



# GPU Programming

(in CUDA)

Summer 2019

**Designed by Julian Gutierrez, Presented by Nicolas Agostini**

---

Session 3





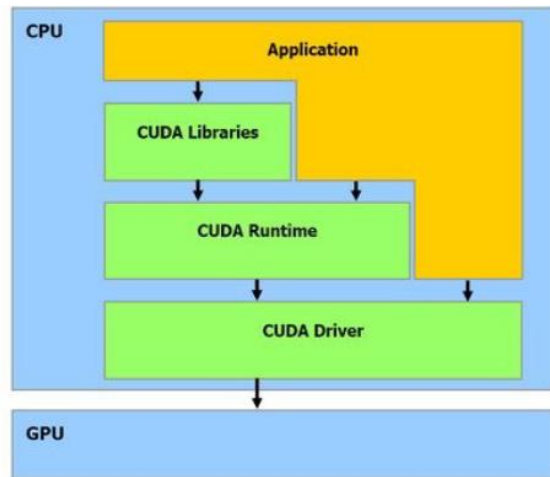
- CUDA Basics
- CUDA Execution Model
- Profiling Tools

# CUDA Basics

- 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 is a general purpose parallel platform
- CUDA is a standard ANSI C-like language
- CUDA's Application Programming Interface (API) manages:
  - Devices
  - Memory
  - Synchronization
  - Etc.

- CUDAs 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 Basics



C Code

```
for (int i=0;i < MAXi;i++)  
  for(int j=0;j< MAXj;j++){  
    ...code that uses i and j....  
  }
```

CUDA Code

```
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

## Concepts

- Host
  - CPU
  - Executes the main function and any other CPU related jobs
- Device
  - GPU
  - Executes the kernel functions
- CUDA Kernels: data-parallel function
  - A kernel is a function callable from the host and executed on the CUDA device.
  - The sequential code executed by each thread.
  - It runs “simultaneously” by many threads in parallel.
- CUDA compiler: nvcc
  - Separates the device code from the host code during compilation process.



## Concepts

- 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`)

## Concepts

- 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`)

## Concepts

- Flow of a CUDA program
  1. Copy data from CPU to GPU  
(`cudaMemcpyHostToDevice`)

- 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 )
```

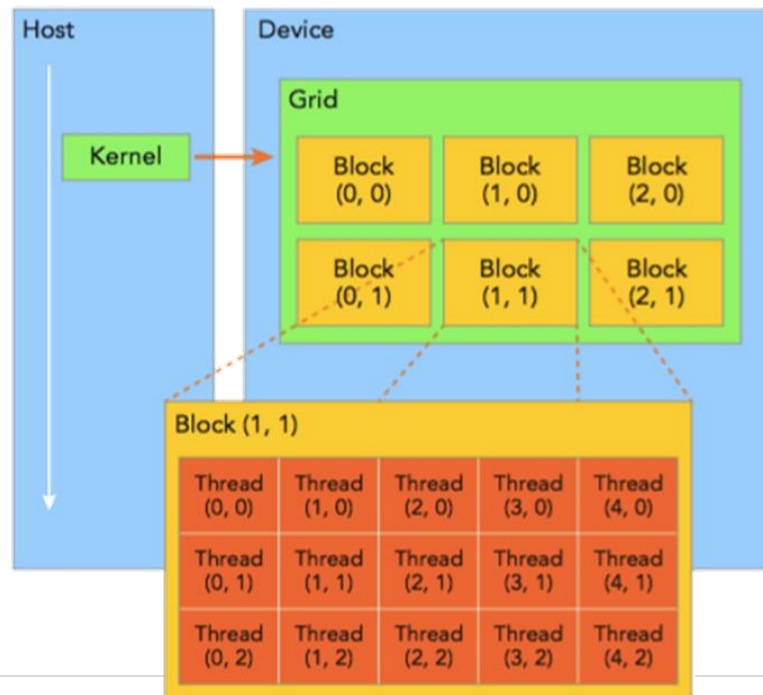
```
cudaMemcpyHostToDevice  
cudaMemcpyDeviceToHost
```

## Concepts

- 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`)

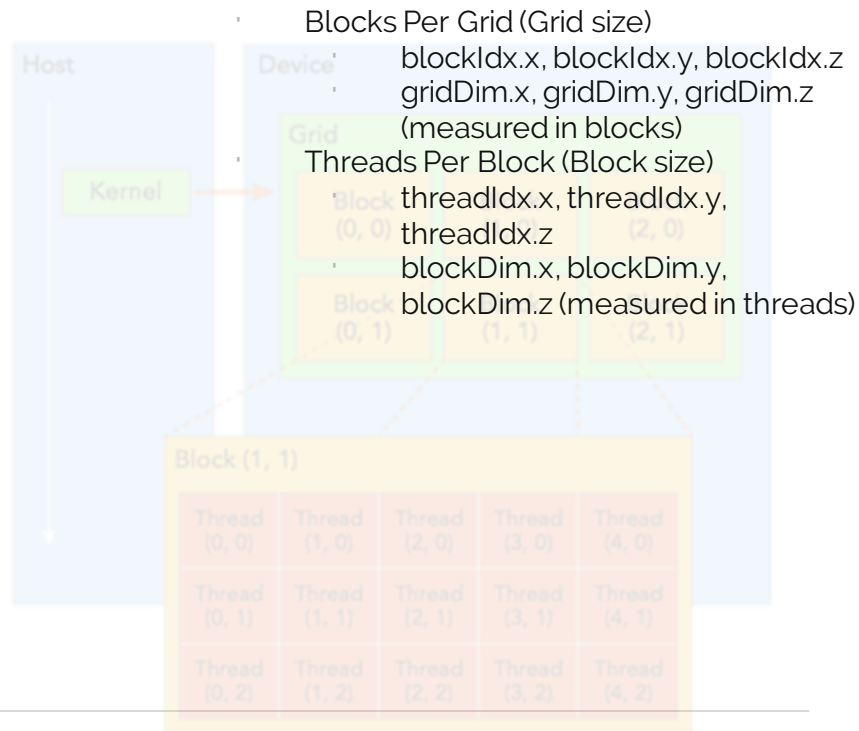
## Concepts

- Flow of a CUDA program
  1. Copy data from CPU to GPU (cudaMemcpyHostToDevice)
  2. **Invoke Kernels to operate over the GPU data (asynchronous call)**
    - Think about adding 2 vectors with a size of 1000 together:  
 $A_i + B_i = C_i$
    - How many threads would you ideally use?
    - 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?



## Concepts

- Flow of a CUDA program
  - Copy data from CPU to GPU (cudaMemcpyHostToDevice)
  - Invoke Kernels to operate over the GPU data (asynchronous call)**
    - Thread organization
      - A Kernel call creates a Grid.
      - Multiple blocks are combined to form a grid. **All the blocks in the same grid contain the same number of threads.**
      - The threads of a block can be indexed using 1 Dimension (x), 2 Dimensions (x,y) or 3 Dimensions indexes (x,y,z)



## Concepts

- Flow of a CUDA program
  - Copy data from CPU to GPU  
(`cudaMemcpyHostToDevice`)
  - Invoke Kernels to operate over the GPU data (asynchronous call)**
    - How do we define the number of blocks per grid, and threads per block?
      - Consider the nature of the problem
      - Consider the nature of the GPU architecture
      - Use `dim3` data type

Indexing from 0  
Imagine `nElem=31`. Below grid =1. (As expected)  
Imagine `nElem=32`. Below grid 2. (As expected)  
The below math just works fine

```
dim3 block(32);  
dim3 grid(((nElem-1)/block.x)+1);
```

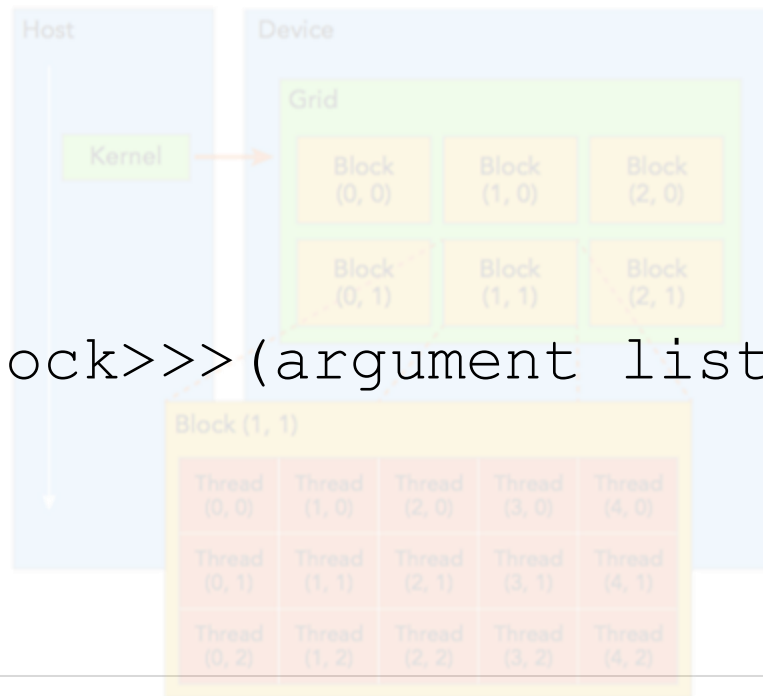
Grid Size is a rounded up multiple of blocks, based on the parallel architecture of the GPU

## Concepts

- Flow of a CUDA program
  - Copy data from CPU to GPU (cudaMemcpyHostToDevice)
  - Invoke Kernels to operate over the GPU data (asynchronous call)**
    - 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.



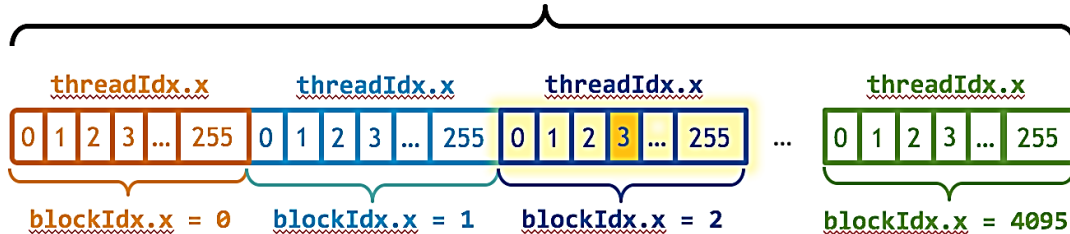


## Concepts

- Kernel Invocation example
  - How many threads are executed?  $4096 \times 256$
  - How do we tell each thread to process one unique element?

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

gridDim.x = 4096



$$\text{index} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$$

blockDim.x=256

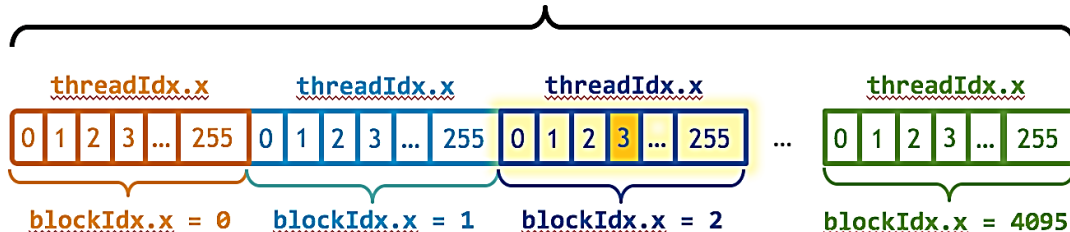
index calculated very similar to if image pixel number were being calculated when image is saved in row major format

## Concepts

- Kernel Invocation example
  - How many threads are executed?
  - How do we tell each thread to process one unique element?

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

gridDim.x = 4096



$\text{index} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$

index = (2) \* (256) + (3) = 515

## Concepts

- 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

## Example

```
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);
}
```

## Example

```
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);
```

Allocation on the GPU

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

Transfer Data CPU to GPU

```
    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);
```

Kernel Call

```
    cudaMemcpy(c, devC, sizeC, cudaMemcpyDeviceToHost);
```

Transfer back to CPU

```
    //Freeing device matrices.
```

```
    cudaFree(devA); cudaFree(devB); cudaFree(devC);
```

```
}
```

## 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;  
    }  
}
```

## Example

- **Compilation**

```
nvcc matrixMul.cu -o matrixMul
```

- **Execution**

```
./matrixMul
```

```
Sequential Job Time: 588.227987 ms
```

```
Parallel Job Time: 108.647108 ms
```

# CUDA Execution Model



## Example

- Why Do we need to learn about the GPU architecture and the CUDA execution model?
  - To understand how to efficiently use the selected configuration
  - Find a guideline to choose a proper grid/block configuration
- We will go over the architecture while explaining how a kernel executes on the GPU
- We will focus on the Kepler architecture

# CUDA Execution Model



## Kepler Architecture - SMX

- The Kepler Architecture
  - Divided into SMXs
  - Contains a big L2 cache shared between all SMXs



## Kepler Architecture - SMX

- Scalable array of Streaming Multiprocessors
  - CUDA Cores
  - Shared Memory
  - Register File
  - Load/Store Units
  - Special Function Units
  - Warp Scheduler
- Streaming Multiprocessor Extreme (SMX)
  - Each SM is designed to support the execution of hundreds of threads.
  - Kepler K40
    - 15 Multiprocessors
    - Number of processors: 2880
    - Each SMX contains
      - 192 CUDA cores
      - 64 Double Precision units
      - 32 SFU
      - 32 load/store units

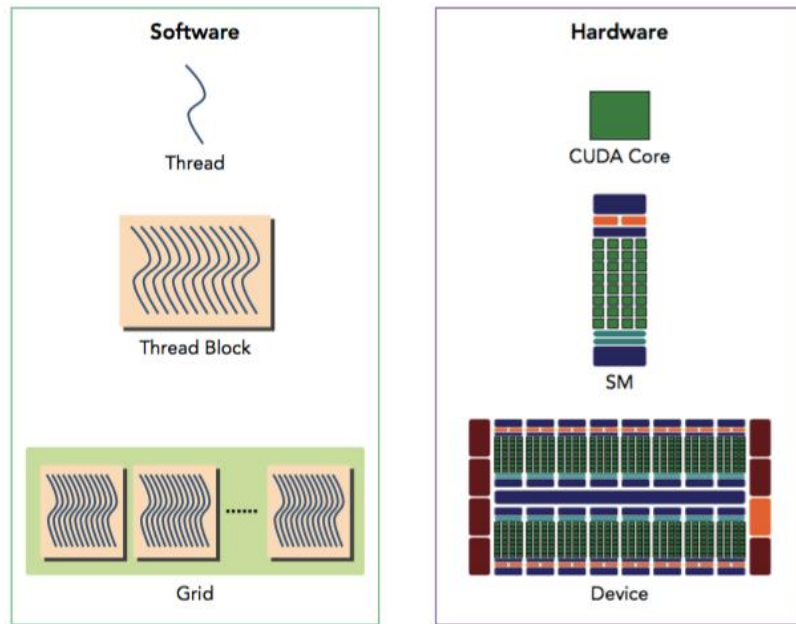
## Launching a Kernel

- 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



## Logical and Physical View



# CUDA Execution Model



## Launching a Kernel

```
kernel<<<4, 256>>>(pointer1);
```



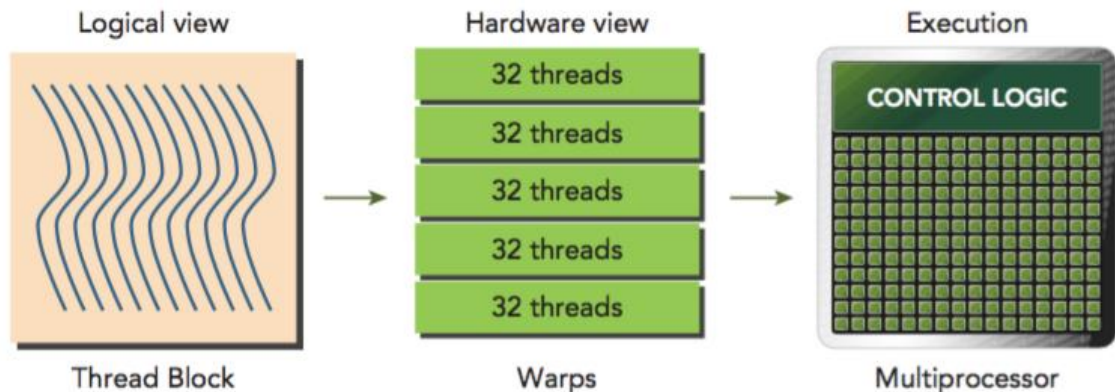
## Kepler Architecture - SMX

- CUDA uses Single Instruction Multiple Thread (SIMT)
  - Threads will be grouped into **warps** (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.**

**In case of conditional branches. See slide 40**

## Warp Execution

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





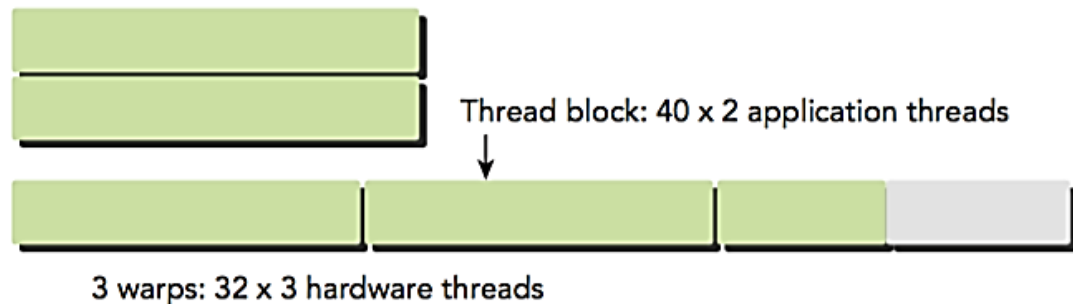
## Warp Execution

- 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 in 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.

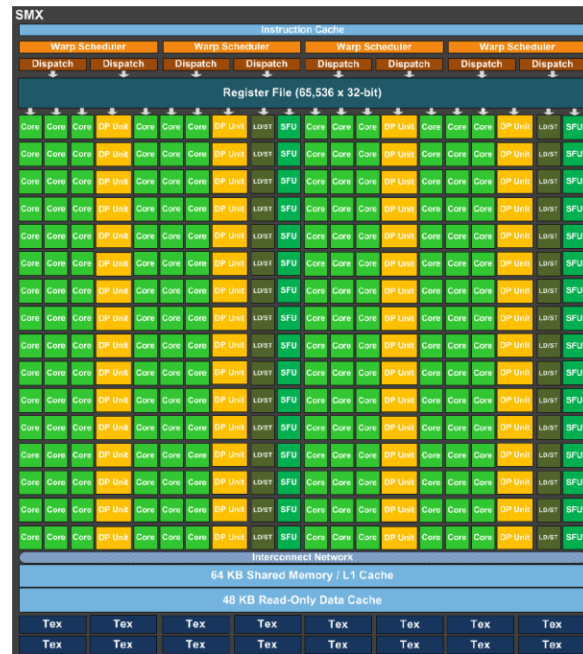
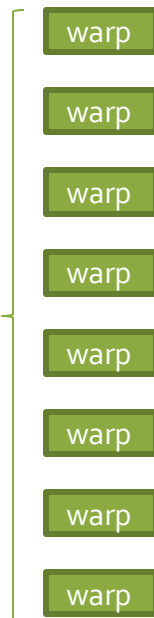
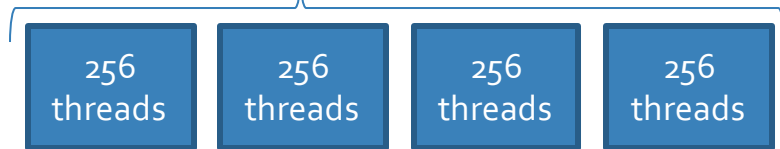


# CUDA Execution Model



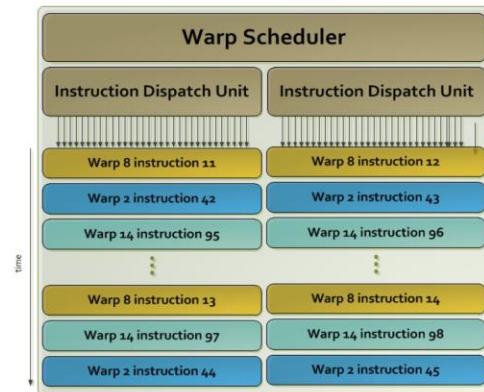
## Warps

```
kernel<<<4, 256>>>(pointer1);
```



## Warps

- 4 warp schedulers
  - Enabling 4 warps to be issued and execute at the same time
  - two independent instructions per warp can be dispatched each cycle.
- Each SM can issue a maximum of 64 warps (e.g. a total threads =  $64 * 32 = 2048$  threads resident in the SM)



## Warps

- 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.

## Warps

- 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 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

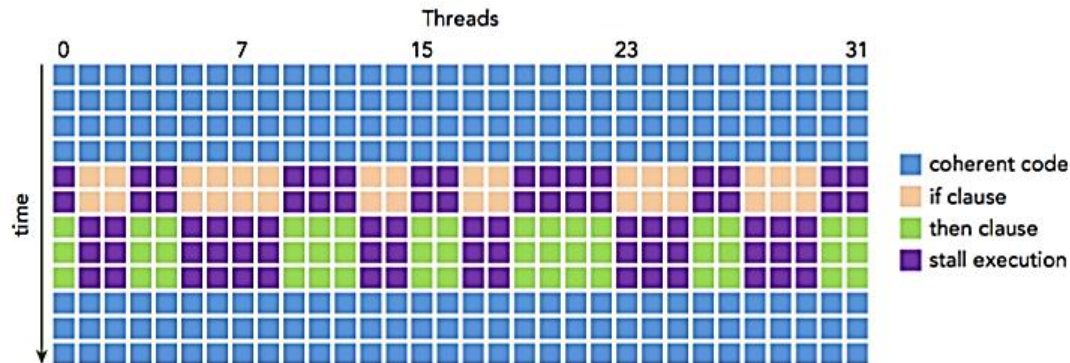
```
if (cond) {  
    ...  
} else {  
    ...  
}
```

### **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





## Other Notes

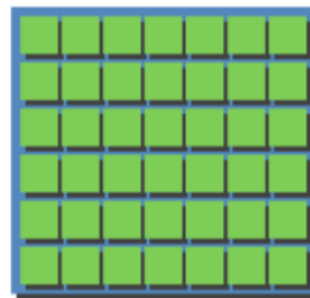
- Instruction Level Parallelism
  - Instruction on the single thread are pipelined to leverage Instruction Level Parallelism( ILP), in addition to the thread level parallelism

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

## Other Notes

- 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



More threads with fewer registers per thread



Fewer threads with more registers per thread

# CUDA Execution Model



## Kepler General Information

| Details                                   | Kepler GK110      |
|---|-------------------|
| Compute Capability                        | 3.5               |
| Threads / Warp                            | 32                |
| Max Warps / Multiprocessor                | 64                |
| Max Threads / Multiprocessor              | 2048              |
| Max Thread Blocks / Multiprocessor        | 16                |
| 32-bit Registers / Multiprocessor         | 65536             |
| Max Registers / Thread                    | 255               |
| Max Threads / Thread Block                | 1024              |
| Shared Memory Size Configurations (bytes) | 16K<br>32K<br>48K |

# Profiling Tools

## Measuring Performance

- Why do we use GPUs?
  - Performance
- How do we measure performance?
  - We measure execution time.
  - Use event handlers provided by CUDA.

## CUDA Events

- Specific data types:

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

Second parameter  
associated to stream,  
usually stream 0

- Using the data types to measure time:

```
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);
```

## CUDA Events

```
//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?
- How can we improve our code?
- Best 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

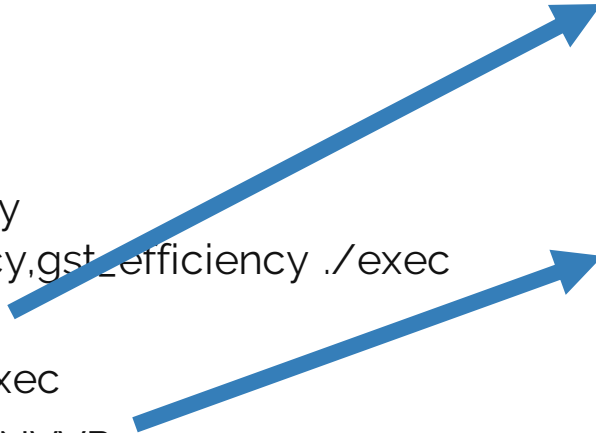
- Compile binary with some information so nvprof / nvvp can track line numbers

```
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`



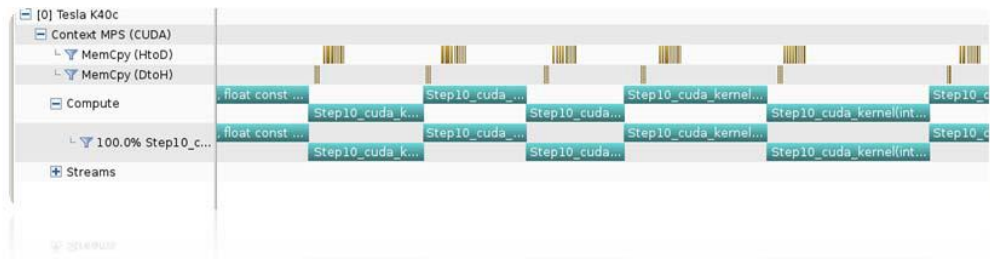
Timeline of CUDA runtime calls, kernel execution times, etc. Basically no run time overhead

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

# Profiling Tools



NVVP



**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.

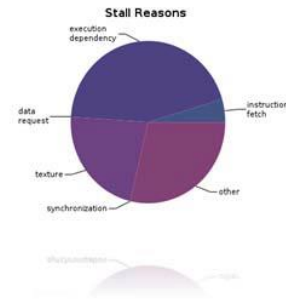
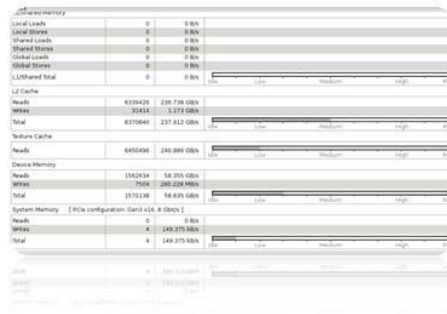
[Perform Latency Analysis](#)

[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.

[Run Analysis](#)

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



## NVVP

- Note: We will not use NVVP today (or tomorrow)
- Instructions
  1. Import nvprof profile into NVVP
    1. Launch nvvp
    2. Click file/ import/ nvprof/ next/ single process/ next /browse
      1. Select profile.timeline
    3. Add metrics to timeline
      1. Click on 2nd browse
      2. Select profile.metrics
    4. Click finish
  2. Explore timeline
    1. Control + mouse drag in timeline to zoom in
    2. Control + mouse drag in measure bar (on top) to measure time

- 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.