+j] = rand()%10;
                }
        }
        for(int i=0; i<k; i++){
                for(int j=0; j<col; j++){
                        b[i*col+j] = rand()%10;
                }
        }
        //Performing sequential job.
        wallS0 = getWallTime();
        for(int i=0; i<row; i++){
                for(int j=0; j<col; j++){
                        sum = 0;
                        for(int w=0; w<k; w++){
                                sum += a[i*k+w] * b[w*col+j];
                        }
                        test[i*col+j]=sum;
                }
        }
        wallS1 = getWallTime();
        printf("Sequential Job Time: %f ms\n", (wallS1-wallS0)*1000);
}
```

CUDA Example

- GridSize roundup based on the elements on C

```
void matrixMultiplication(float *a, float *b, float *c, int row, int col, int k)
{
      int sizeA = row*k*sizeof(float);
      int sizeB = k*col*sizeof(float);
      int sizeC = row*col*sizeof(float);
      float *devA, *devB, *devC;

      cudaMalloc((void**)&devA, sizeA);
      cudaMalloc((void**)&devB, sizeB);
      cudaMalloc((void**)&devC, sizeC);

      cudaMemcpy(devA, a, sizeA, cudaMemcpyHostToDevice);
      cudaMemcpy(devB, b, sizeB, cudaMemcpyHostToDevice);

      dim3 dimBlock(16, 16, 1);
      dim3 dimGrid((COL+dimBlock.x-1)/dimBlock.x, (ROW+dimBlock.y-1)/dimBlock.y, 1);

      matrixMulKernel<<<dimGrid, dimBlock>>>(devA, devB, devC, row, col, k);

      cudaMemcpy(c, devC, sizeC, cudaMemcpyDeviceToHost);

      //Freeing device matrices.
      cudaFree(devA); cudaFree(devB); cudaFree(devC);
}
```

Allocation on the GPU

Transfer Data CPU to GPU

Kernel Call

Transfer back to CPU

CUDA Example

```
__global__ void matrixMulKernel( float *devA, float *devB, float *devC, int row, int col, int k){

        int txID = blockIdx.x * blockDim.x + threadIdx.x;
        int tyID = blockIdx.y * blockDim.y + threadIdx.y;

        if ((txID < col) && (tyID < row))
        {
                float Pvalue = 0;
                for(int w=0; w<k; w++)
                {
                        Pvalue += devA[tyID*k+w] * devB[w*k+txID];
                }
                devC[tyID*k+txID] = Pvalue;
        }
}
```

CUDA Example

- **Compilation process**

```
all:
	nvcc matrixMul.cu -o matrixMul
```

- **Execution**

```
./matrixMul
	Sequential Job Time: 588.227987 ms
	Parallel Job Time: 108.647108 ms
```

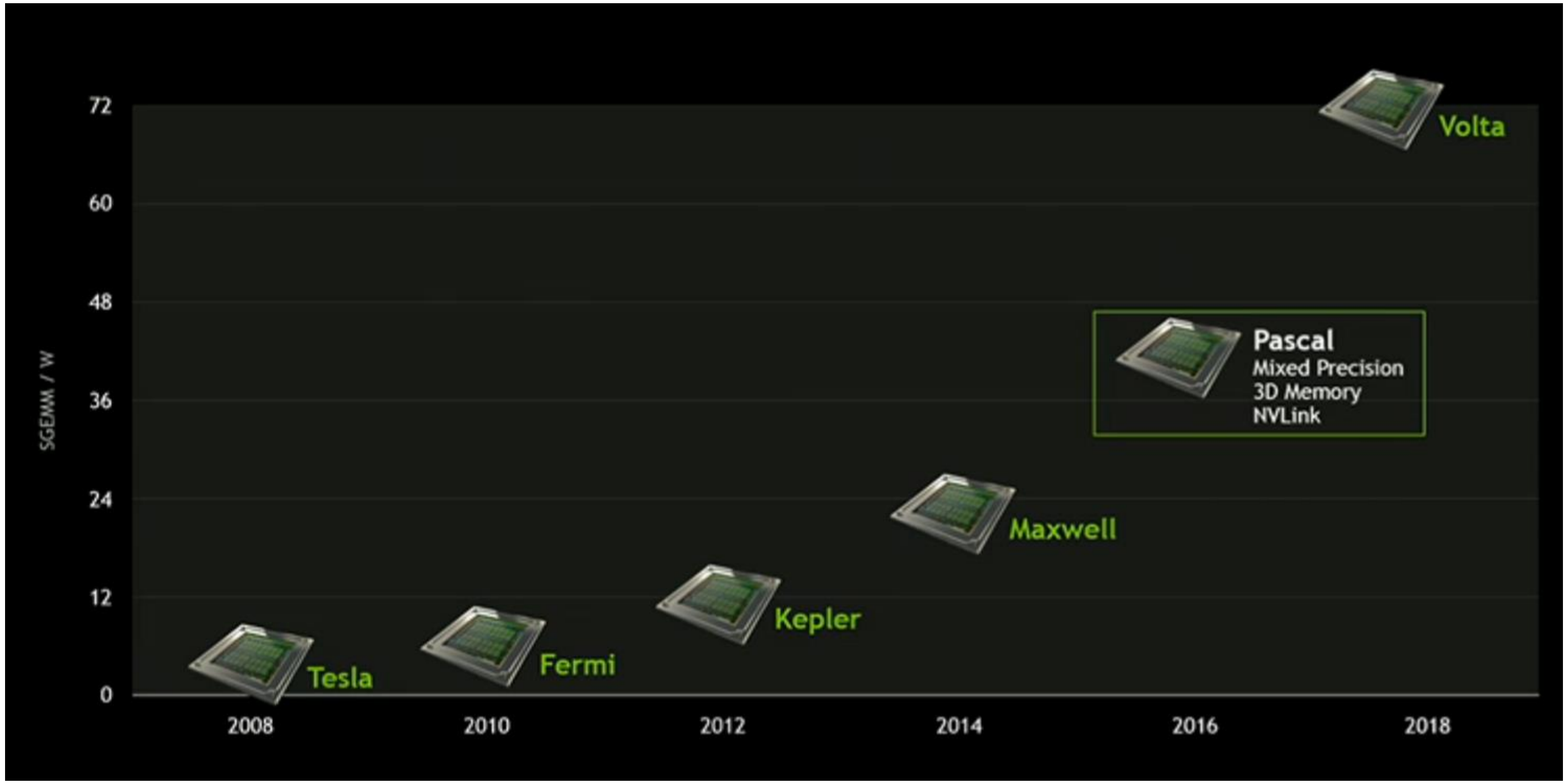
GPU Architecture Basics

Why Do we need to learn about the GPU architecture and the CUDA execution model?

Why Do we need to learn about the GPU architecture and the CUDA execution model?

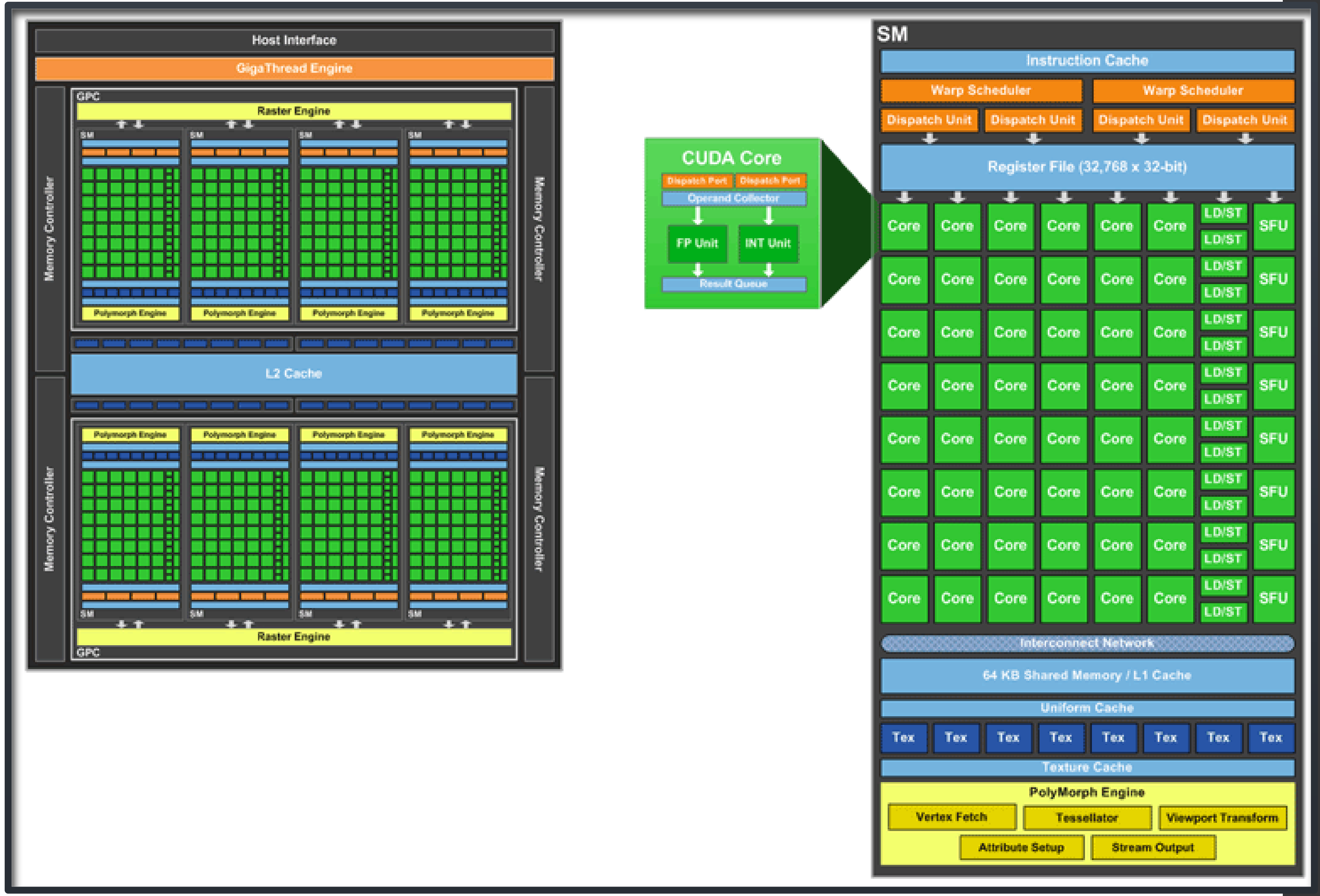
- Understand why the selected configuration outperforms others.
- Find a guideline to choose grid/block configuration

NVIDIA GPUs Roadmap



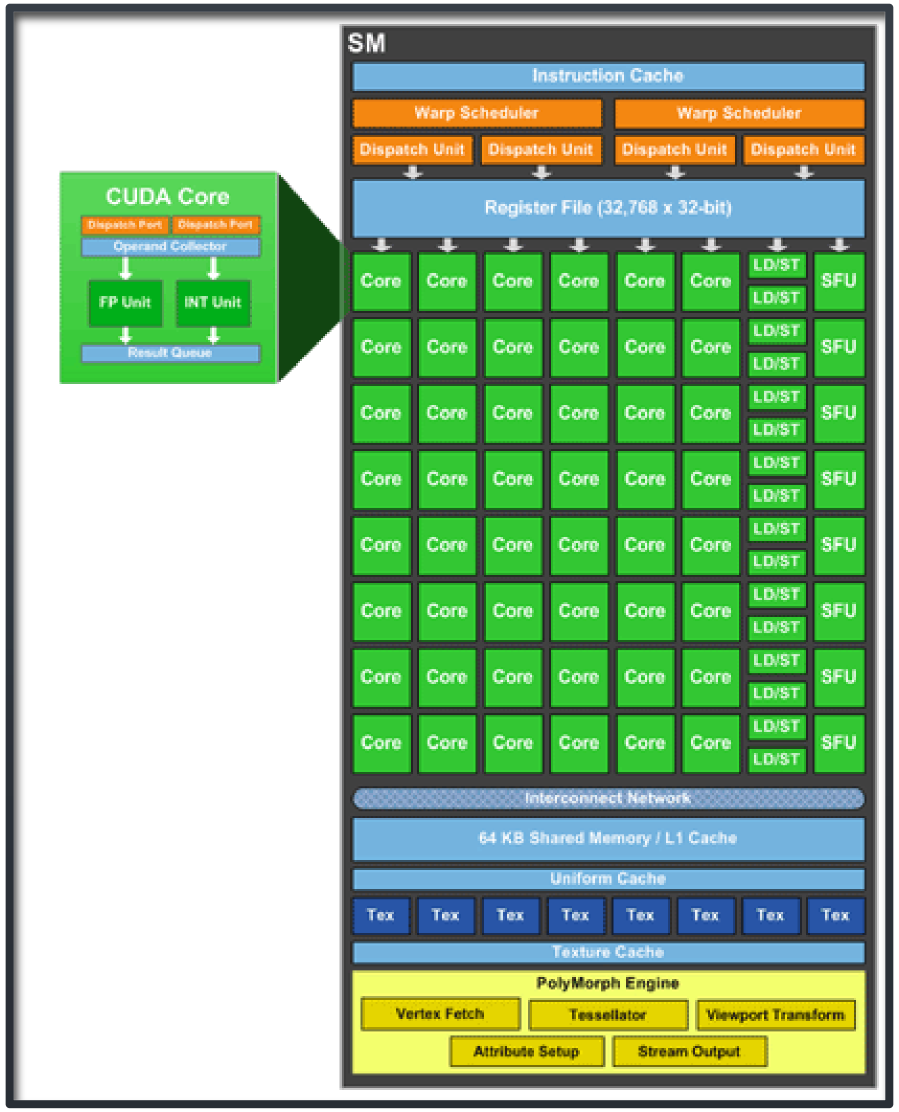
GPU Architecture

- Streaming Multiprocessor
- Kepler Architecture – Streaming Multiprocessor Extreme (SMX)



GPU Architecture

- Scalable array of Streaming Multiprocessors
  - CUDA Cores
  - Shared Memory
  - Register File
  - Load/Store Units
  - Special Function Units
  - Warp Scheduler

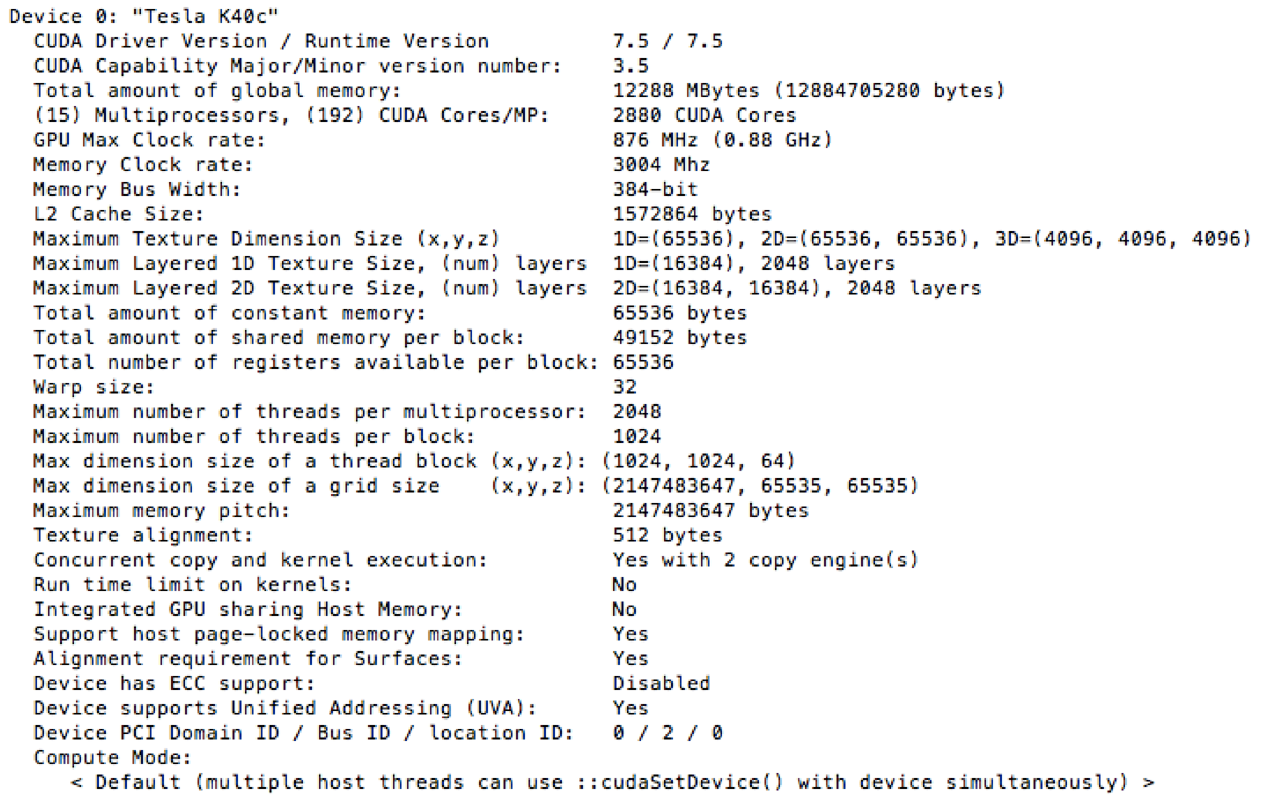


GPU Architecture

- Each SM is designed to support the execution of hundreds of threads.
- Multiple SMs per GPU
- Kepler K40
  - 15 Multiprocessors
  - 192 CUDA cores per SM
  - Number of Cores of processors: 2880

GPU Architecture

- CUDA deviceQuery in sample Utilities:
  - cp /shared/apps/cuda7.0/samples/1_Utilities/deviceQuery /home/<user>/ -r
  - cd deviceQuery
  - make
  - ./deviceQuery

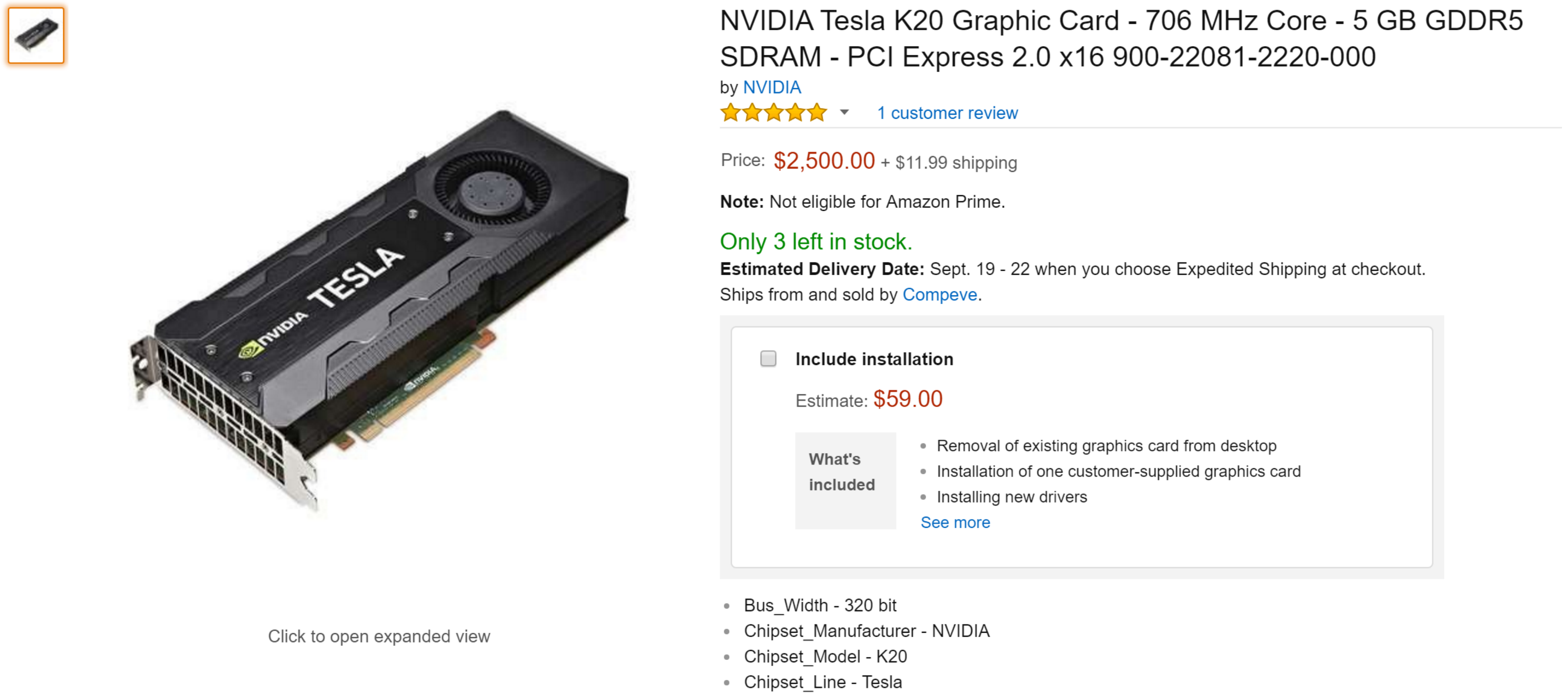


```
Device 0: "Tesla K40c"
  CUDA Driver Version / Runtime Version          7.5 / 7.5
  CUDA Capability Major/Minor version number:    3.5
  Total amount of global memory:                 12288 MBytes (12884705280 bytes)
  (15) Multiprocessors, (192) CUDA Cores/MP:     2880 CUDA Cores
  GPU Max Clock rate:                            876 MHz (0.88 GHz)
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 2 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

Kepler Architecture

- Kepler K20
  - 13 SMX
  - Enhanced SM (SMX)
  - Dynamic Parallelism
  - Hyper-Q
  - Faster atomic operations than previous generation

Tesla K40/K20



Click to open expanded view

NVIDIA Tesla K20 Graphic Card - 706 MHz Core - 5 GB GDDR5 SDRAM - PCI Express 2.0 x16 900-22081-2220-000

by NVIDIA

★★★★★ 1 customer review

Price: $2,500.00 + $11.99 shipping

Note: Not eligible for Amazon Prime.

Only 3 left in stock.

Estimated Delivery Date: Sept. 19 - 22 when you choose Expedited Shipping at checkout.
Ships from and sold by Compeve.

☐ **Include installation**

Estimate: $59.00

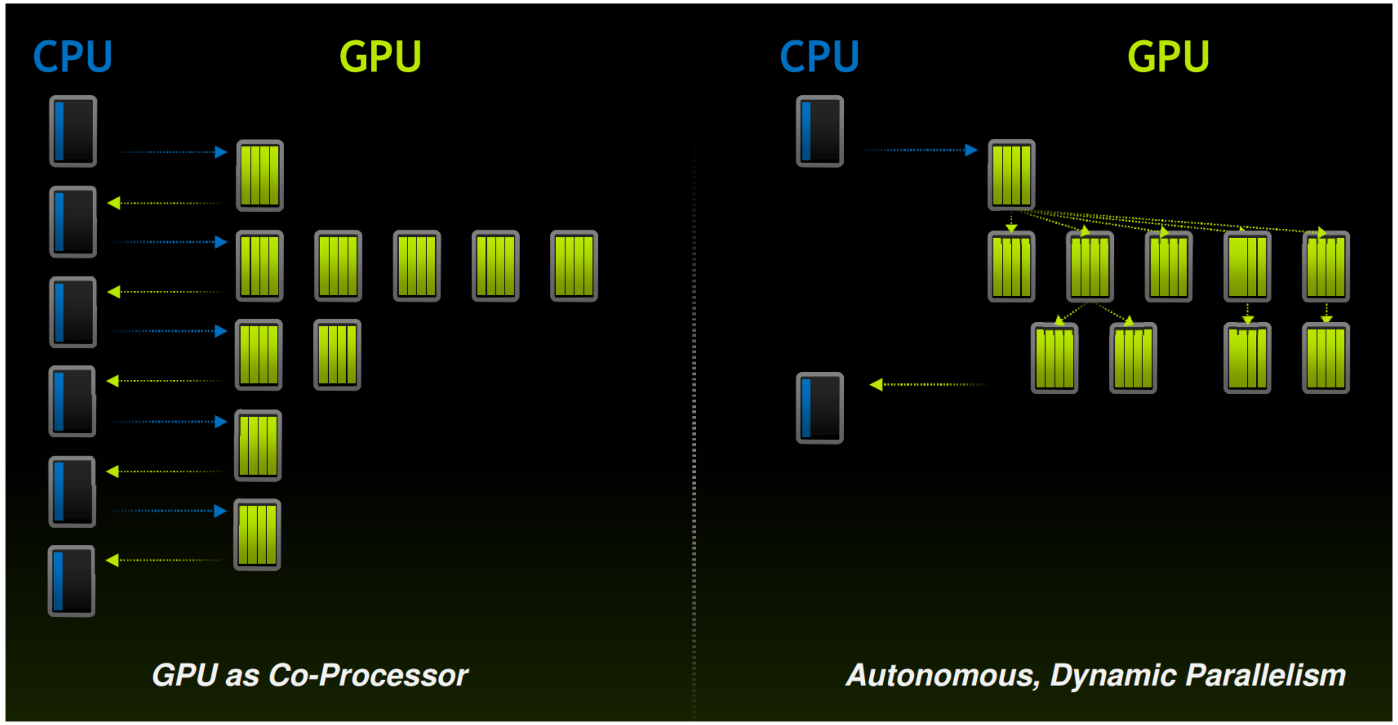
What's included
- Removal of existing graphics card from desktop
- Installation of one customer-supplied graphics card
- Installing new drivers

See more

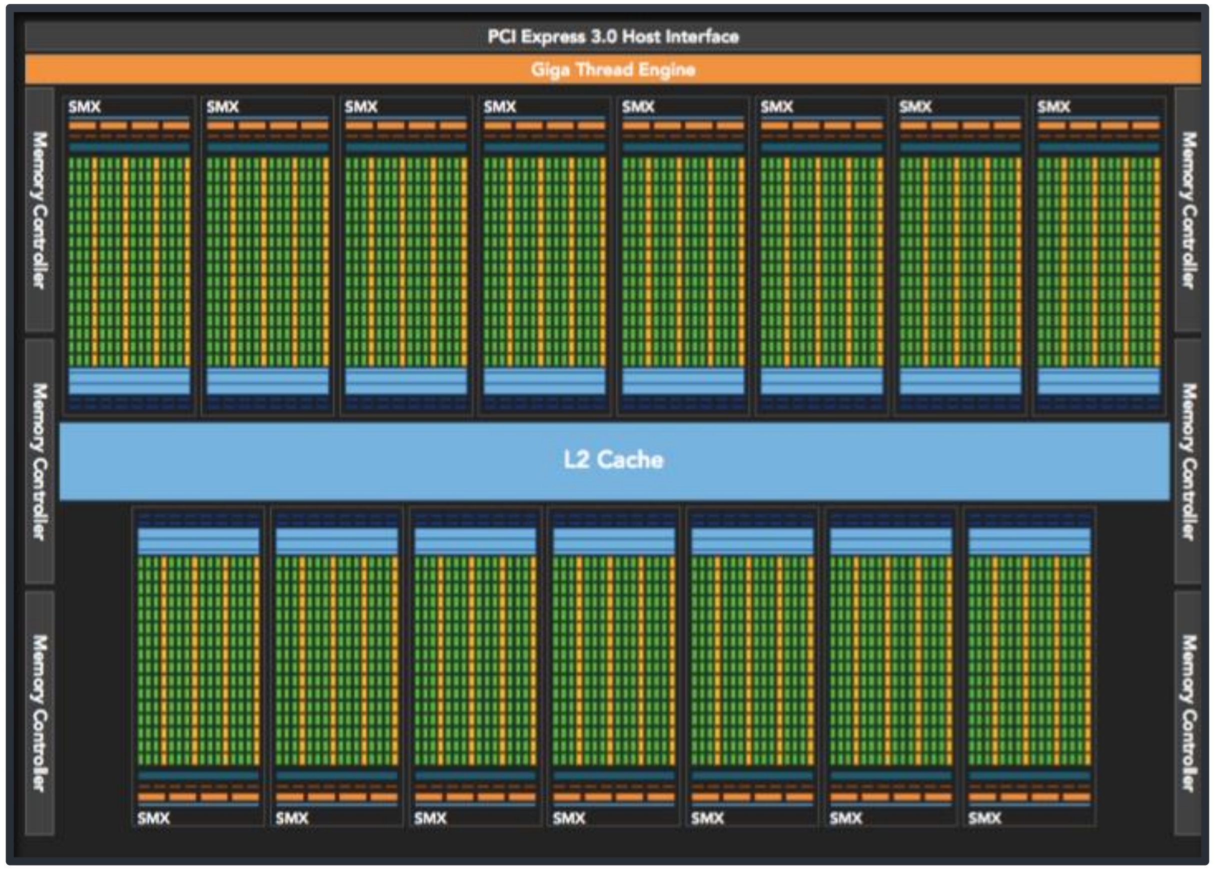
- Bus_Width - 320 bit
- Chipset_Manufacturer - NVIDIA
- Chipset_Model - K20
- Chipset_Line - Tesla

Kepler Architecture

- Dynamic Parallelism: We will talk about this in a future session.

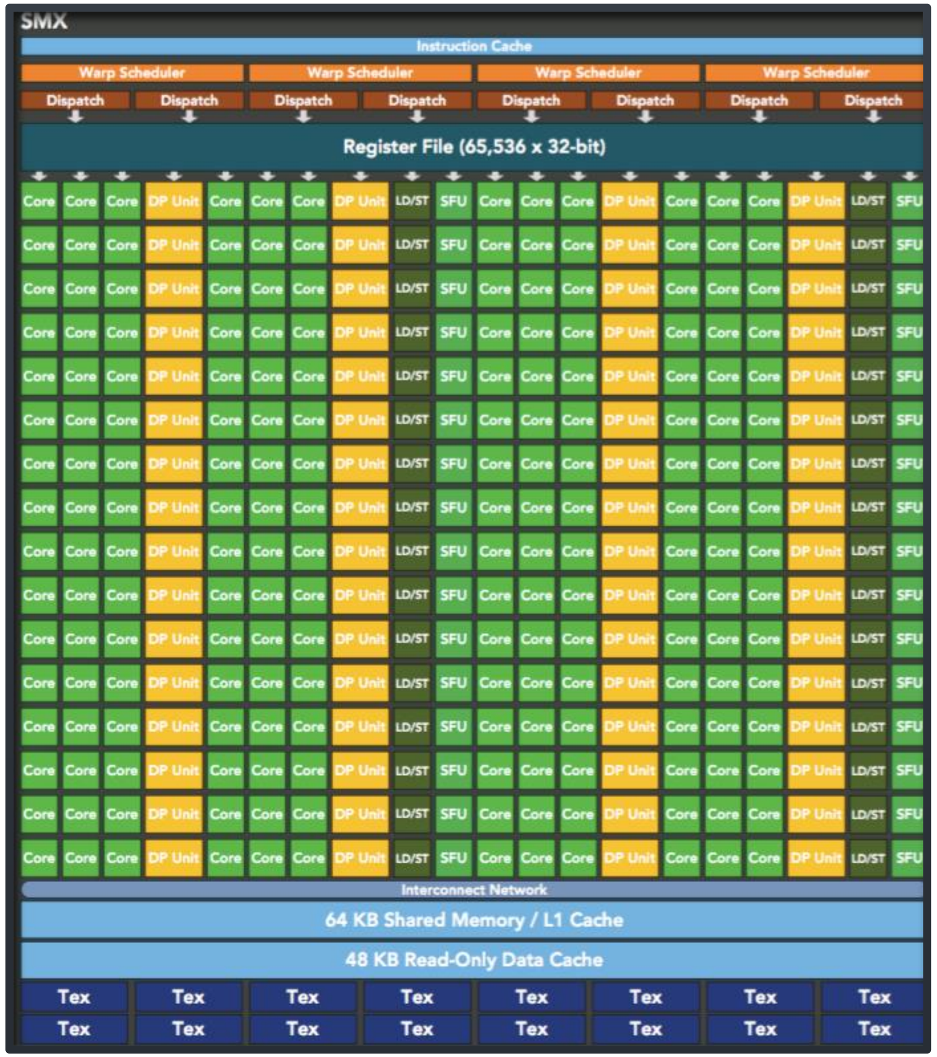


Kepler Architecture



Kepler Architecture

- Each SMX contains:
  - 192 CUDA cores
  - 64 double-precision units
  - 32 special function units (SFU)
  - 32 Load/Store (LD/ST)


SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

Kepler Architecture - SMX

- 4 warp schedulers
  - Enabling 4 warps to be issued and execute at the same time
- Each SM can issue a maximum of 64 warps (e.g. a total threads = 64 * 32 = 2048 threads resident at the same time)

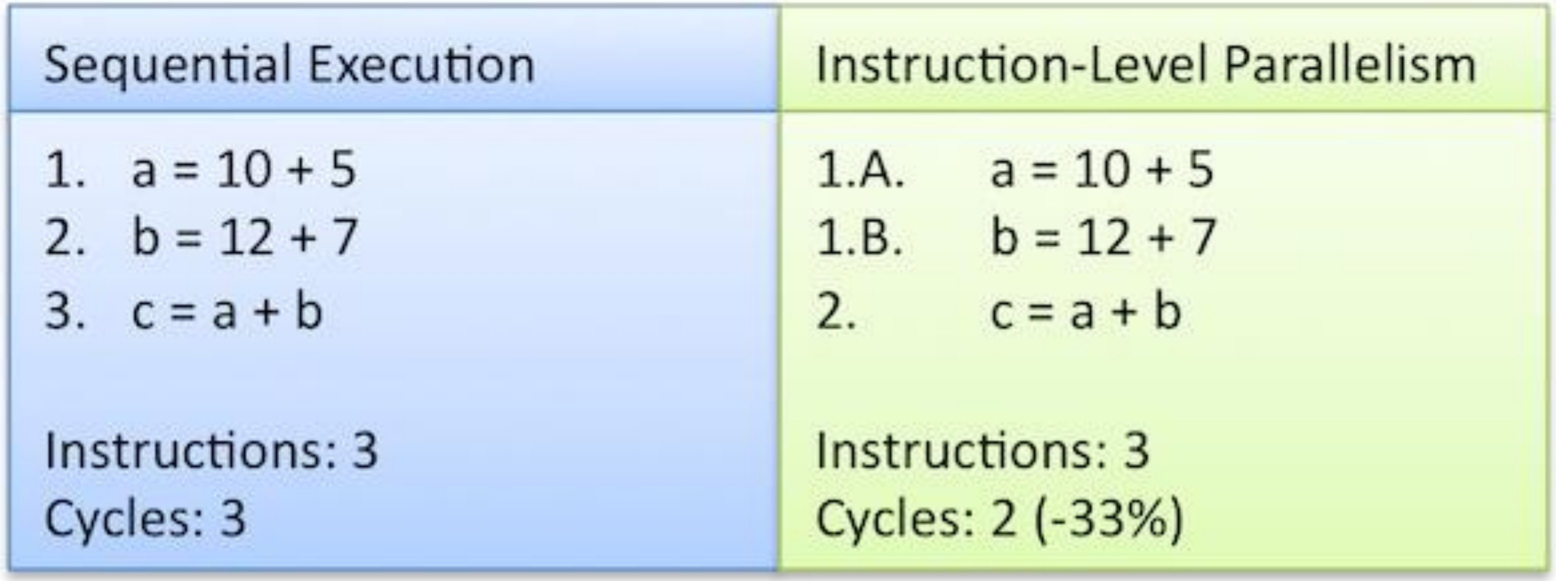
CUDA Execution Model

CUDA Execution Model

- When a kernel Grid is launched:
  - Thread-blocks are divided among the SMs for execution.
  - Threads on the same blocks will be executed simultaneously (logically speaking).
  - Multiple blocks could be assigned to the same SM but that doesn't mean they will be executed simultaneously, it will depend on the available resources.

CUDA Execution Model

- Instruction on the single thread are pipelined to leverage Instruction Level Parallelism(ILP). In addition to the thread level parallelism



Sequential Execution	Instruction-Level Parallelism
1. a = 10 + 5 2. b = 12 + 7 3. c = a + b Instructions: 3 Cycles: 3	1.A. a = 10 + 5 1.B. b = 12 + 7 2. c = a + b Instructions: 3 Cycles: 2 (-33%)

CUDA Execution Model

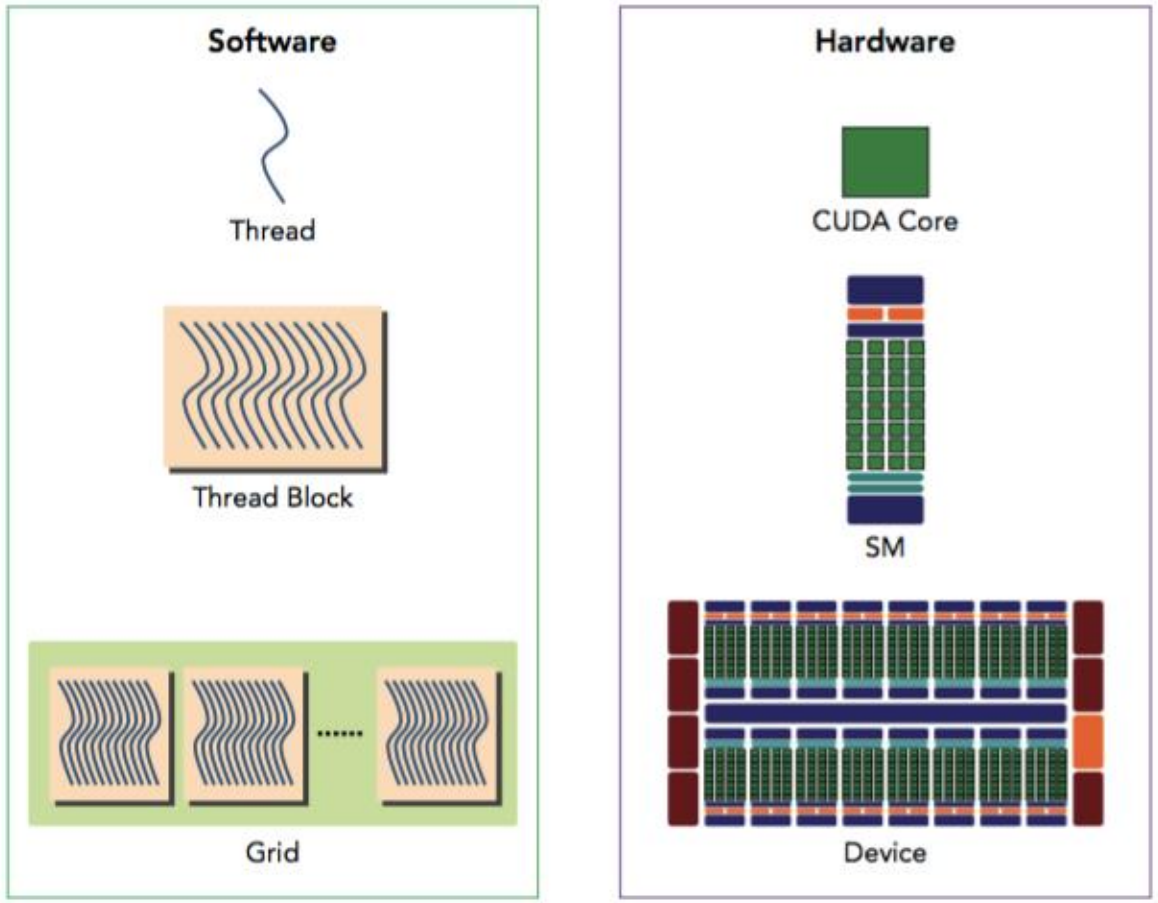
- CUDA uses Single Instruction Multiple Thread (SIMT)
  - Threads will be grouped into **warp** sizes (32 threads per warp)
  - **All threads in a warp execute the same instruction at the same time**
- Each SM will partition the blocks into warps and then schedule them for execution depending on available hardware resources.
- It is possible that threads on the same warp could have different behavior.

CUDA Execution Model

- SIMT (Single Instruction Multiple Thread) offers:
  - Each thread has its own instruction address counter.
  - Each thread has it own register state.
  - Each thread can have an independent execution path.

CUDA Execution Model

- Logical and Physical View:



CUDA Execution Model

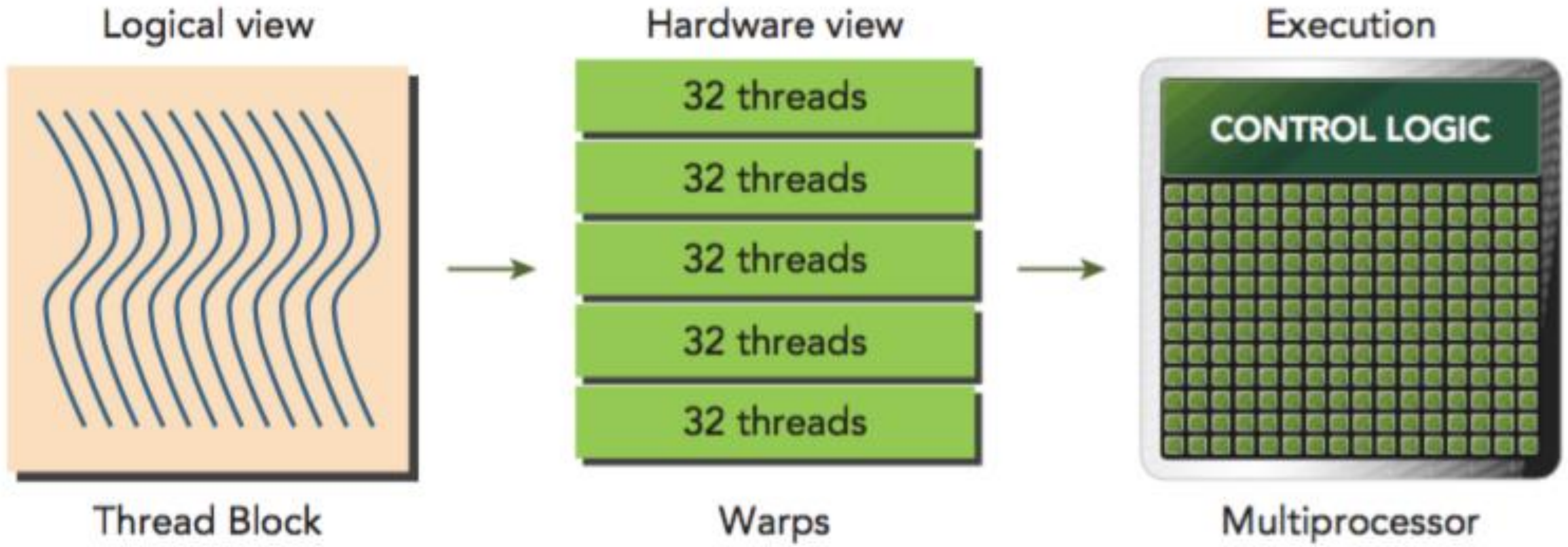
- Logically all thread in a block run simultaneously, physically they **MIGHT NOT**!
- We have the ability to synchronize threads inside the block to ensure consistent access to shared resources (such as shared memory).
- **THERE IS NO** explicit inter-block synchronization

CUDA Execution Model

- Number of active warps will be limited by physical resources.
- if a warp is idle for any reason, SM is free to schedule another warp (from any thread-block that exist already on the SM).

Warp Execution

- Warp → it's the basic unit of execution
- Each thread in a warp must executed the same instruction.



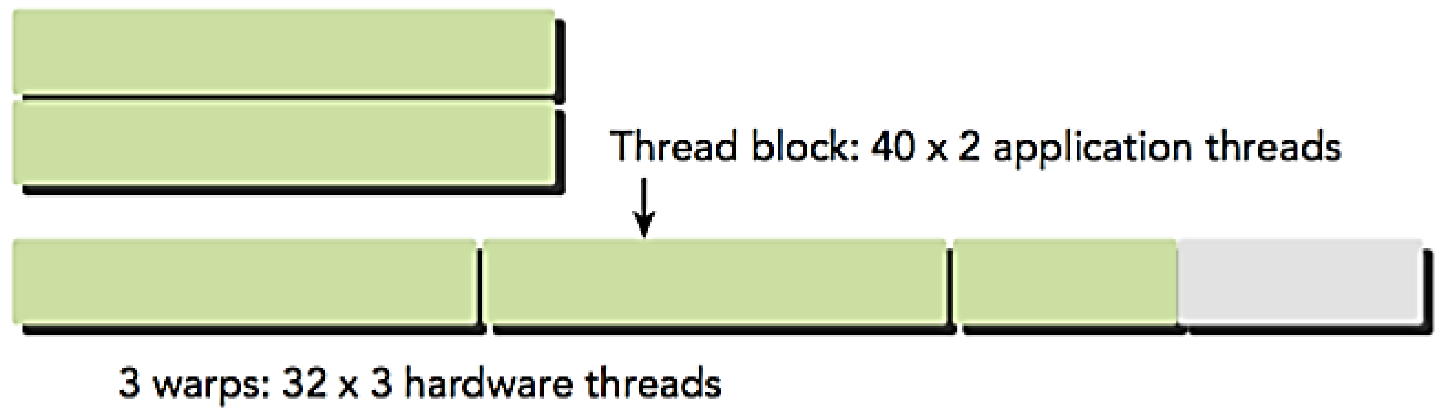
Warp Execution

- Blocks can be 3D (x, y, and z dimension). However from the hardware point of view, we can see the threads as one dimension.
- Threads are grouped into warps based on the built-in variable threadIdx
  - E.g. blocks of 128 threads will be partition on 4 warps as follow:

```
Warp 0: thread  0, thread  1, thread  2, ... thread 31
Warp 1: thread 32, thread 33, thread 34, ... thread 63
Warp 3: thread 64, thread 65, thread 66, ... thread 95
Warp 4: thread 96, thread 97, thread 98, ... thread 127
```

Warp Execution

- Not taking warp size into account can lead to misuse
  - If your block has a certain number of threads which is not a multiple of the warp size, then threads on a warp will be wasted.
    - E. g. a threadBlock of 80 threads.

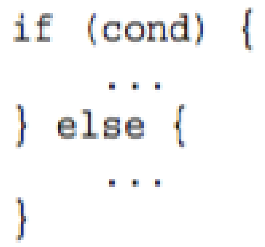

Thread block: 40 x 2 application threads
3 warps: 32 x 3 hardware threads

Warp Divergence

- All threads on a warp MUST execute the same instruction.
- What happen when there is a branch behavior?

CPU

It has complex hardware to specifically handle branch prediction



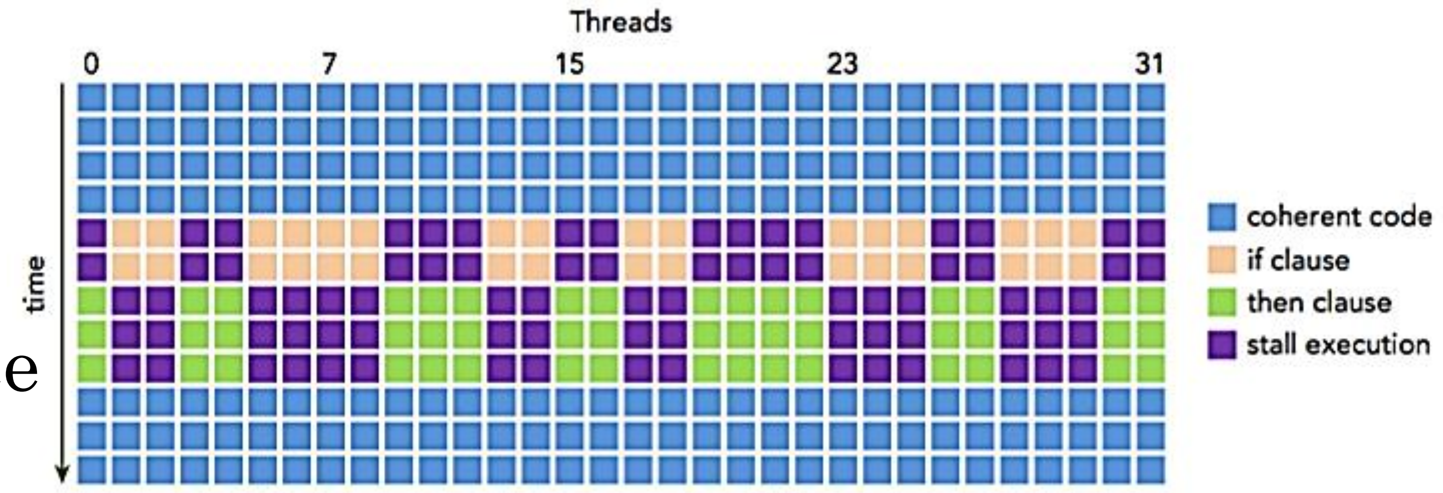
```
if (cond) {
    ...
} else {
    ...
}
```

GPU

No complex branch prediction. Stalling of threads in a warp

Warp Divergence

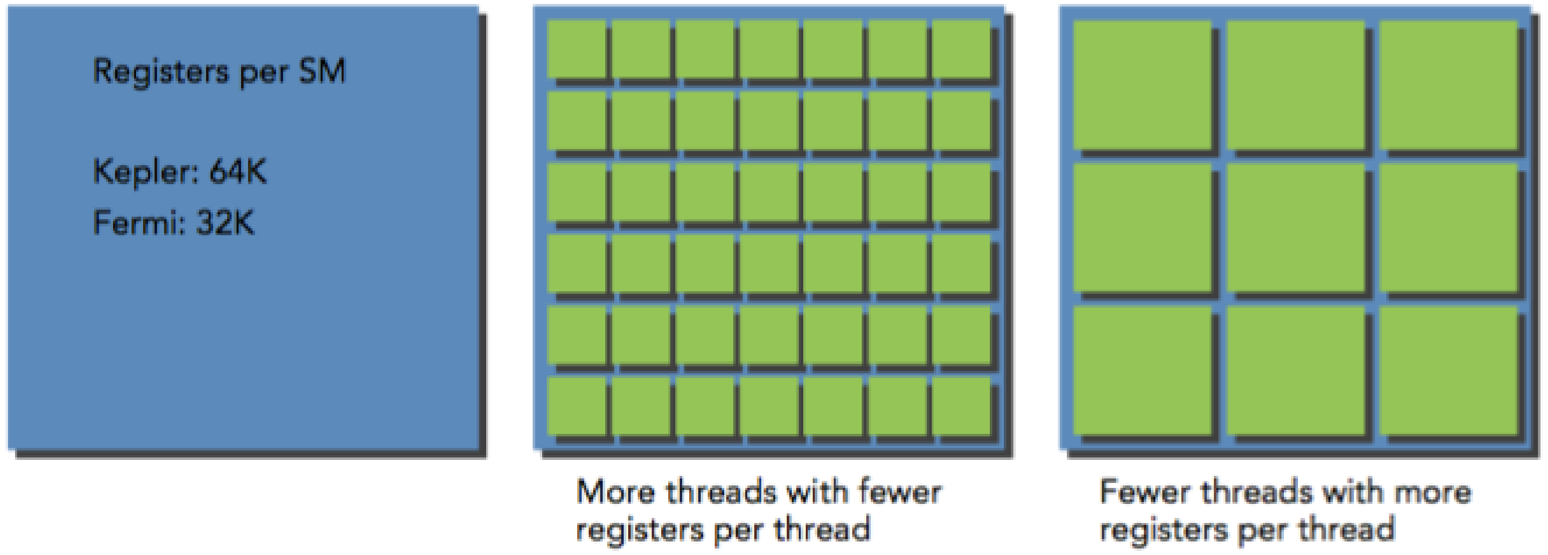
- Avoid branch divergence!
- Stalling threads is never a good thing
- Only threads on the same warp can decrease performance by divergence



Warp Execution

- Resources on a local context:
  - Program Counters
  - Registers
  - Shared Memory

NOTE: If there is not enough resources for at least one block then the launch of the kernel will fail


Registers per SM
Kepler: 64K
Fermi: 32K
More threads with fewer registers per thread
Fewer threads with more registers per thread

Warp Execution

- Active block: when resources such as registers and shared memory have been allocated to it.
- Active warp: warps that belong to the active blocks
  - Selected warp. Warp that is actively executing
  - Eligible warp. Warp that is ready for execution but is not currently executing.
  - Stalled warp. Warp that is not ready for execution.

Profiling Tools

Measuring Time

- Why do we use GPUs?

Measuring Time

- Why do we use GPUs?
  - Performance

Measuring Time

- Why do we use GPUs?
  - Performance
- How do we measure performance?

Measuring Time

- Why do we use GPUs?
  - Performance
- How do we measure performance?
  - Best way: use event handlers provided by CUDA.

CUDA Events

- How do we measure time with CUDA?
- Variables:

```
//Time variables
        cudaEvent_t start, stop;
        float time;
        cudaEventCreate(&start);
        cudaEventCreate(&stop);
```

- Measure your time performance:

```
        cudaEventRecord(start, 0);
// Put your code here…. (Kernel call)
        cudaEventRecord(stop, 0);
        cudaEventSynchronize(stop); // Wait for
event to happen
//Display time
        cudaEventElapsedTime(&time, start, stop);
        printf("Parallel Job time: %.2f ms", time);
```

Second parameter associated to stream, usually stream 0

Cuda Events Example

```
//create events
cudaEvent_t event1, event2;
cudaEventCreate(&event1);
cudaEventCreate(&event2);

//record events around kernel launch
cudaEventRecord(event1, 0); //where 0 is the default stream
kernel<<<grid,block>>>(...); //also using the default stream
cudaEventRecord(event2, 0);

//synchronize
cudaEventSynchronize(event1); //optional
cudaEventSynchronize(event2); //wait for the event to be executed!

//calculate time
float dt_ms;
cudaEventElapsedTime(&dt_ms, event1, event2);
```

Profiling Performance

- Now that we can measure performance, what if our code is taking too long?

Profiling Performance

- Now that we can measure performance, what if our code is taking too long?
- How can we improve our code?

Profiling Performance

- Now that we can measure performance, what if our code is taking too long?
- How can we improve our code?
- Any guideline?

Profiling Performance

- Now that we can measure performance, what if our code is taking too long?
- How can we improve our code?
- Any guideline?
  - Use profiling tools

NVPROF

- Command line profiler
  - Compute time in each kernel
  - Compute memory transfer time
  - Collect metrics and events
  - Support complex process hierarchy’s
  - Collect profiles for NVIDIA Visual Profiler
  - No need to recompile

NVPROF

- Compile binary with some information so nvprof / nvvp can track line numbers

```
nvcc -lineinfo ${your flags and files, etc}
```

NVPROF

- Instructions:

1. Collect profile information for the program by running
    1. nvprof ./exec
2. View available metrics
    1. nvprof --query-metrics
3. View global load/store efficiency
    1. nvprof -metrics gld_efficiency,gst_efficiency ./exec
4. Store a timeline to load in NVVP
    1. nvprof -o profile.timeline ./exec
5. Store analysis metrics to load in NVVP
    1. nvprof –o profile.metrics –analysis-metrics ./exec

NVPROF

- Instructions:

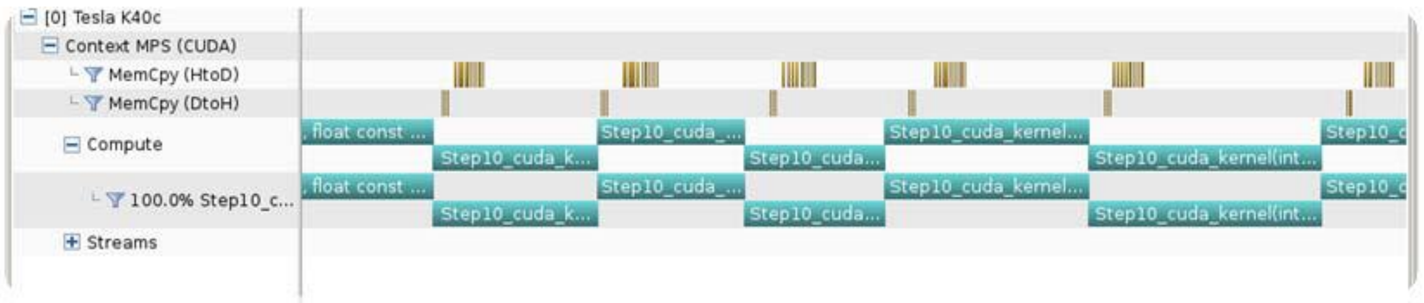
1. Collect profile information for the program by running
   1. nvprof ./exec
2. View available metrics
   1. nvprof --query-metrics
3. View global load/store efficiency
   1. nvprof -metrics gld_efficiency,gst_efficiency ./exec
4. Store a timeline to load in NVVP
   1. nvprof -o profile.timeline ./exec
5. Store analysis metrics to load in NVVP
   1. nvprof –o profile.metrics –analysis-metrics ./exec

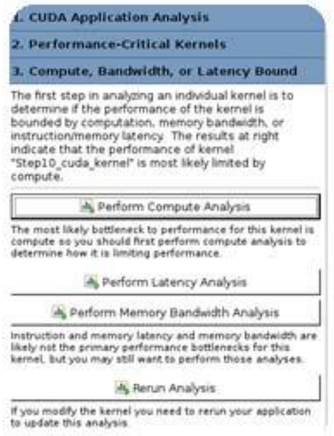
Timeline of CUDA runtime calls, kernel execution times, etc.
Basically no run time overhead

Detailed performance data from each kernel execution.
Large run time overhead

NVVP

- Example




1. CUDA Application Analysis

2. Performance-Critical Kernels

3. Compute, Bandwidth, or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "Step10_cuda_kernel" is most likely limited by compute.

Perform Compute Analysis

The most likely bottleneck to performance for this kernel is compute so you should first perform compute analysis to determine how it is limiting performance.

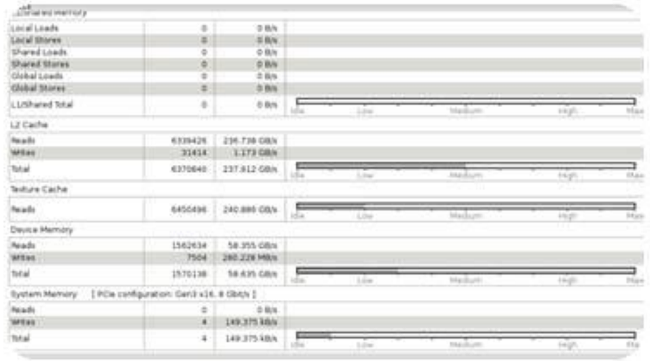
Perform Latency Analysis

Perform Memory Bandwidth Analysis

Instruction and memory latency and memory bandwidth are likely not the primary performance bottlenecks for this kernel, but you may still want to perform those analyses.

Rerun Analysis

If you modify the kernel you need to rerun your application to update this analysis.


L1/Shared Memory

Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Global Loads	0	0 B/s	
Global Stores	0	0 B/s	
L1/Shared Total	0	0 B/s	Idle Low Medium High Max

L2 Cache

Reads	6339426	236.738 GB/s	
Writes	31414	1.173 GB/s	
Total	6370840	237.912 GB/s	Idle Low Medium High Max

Texture Cache

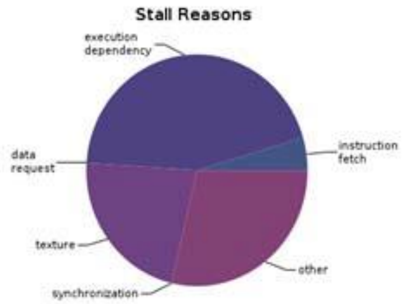
Reads	6450496	240.886 GB/s	Idle Low Medium High Max

Device Memory

Reads	1562634	58.355 GB/s	
Writes	7504	280.228 MB/s	
Total	1570138	58.635 GB/s	Idle Low Medium High Max

System Memory [PCIe configuration: Gen3 x16, 8 Gbit/s]

Reads	0	0 B/s	
Writes	4	149.375 kB/s	
Total	4	149.375 kB/s	Idle Low Medium High Max



NVVP

- Instructions
- Import nvprof profile into NVVP
  - Launch nvvp
  - Click file/ import/ nvprof/ next/ single process/ next /browse
    - Select profile.timeline
  - Add metrics to timeline
    - Click on 2nd browse
    - Select profile.metrics
  - Click finish
- Expore timeline
  - Control + mouse drag in timeline to zoom in
  - Control + mouse drag in measure bar (on top) to measure time

Takeaways

- How to get close to peak performance?
  - Potential for floating point performance on GPUs is huge
    - Integers less so
    - Difficult to achieve!
  - Use memories efficiently:
    - Avoid unnecessary data transfers
    - Keep data being accessed often close to the processing elements
    - Use registers and shared memory
  - Avoid control flow divergence
    - Very few if statements

Takeaways

- Writing a CUDA kernel is becoming easier, but getting good performance is not.
- Know the tools you have available. Profiling is key to performance
- Fitting your application to the GPU memory hierarchy is critical for performance
- Resources are not infinite, optimization without thinking about the available resources could adversely affect performance.