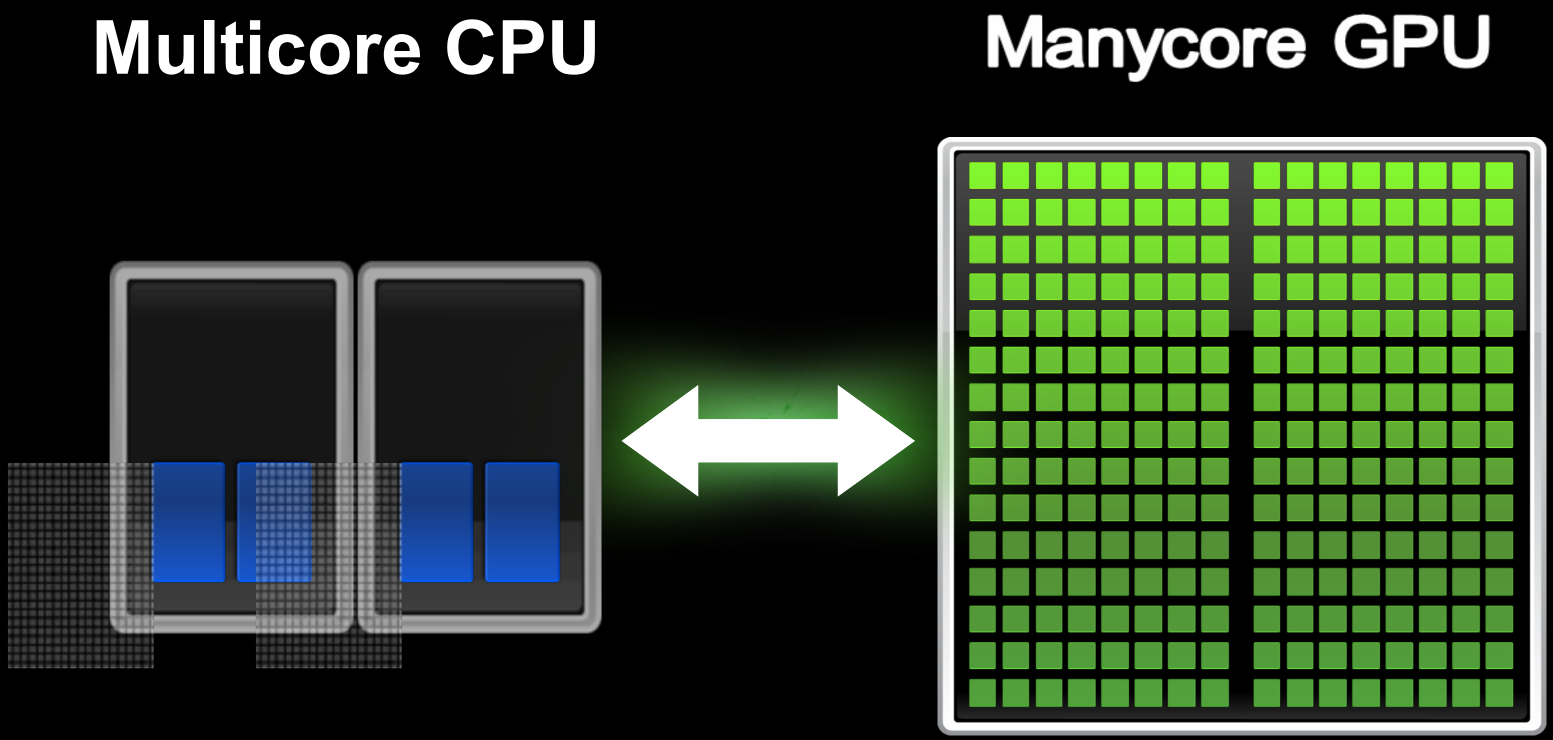
Προγραμματισμός Συστημάτων Υψηλών Επιδόσεων (ΗΥ421 / ΜΔΕ646)

Εισαγωγή στο Μοντέλο Προγραμματισμού CUDA

Heterogeneous Computing



C for CUDA

- Philosophy: provide minimal set of extensions necessary to expose power
-
- Function qualifiers:

```
__global__ void my_kernel() { }
__device__ float my_device_func() { }
```

- Variable qualifiers:

```
__constant__ float my_constant_array[32];
__shared__   float my_shared_array[32];
```

- Execution configuration:

```
dim3 grid_dim(100, 50);  // 5000 thread blocks
dim3 block_dim(4, 8, 8); // 256 threads per block
my_kernel <<< grid_dim, block_dim >>> (...); // Launch kernel
```

- Built-in variables and functions valid in device code:

```
dim3 gridDim;   // Grid dimension
dim3 blockDim;  // Block dimension
dim3 blockIdx;  // Block index
dim3 threadIdx; // Thread index
void __syncthreads(); // Thread synchronization
```

Example: vector_addition

Device Code

```
// compute vector sum c = a + b
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // elided initialization code
    ...
    // Run N/256 blocks of 256 threads each
    vector_add<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: vector_addition

```
// compute vector sum c = a + b
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

Host Code

```
int main()
{
    // elided initialization code
    ...
    // launch N/256 blocks of 256 threads each
    vector_add<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: Initialization code for vector_addition

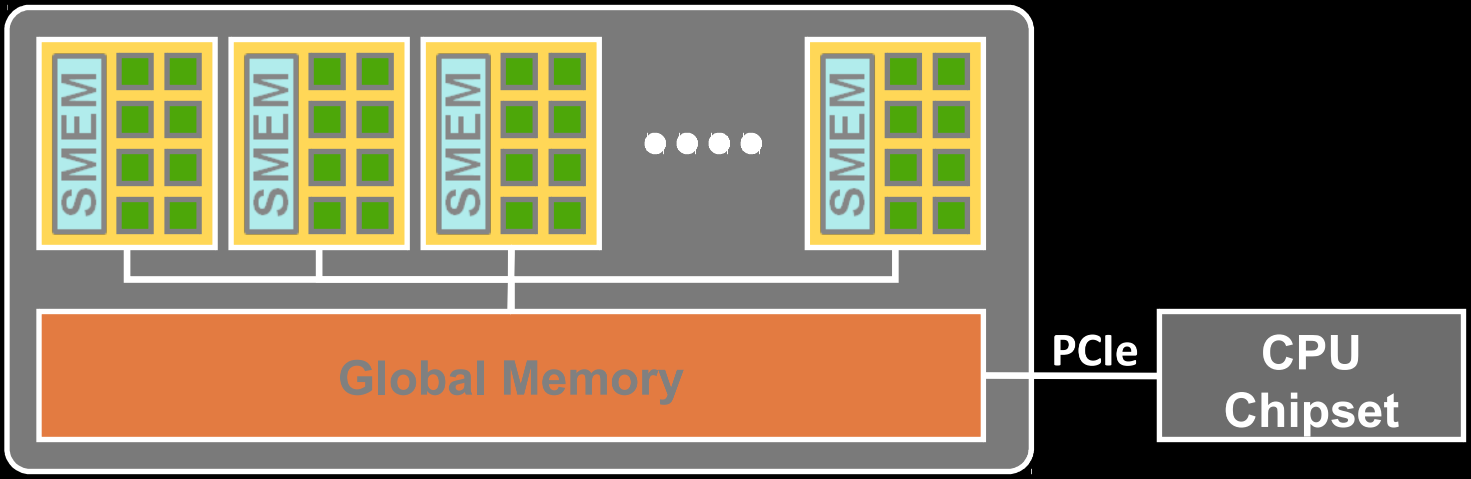
```
// allocate and initialize host (CPU) memory
float *h_A = …,   *h_B = …;

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

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

// launch N/256 blocks of 256 threads each
vector_add<<<N/256, 256>>>(d_A, d_B, d_C);
```

BASIC KERNELS AND EXECUTION ON GPU

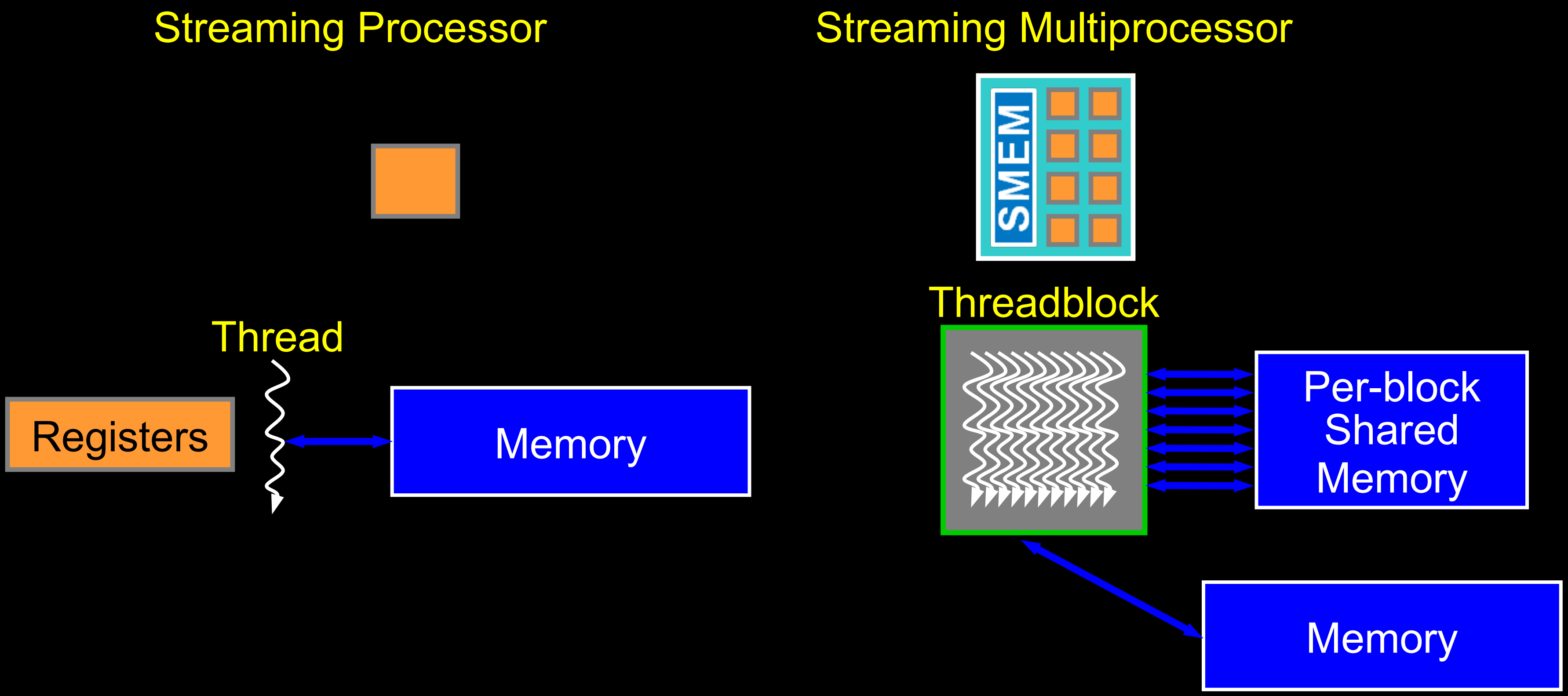
CUDA Programming Model

- Parallel code (kernel) is launched and executed on a device by many threads
- Launches are hierarchical
  - Threads are grouped into blocks
  - Blocks are grouped into grids
- Familiar serial code is written for a thread
  - Each thread is free to execute a unique code path
  - Built-in thread and block ID variables

High Level View

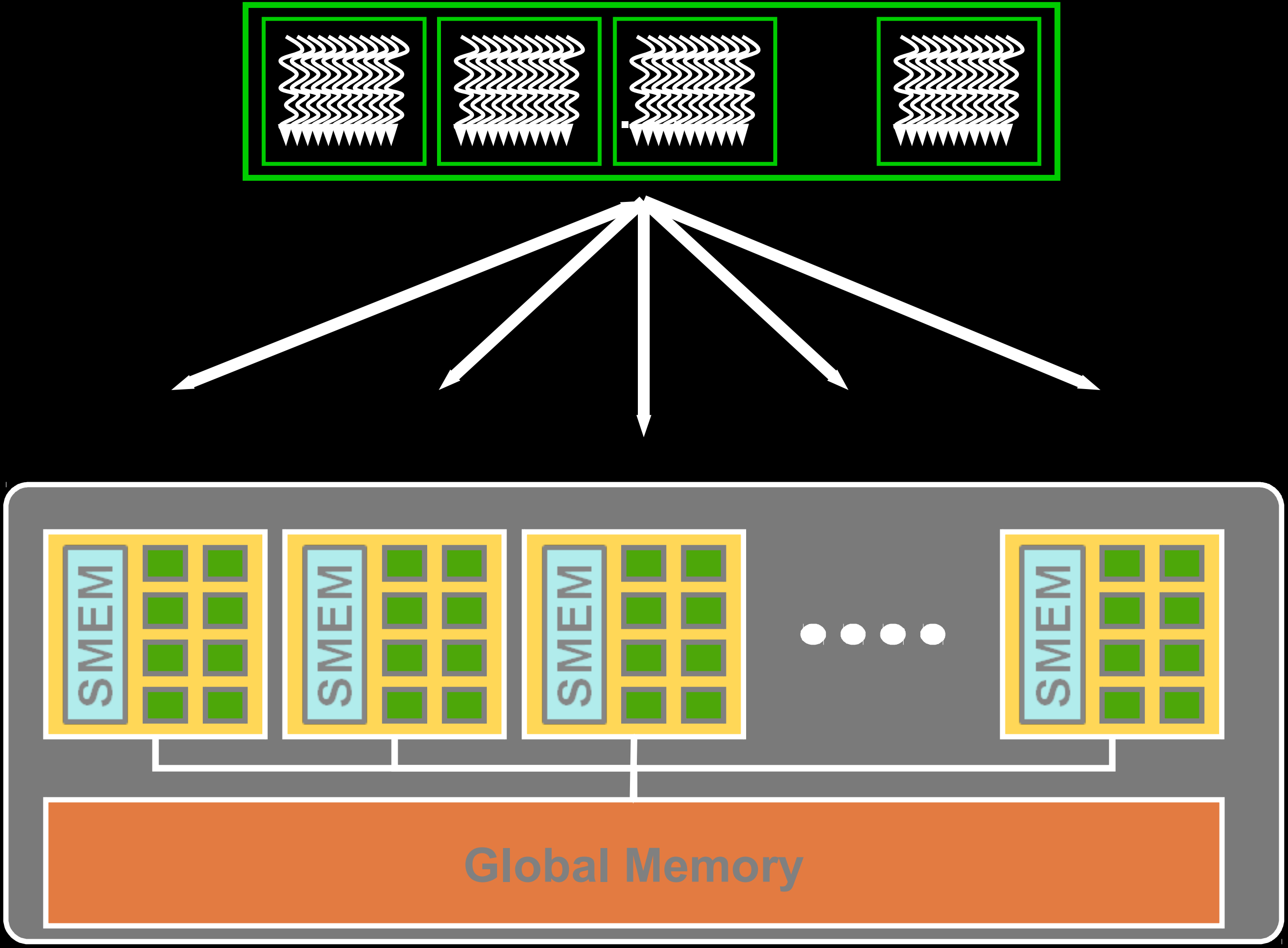


Blocks of threads run on an SM



Whole grid runs on GPU

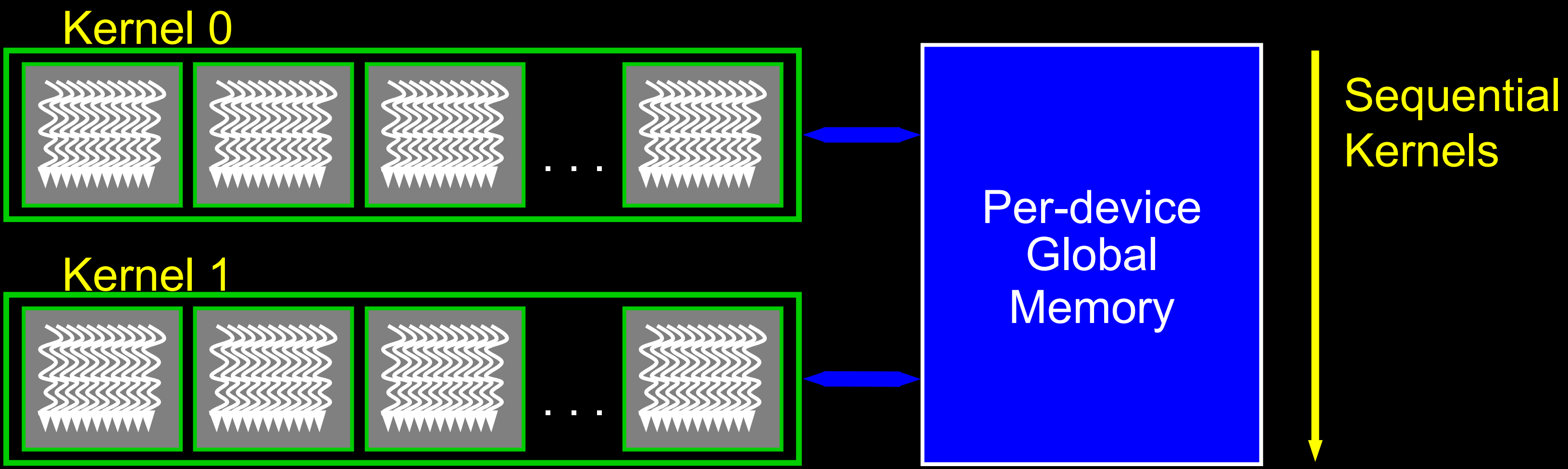
Many blocks of threads



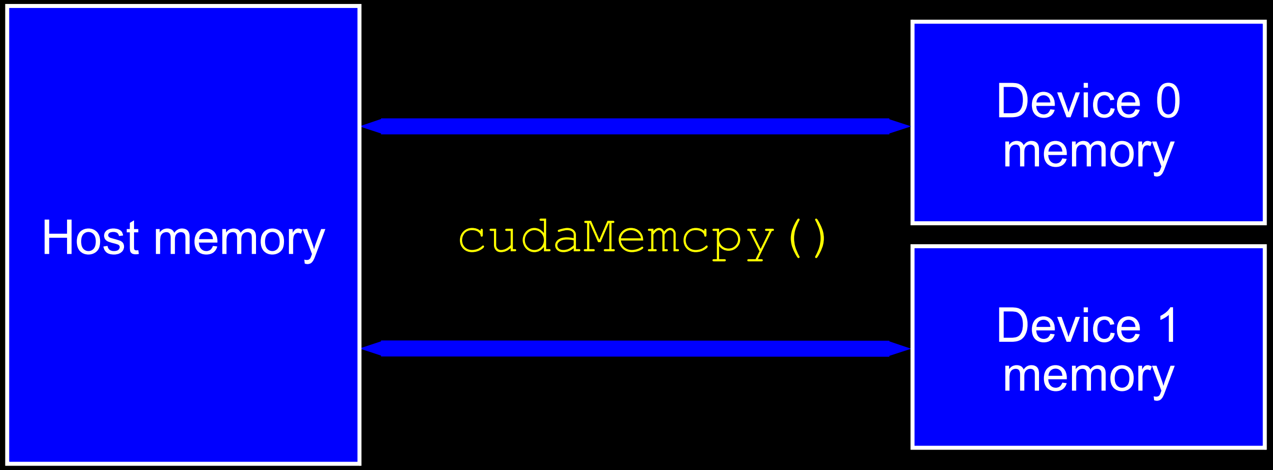
Thread Hierarchy

- Threads launched for a parallel section are partitioned into thread blocks
- Grid = all blocks for a given launch
- Thread block is a group of threads that can:
- Synchronize their execution
- Communicate via shared memory

Memory Model



Memory Model



Example: Vector Addition Kernel

Device Code

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

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

Example: Vector Addition Kernel

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

Host Code

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

Example: Host code for vecAdd

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

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

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

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

Example: Host code for vecAdd (2)

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

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

// do something with the result…

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

Kernel Variations and Output

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}
```

```
Output: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
```

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = blockIdx.x;
}
```

```
Output: 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
```

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = threadIdx.x;
}
```

```
Output: 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
```

Code executed on GPU

- C/C++ with some restrictions:
- Can only access GPU memory (well...)
- No variable number of arguments
- No static variables
- No dynamic polymorphism

- Must be declared with a qualifier:
- __global__ : launched by CPU,
  cannot be called from GPU must return void
- __device__ : called from other GPU functions,
  cannot be called by the CPU
- __host__ : can be called by CPU
- __host__ and __device__ qualifiers can be combined
- sample use: overloading operators

Memory Spaces

- CPU and GPU have separate memory spaces
- Data is moved across PCIe bus
- Use functions to allocate/set/copy memory on GPU
- Very similar to corresponding C functions

- Pointers are just addresses
- Can’t tell from the pointer value whether the address is on CPU or GPU
- Must exercise care when dereferencing:
- Dereferencing CPU pointer on GPU will likely crash
– Except on really new architectures and versions of CUDA (unified address space)
- Same for vice versa

GPU Memory Allocation / Release

- Host (CPU) manages device (GPU) memory:
- cudaMalloc (void ** pointer, size_t nbytes)
- cudaMemset (void * pointer, int value, size_t count)
- cudaFree (void* pointer)

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int * d_a = 0;
cudaMalloc( (void**)&d_a,  nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```

Data Copies

- cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);
- returns after the copy is complete
- blocks CPU thread until all bytes have been copied
- doesn’t start copying until previous CUDA calls complete
- enum cudaMemcpyKind
- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice
- Non-blocking copies are also available

Dummy Code Walkthrough 1

```
// walkthrough1.cu
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers
```

Dummy Code Walkthrough 1

```
// walkthrough1.cu
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( h_a == NULL || d_a == NULL )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }
```

Dummy Code Walkthrough 1

```
// walkthrough1.cu
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=NULL, *h_a=NULL   // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if(h_a == NULL || d_a == NULL )
    {
        printf("couldn't allocate memory\n");
        return 1;
    }

    cudaMemset( d_a, 0, num_bytes );
    // do something on the device to fill the buffer
    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );
```

Dummy Code Walkthrough 1

```
// walkthrough1.cu
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a ) {
        printf("couldn't allocate memory\n");
        return 1;
    }

    cudaMemset( d_a, 0, num_bytes );

    //do something on the device to fill the buffer

    cudaMemcpy( h_a, d_a, num_bytes,
cudaMemcpyDeviceToHost );

    for(int i=0; i<dimx; i++)
        printf("%d ", h_a[i] );
    printf("\n");

    free( h_a );
    cudaFree( d_a );

    return 0;
}
```

Example: Shuffling Data

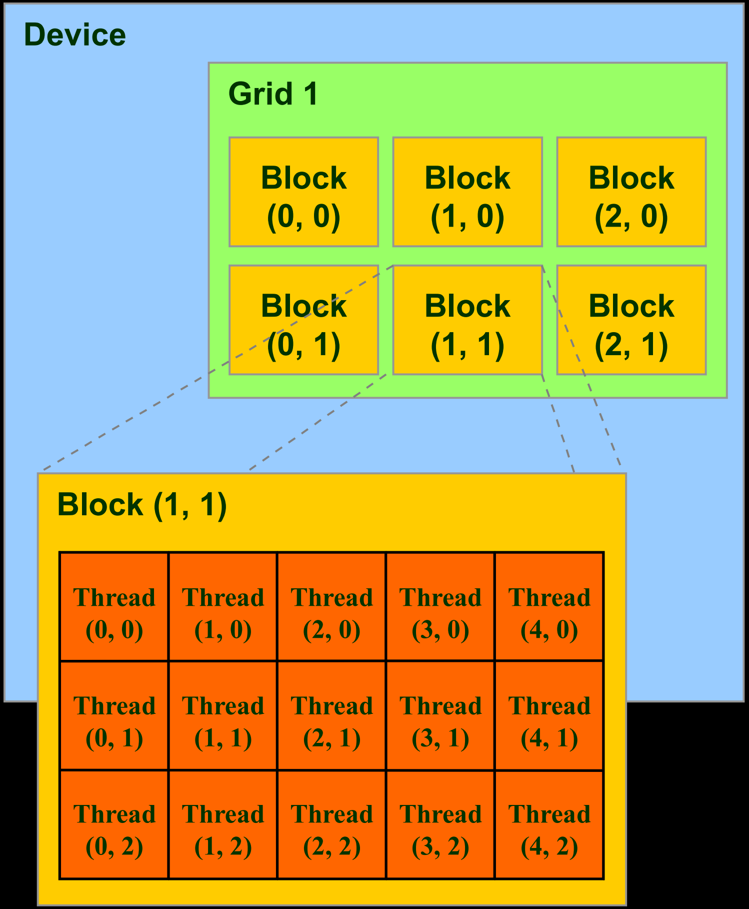
```
// Reorder values based on keys
// Each thread moves one element
__global__ void shuffle(int* prev_array, int*
  new_array, int* indices)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    new_array[i] = prev_array[indices[i]];
}
```

Host Code

```
int main()
{
    // Run grid of N/256 blocks of 256 threads
  each
    shuffle<<< N/256, 256>>>(d_old, d_new, d_ind);
}
```

IDs and Dimensions

- Threads:
  - 3D IDs, unique within a block
- Blocks:
  - 2D IDs, unique within a grid
- Dimensions set at launch
  - Can be unique for each grid
- Built-in variables:
  - threadIdx, blockIdx
  - blockDim, gridDim



Kernel with 2D Indexing

```
__global__ void kernel( int *a, int dimx, int dimy )
{
    int ix   = blockIdx.x*blockDim.x + threadIdx.x;
    int iy   = blockIdx.y*blockDim.y + threadIdx.y;
    Int dimx = blockDim.x * gridDim.x;
    int idx = iy*dimx + ix;

    a[idx]  = a[idx]+1;
}
```

```
__global__ void kernel( int *a, int dimx, int dimy )
{
    int ix   = blockIdx.x*blockDim.x + threadIdx.x;
    int iy   = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;

    a[idx]  = a[idx]+1;
}
```

```
int main() {
    int dimx = 16;
    int dimy = 16;
    int num_bytes = dimx*dimy*sizeof(int);
     int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a ) {
        printf("couldn't allocate memory\n");
        return 1;
    }

    cudaMemset( d_a, 0, num_bytes );

    dim3 grid, block;
    block.x = 4;
    block.y = 4;
    grid.x  = dimx / block.x;
    grid.y  = dimy / block.y;

    kernel<<<grid, block>>>( d_a, dimx, dimy );

    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );

    for(int row=0; row<dimy; row++) {
        for(int col=0; col<dimx; col++)
            printf("%d ", h_a[row*dimx+col] );
        printf("\n");
    }

    free( h_a );
    cudaFree( d_a );

    return 0;
}
```

Blocks must be independent

- Any possible interleaving of blocks should be valid
- presumed to run to completion without pre-emption
- can run in any order
- can run concurrently OR sequentially

- Blocks may coordinate but not synchronize
- shared queue pointer: OK
- shared lock: BAD … can easily deadlock
-

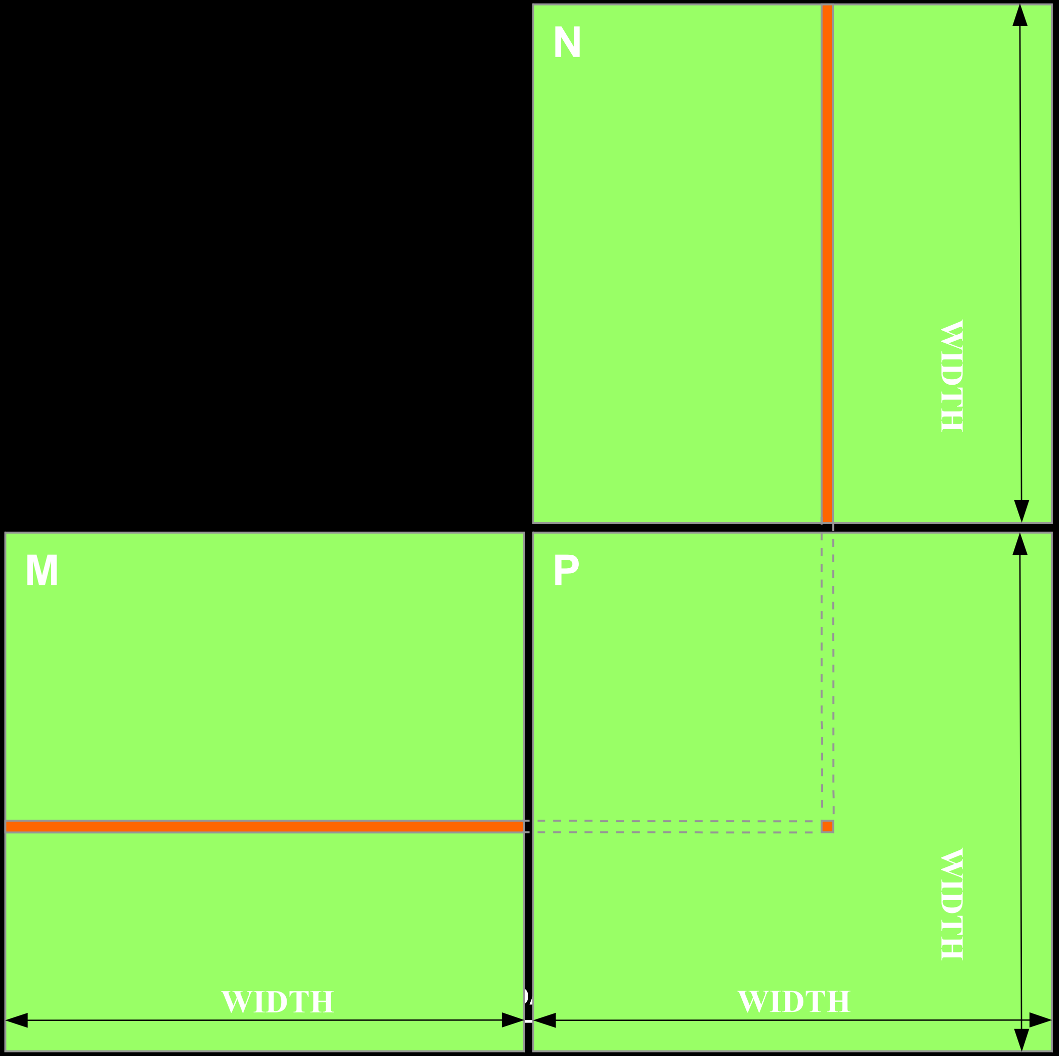
- Independence requirement gives scalability

A Simple Running Example
Matrix Multiplication

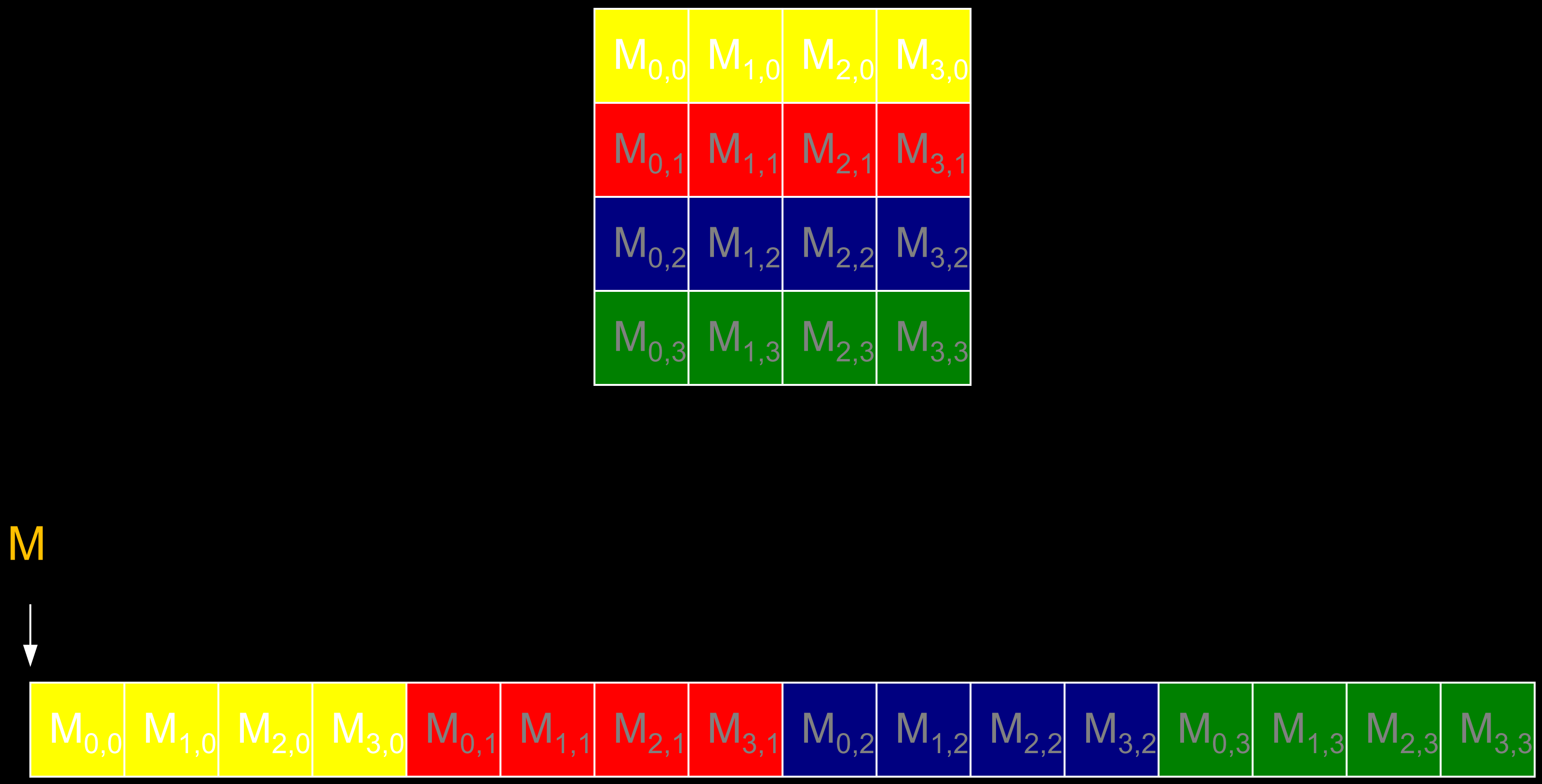
- A simple matrix multiplication example that illustrates the basic features of memory and thread management in CUDA programs
- Leave shared memory usage until later
- Local, register usage
- Thread ID usage
- Memory data transfer API between host and device
- Assume square matrix for simplicity

Programming Model: Square Matrix Multiplication Example

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



Memory Layout of a Matrix in C



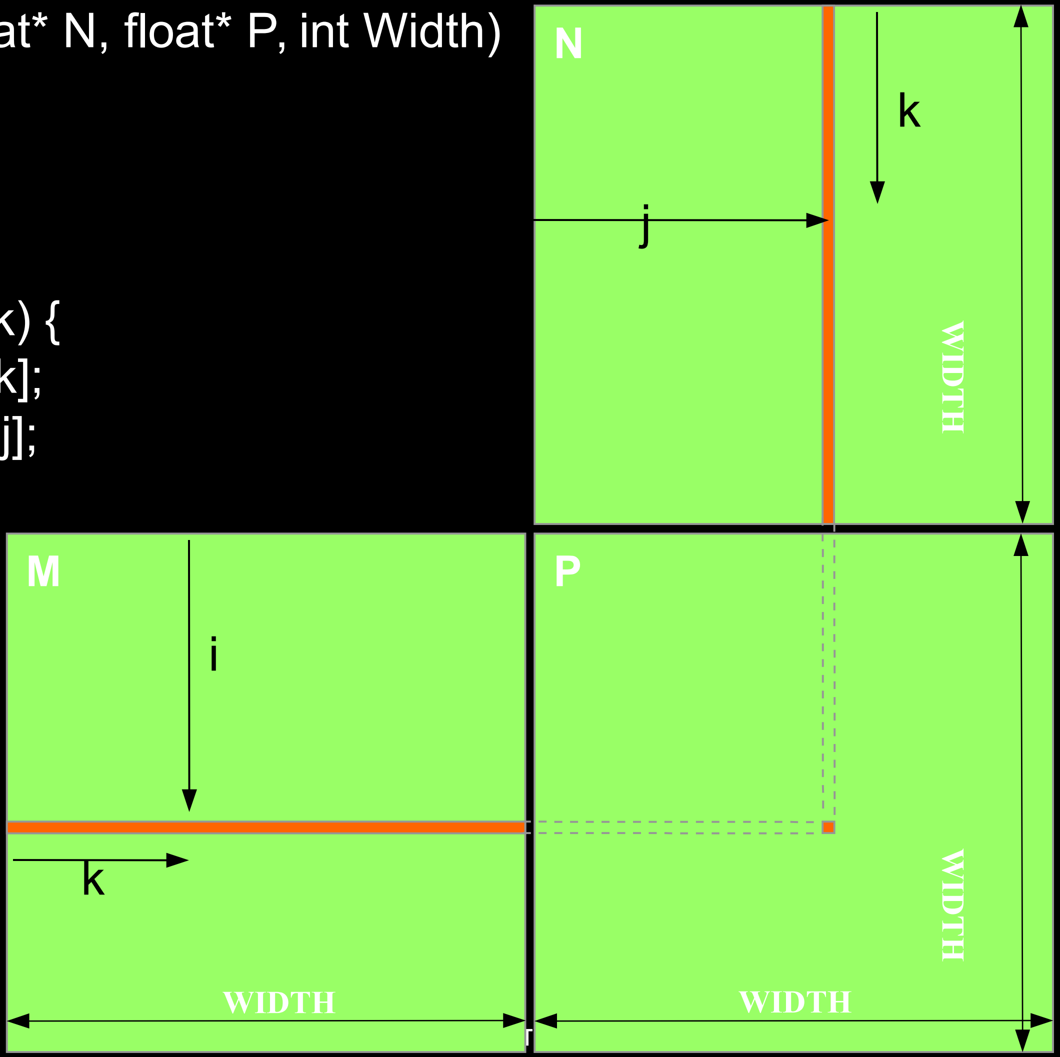
$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$
$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$
$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$
$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$

M

$M_{0,0}$	$M_{1,0}$	$M_{2,0}$	$M_{3,0}$	$M_{0,1}$	$M_{1,1}$	$M_{2,1}$	$M_{3,1}$	$M_{0,2}$	$M_{1,2}$	$M_{2,2}$	$M_{3,2}$	$M_{0,3}$	$M_{1,3}$	$M_{2,3}$	$M_{3,3}$

Step 1: Matrix Multiplication
A Simple Host Version in C

```
// Matrix multiplication on the (CPU) host in double precision
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            double sum = 0;
            for (int k = 0; k < Width; ++k) {
                double a = M[i * width + k];
                double b = N[k * width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```



Step 2: Input Matrix Data Transfer (Host-side Code)

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
  int size = Width * Width * sizeof(float);
   float* Md, Nd, Pd;
  …
   // 1. Allocate and Load M, N to device memory
   cudaMalloc(&Md, size);
   cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

    cudaMalloc(&Nd, size);
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

    // Allocate P on the device
   cudaMalloc(&Pd, size);
```

Step 3: Output Matrix Data Transfer (Host-side Code)

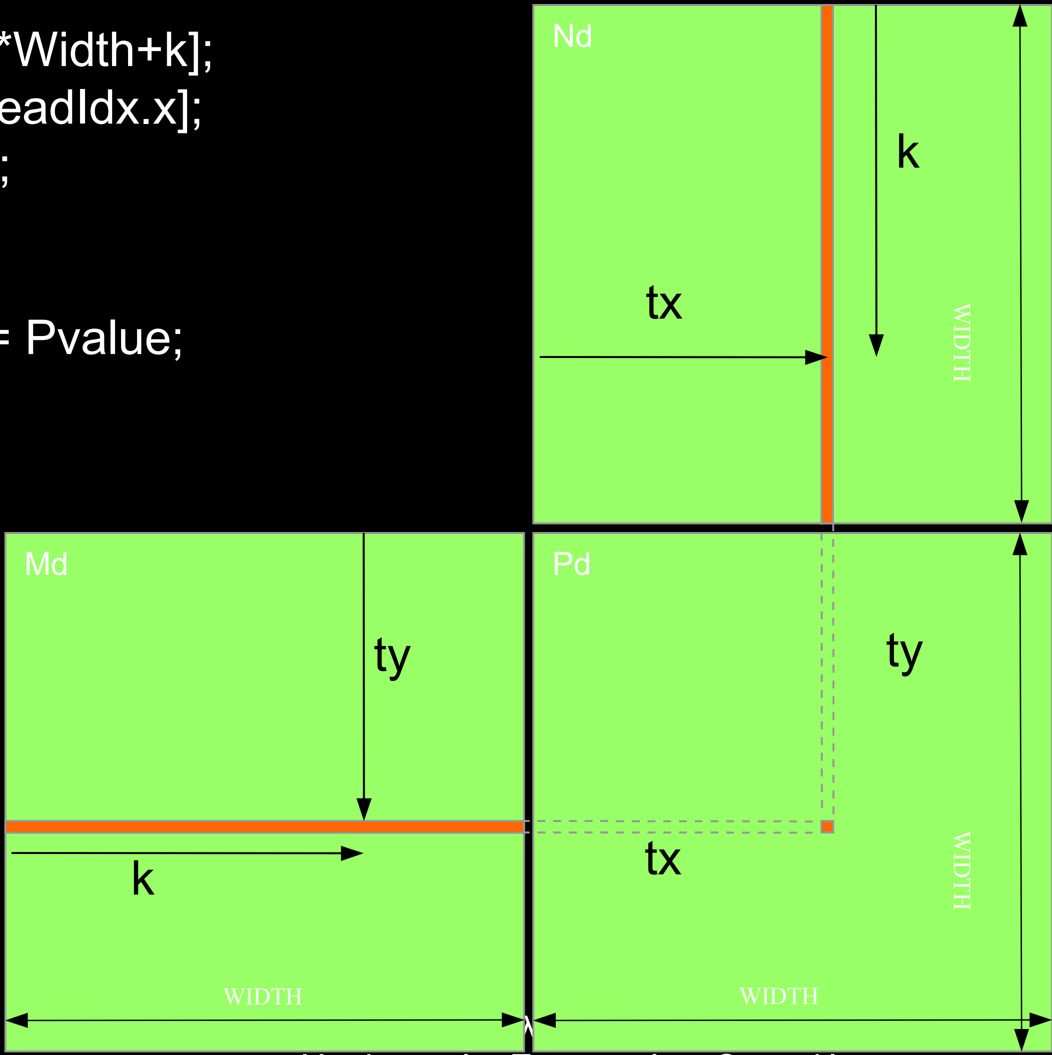
```
2.  // Kernel invocation code – to be shown later
   …

3.   // Read P from the device
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

     // Free device matrices
    cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
   }
```

Step 4: Kernel Function

```
// Matrix multiplication kernel – per thread code

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

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
```

Step 4: Kernel Function (cont.)

```
    for (int k = 0; k < Width; ++k) {
        float Melement = Md[threadIdx.y*Width+k];
        float Nelement = Nd[k*Width+threadIdx.x];
        Pvalue += Melement * Nelement;
    }

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



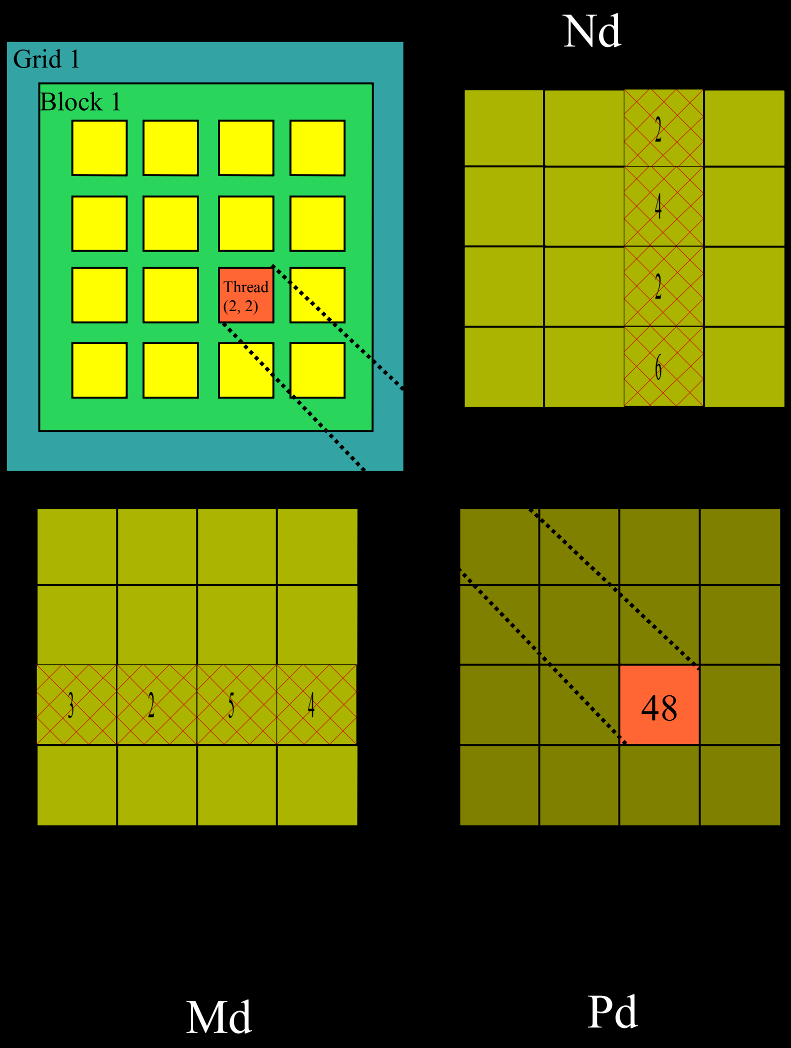
Step 5: Kernel Invocation (Host-side Code)

```
// Setup the execution configuration
    dim3 dimGrid(1, 1);
   dim3 dimBlock(Width, Width);
```

```
// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

Only One Thread Block Used

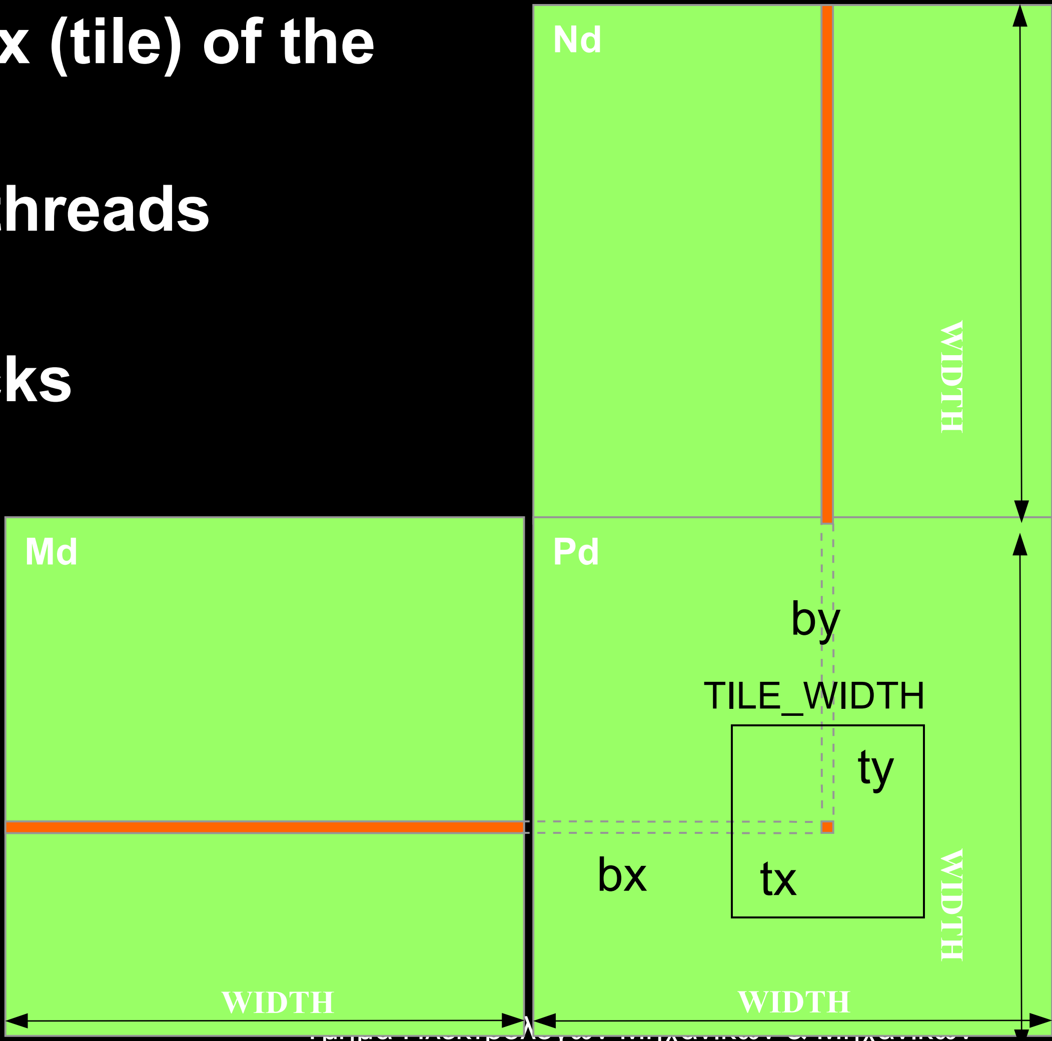
- One Block of threads compute matrix Pd
  - Each thread computes one element of Pd
- Each thread
  - Loads a row of matrix Md
  - Loads a column of matrix Nd
  - Perform one multiply and addition for each pair of Md and Nd elements
  - Compute to off-chip memory access ratio close to 1:1 (not very high)
- Size of matrix limited by the number of threads allowed in a thread block



Step 7: Handling Arbitrary Sized Square Matrices (will cover later)

- Have each 2D thread block to compute a $(\text{TILE\_WIDTH})^2$ sub-matrix (tile) of the result matrix
- Each has $(\text{TILE\_WIDTH})^2$ threads
- Generate a 2D Grid of $(\text{WIDTH/TILE\_WIDTH})^2$ blocks

You still need to put a loop around the kernel call for cases where WIDTH/TILE_WIDTH is greater than max grid size (64K)!



USEFUL INFORMATION ON TOOLS

NVCC Compiler

- NVIDIA provides a CUDA-C compiler
  - nvcc
- NVCC compiles device code then forwards code on to the host compiler (e.g. g++)
- Can be used to compile & link host only applications

Example 1: Hello World

```
int main() {
    printf("Hello World!\n");
    return 0;
}
```

Instructions:

1. Build and run the hello world code
2. Modify Makefile to use nvcc instead of g++
3. Rebuild and run

CUDA Example 1: Hello World

```
__global__ void mykernel(void) {
}

int main(void) {
   mykernel<<<1,1>>>();
   printf("Hello World!\n");
   return 0;
}
```

Instructions:

1. Add kernel and kernel launch to main.cu
2. Try to build

CUDA Example 1: Build Considerations

- **Build failed**
  - Nvcc only parses .cu files for CUDA
- **Fixes:**
  - Rename main.cc to main.cu

  OR
  - nvcc –x cu
    - Treat all input files as .cu files

Instructions:

1. Rename main.cc to main.cu
2. Rebuild and Run

Hello World! with Device Code

```
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

Output:

```
$ nvcc main.cu
$ ./a.out
Hello World!
```

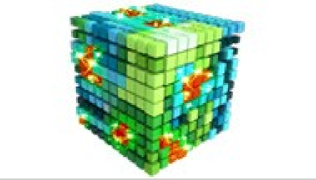
- `mykernel` **(does nothing, somewhat anticlimactic!)**

Developer Tools - Debuggers

NSIGHT

CUDA-GDB

CUDA MEMCHECK



NVIDIA Provided

allinea DDT

TotalView®

3rd Party

https://developer.nvidia.com/debugging-solutions

Compiler Flags

- **Remember there are two compilers being used**
  - NVCC: Device code
  - Host Compiler: C/C++ code
- **NVCC supports some host compiler flags**
  - If flag is unsupported, use –Xcompiler to forward to host
    - e.g. –Xcompiler –fopenmp
- **Debugging Flags**
  - -g: Include host debugging symbols
  - -G: Include device debugging symbols
  - -lineinfo: Include line information

CUDA-MEMCHECK

- **Memory debugging tool**
  - No recompilation necessary

```
%> cuda-memcheck ./exe
```

- **Can detect the following errors**
  - Memory leaks
  - Memory errors (OOB, misaligned access, illegal instruction, etc)
  - Race conditions
  - Illegal Barriers
  - Uninitialized Memory
- **For line numbers use the following compiler flags:**
  - -Xcompiler -rdynamic -lineinfo

http://docs.nvidia.com/cuda/cuda-memcheck

Example 2: CUDA-MEMCHECK

Instructions:

1. Build & Run Example 2

   Output should be the numbers 0-9

   Do you get the correct results?

2. Run with cuda-memcheck

   ```
   %> cuda-memcheck ./a.out
   ```

3. Add nvcc flags "–Xcompiler –rdynamic –lineinfo"
4. Rebuild & Run with cuda-memcheck
5. Fix the illegal write

http://docs.nvidia.com/cuda/cuda-memcheck

CUDA-GDB

- **cuda-gdb is an extension of GDB**
  - Provides seamless debugging of CUDA and CPU code
- **Works on Linux and Macintosh**
  - For a Windows debugger use NSIGHT Visual Studio Edition

http://docs.nvidia.com/cuda/cuda-gdb

Example 3: cuda-gdb

Instructions:

1. Run exercise 3 in cuda-gdb

   %> cuda-gdb --args ./a.out

2. Run a few cuda-gdb commands:

```
(cuda-gdb) b main                    //set break point at main
(cuda-gdb) r                        //run application
(cuda-gdb) l                        //print line context
(cuda-gdb) b foo                         //break at kernel foo
(cuda-gdb) c                        //continue
(cuda-gdb) cuda thread               //print current thread
(cuda-gdb) cuda thread 10 //switch to thread 10
(cuda-gdb) cuda block                //print current block
(cuda-gdb) cuda block 1              //switch to block 1
(cuda-gdb) d                        //delete all break points
(cuda-gdb) set cuda memcheck on      //turn on cuda
memcheck
(cuda-gdb) r                        //run from the beginning
```

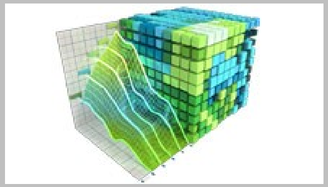
3. Fix Bug

http://docs.nvidia.com/cuda/cuda-gdb

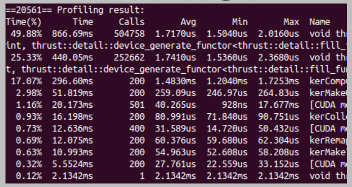
Developer Tools - Profilers

NSIGHT

NVVP



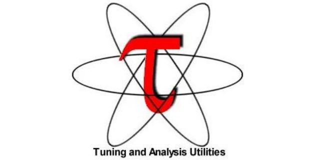
NVPROF



```
==20561== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 49.88%  866.69ms    504758  1.7170us  1.5040us  2.0160us  void th
int, thrust::detail::device_generate_functor<thrust::detail::fill_
 25.33%  440.05ms    252662  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  1.2040ms  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  kerRema
  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
```

NVIDIA Provided

TAU



VampirTrace

3rd Party

https://developer.nvidia.com/performance-analysis-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

Example 4: nvprof

Instructions:

1. Collect profile information for the matrix add example

   ```
   %> nvprof ./a.out
   ```

2. How much faster is add_v2 than add_v1?
3. View available metrics

   ```
   %> nvprof --query-metrics
   ```

4. View global load/store efficiency

   ```
   %> nvprof --metrics gld_efficiency,gst_efficiency ./a.out
   ```

5. Store a timeline to load in NVVP

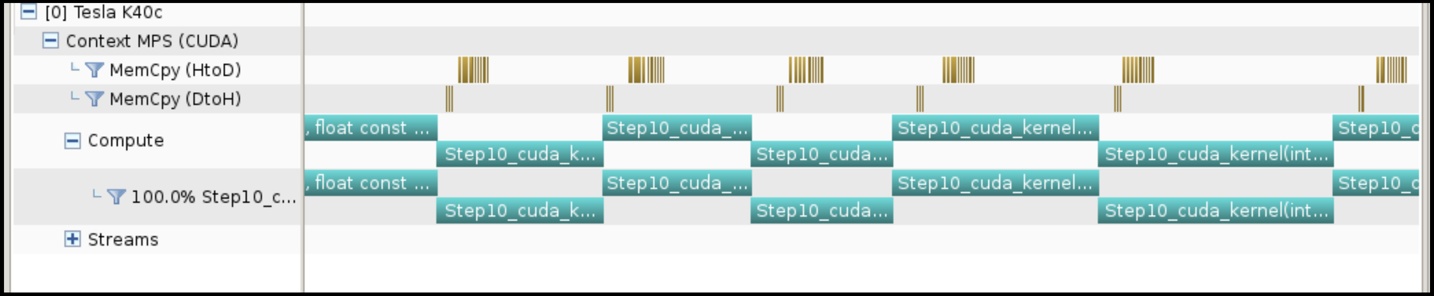
   ```
   %> nvprof –o profile.timeline ./a.out
   ```

6. Store analysis metrics to load in NVVP

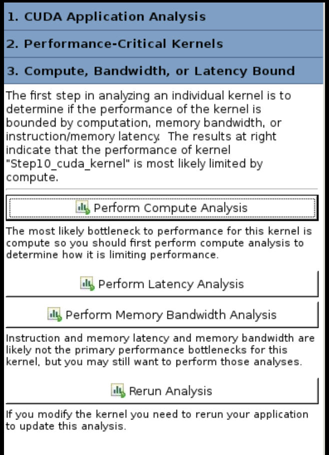
   ```
   %> nvprof –o profile.metrics --analysis-metrics ./a.out
   ```

NVIDIA’s Visual Profiler (NVVP)

Timeline



Guided System



1. CUDA Application Analysis

2. Performance-Critical Kernels

3. Compute, Bandwidth, or Latency Bound

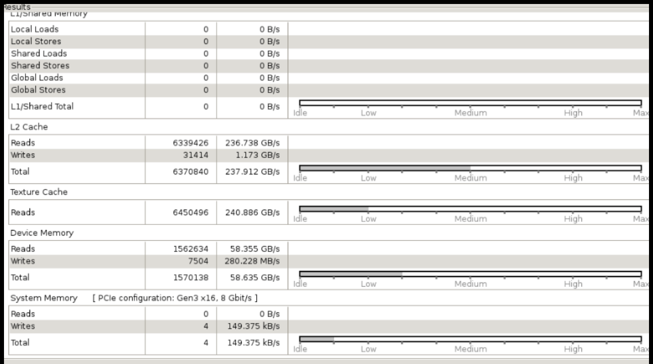
The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "Step10_cuda_kernel" is most likely limited by compute.

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.

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.

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

Analysis



Results

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

L2 Cache

Reads	6339426	236.738 GB/s
Writes	31414	1.173 GB/s
Total	6370840	237.912 GB/s

Texture Cache

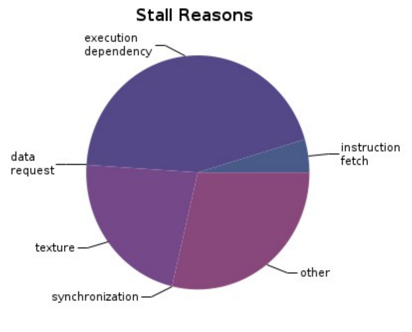
Reads	6450496	240.886 GB/s

Device Memory

Reads	1562634	58.355 GB/s
Writes	7504	280.228 MB/s
Total	1570138	58.635 GB/s

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



Example 4: NVVP

Instructions:

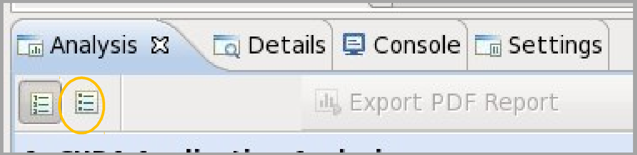
1. Import nvprof profile into NVVP
    - Launch nvvp
    - Click File/ Import/ Nvprof/ Next/ Single process/ Next / Browse
        - Select profile.timeline
    - Add Metrics to timeline
        - Click on 2nd Browse
        - Select profile.metrics
    - Click Finish
2. Explore Timeline
    - Control + mouse drag in timeline to zoom in
    - Control + mouse drag in measure bar (on top) to measure time

Example 4: NVVP

Instructions:

1. Click on a kernel
2. On Analysis tab click on the unguided analysis
3.

1.



2. Click Analyze All

   Explore metrics and properties

   What differences do you see between the two kernels?

Note:

If kernel order is non-deterministic you can only load the timeline or the metrics but not both.

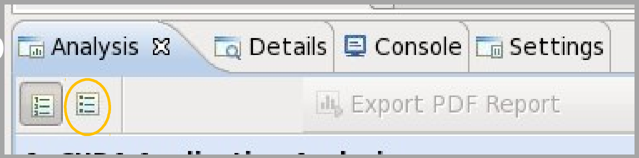
If you load just metrics the timeline looks odd but metrics are correct.

Example 4: NVVP

Let’s now generate the same data within NVVP

Instructions:

1. Click File / New Session / Browse
   Select Example 4/a.o
   Click Next / Finish



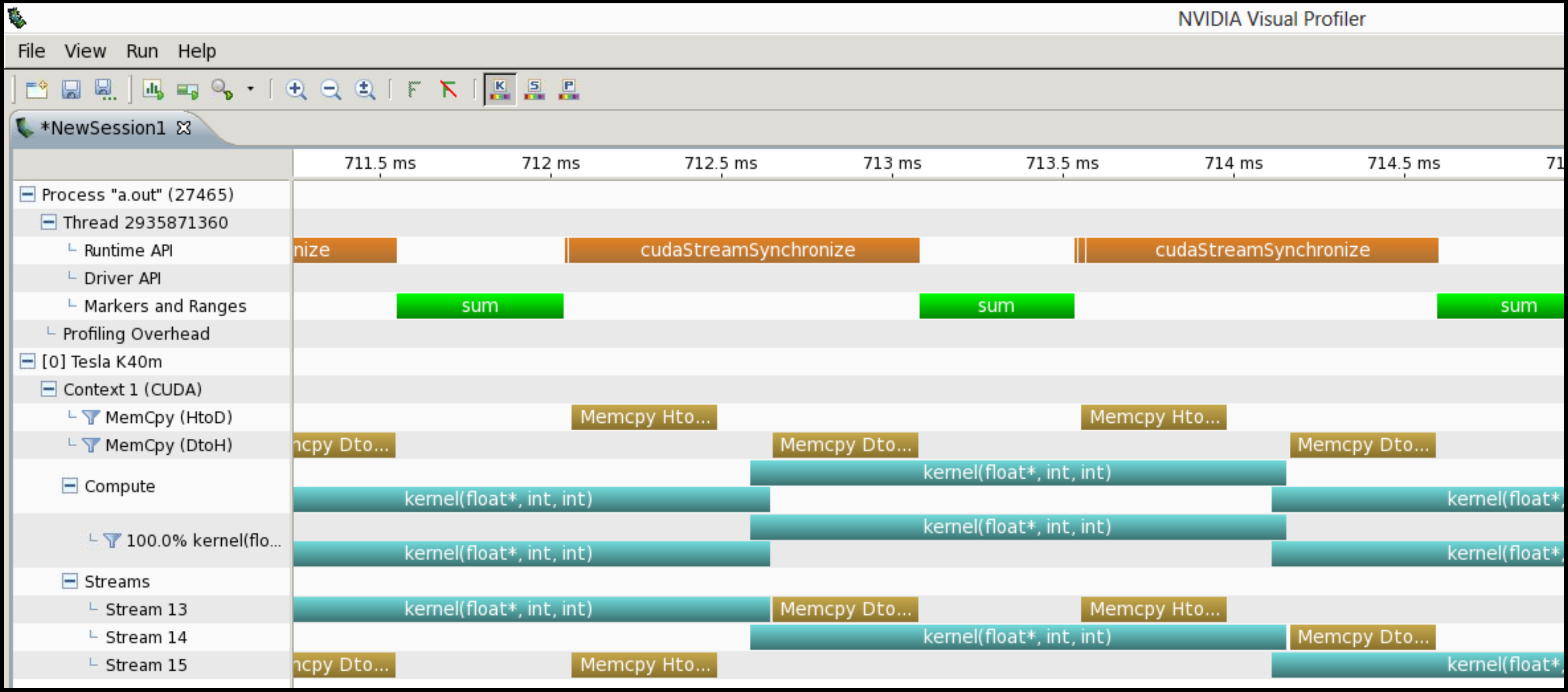
2. Click on a kernel
   Select Unguided Analysis
   Click Analyze All

NVTX

- Our current tools only profile API calls on the host
  - What if we want to understand better what the host is doing?
- The NVTX library allows us to annotate profiles with ranges
  - Add: #include <nvToolsExt.h>
  - Link with: -lnvToolsExt
- Mark the start of a range
  - nvtxRangePushA(“description”);
- Mark the end of a range
  - nvtxRangePop();
- Ranges are allowed to overlap

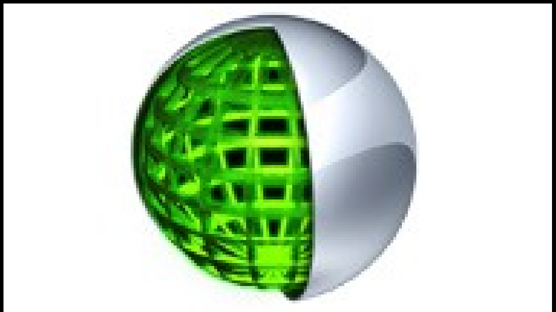
http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-generate-custom-application-profile-timelines-nvtx/

NVTX Profile



NSIGHT

- **CUDA enabled Integrated Development Environment**
  - Source code editor: syntax highlighting, code refactoring, etc
  - Build Manger
  - Visual Debugger
  - Visual Profiler
- **Linux/Macintosh**
  - Editor = Eclipse
  - Debugger = cuda-gdb with a visual wrapper
  - Profiler = NVVP
- **Windows**
  - Integrates directly into Visual Studio
  - Profiler is NSIGHT VSE



Example 4: NSIGHT

Let’s import an existing Makefile project into NSIGHT

–

–

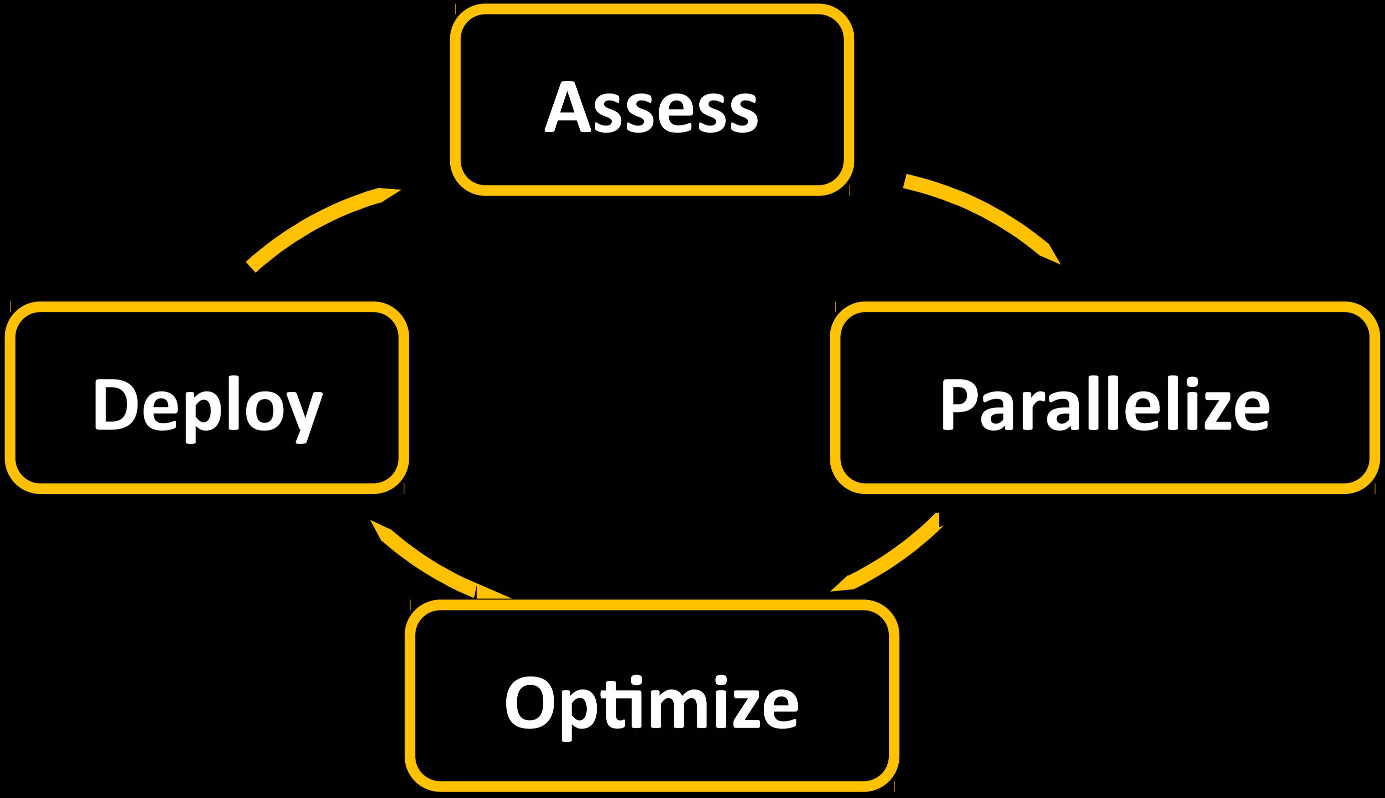
Instructions:

1. Run nsight
   Select default workspace
2. Click File / New / Makefile Project With Existing CodeTest
3. Enter Project Name and select the Example15 directory
4. Click Finish
5. Right Click On Project / Properties / Run Settings / New / C++ Application
6. Browse for Example 4/a.out
7. In Project Explorer double click on main.cu and explore source
8. Click on the build icon
9. Click on the run icon
10. Click on the profile icon

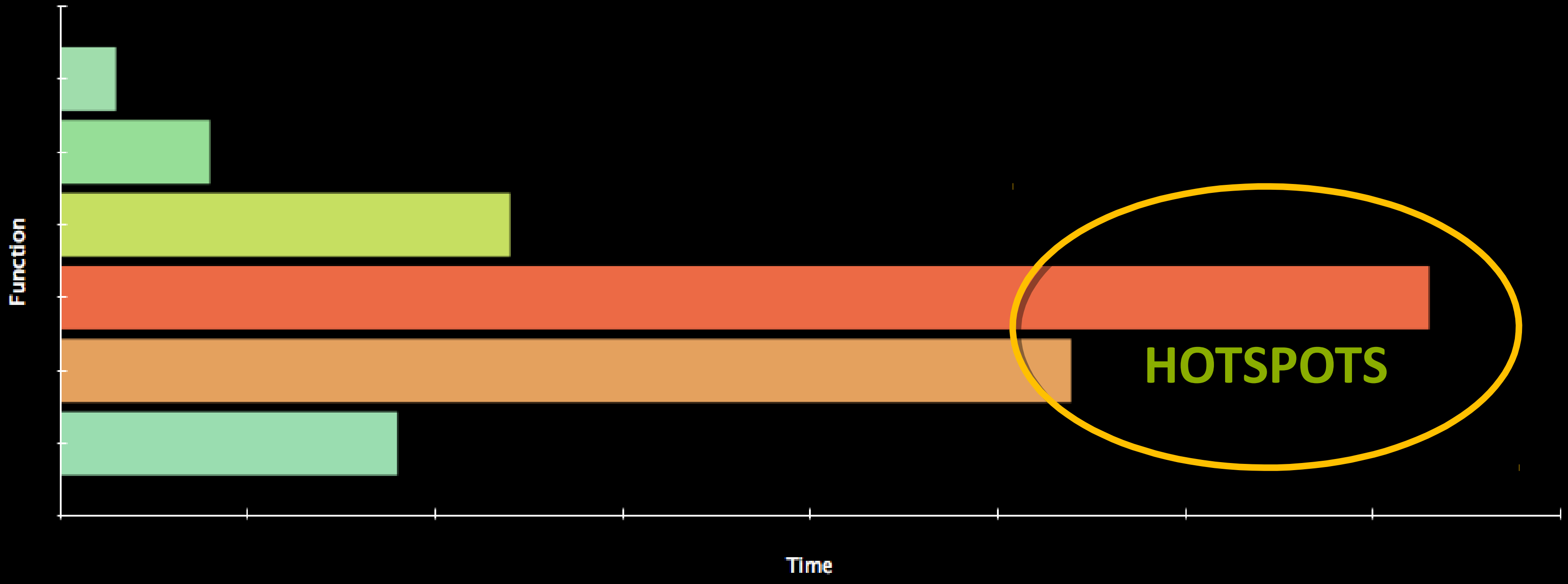
Profiler Summary

- **Many profile tools are available**
- **NVIDIA Provided**
  - NVPROF: Command Line
  - NVVP: Visual profiler
  - NSIGHT: IDE (Visual Studio and Eclipse)
- **3rd Party**
  - TAU
  - VAMPIR

Optimization

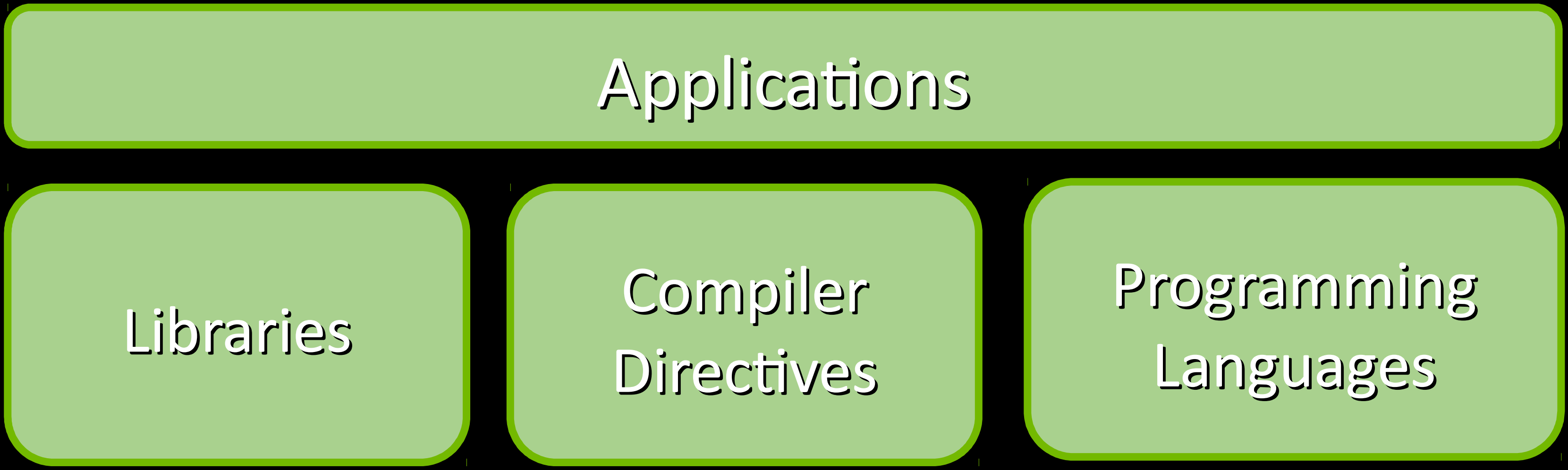


Assess



- Profile the code, find the hotspot(s)
- Focus your attention where it will give the most benefit

Parallelize



Applications

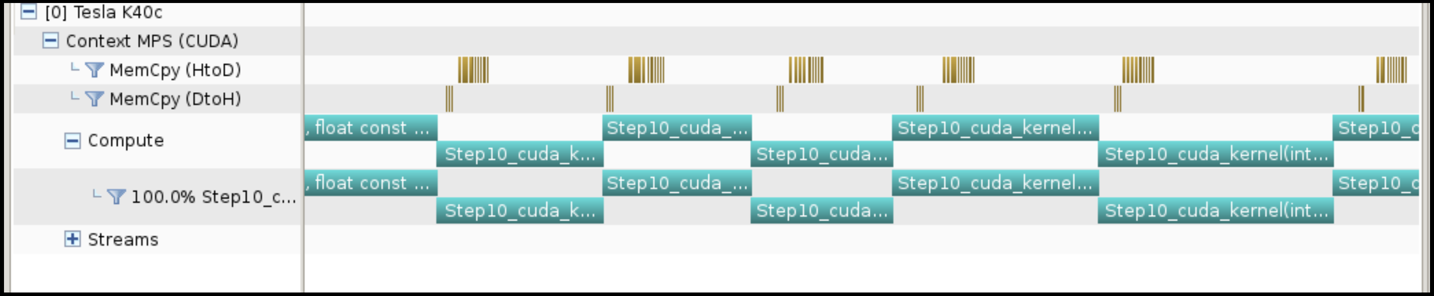
Libraries

Compiler Directives

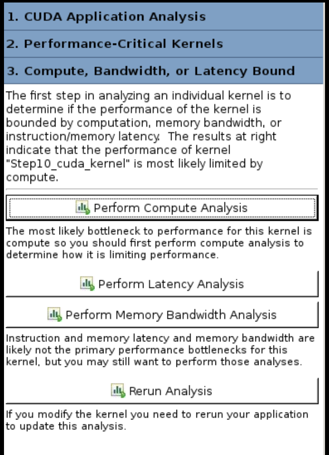
Programming Languages

Optimize

Timeline



Guided System



1. CUDA Application Analysis

2. Performance-Critical Kernels

3. Compute, Bandwidth, or Latency Bound

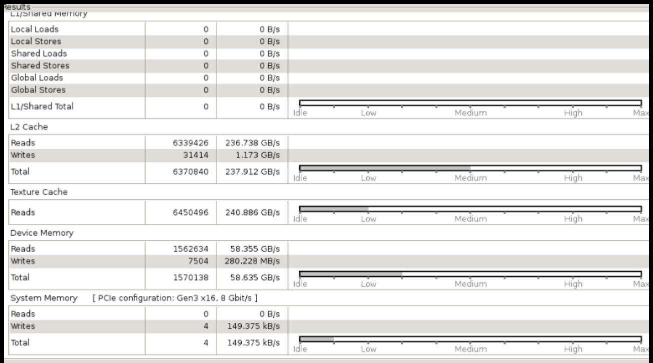
The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "Step10_cuda_kernel" is most likely limited by compute.

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.

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.

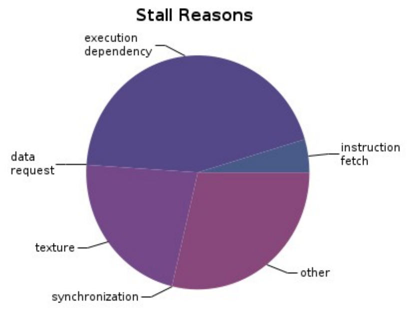
If you modify the kernel you need to rerun your application to update this analysis.

Analysis



Results

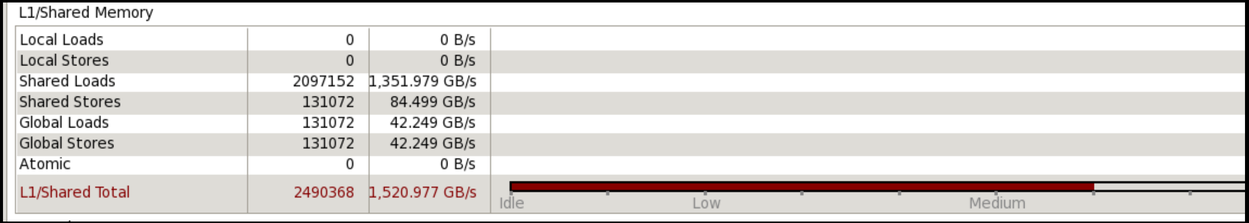
L1/Shared Memory		
Local Loads	0	0 B/s
Local Stores	0	0 B/s
Shared Loads	0	0 B/s
Shared Stores	0	0 B/s
Global Loads	0	0 B/s
Global Stores	0	0 B/s
L1/Shared Total	0	0 B/s
L2 Cache		
Reads	6339426	236.738 GB/s
Writes	31414	1.173 GB/s
Total	6370840	237.912 GB/s
Texture Cache		
Reads	6450496	240.886 GB/s
Device Memory		
Reads	1562634	58.355 GB/s
Writes	7504	280.228 MB/s
Total	1570138	58.635 GB/s
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



Bottleneck Analysis

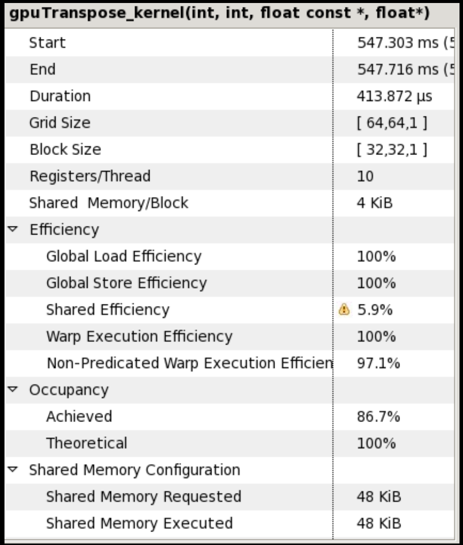
- Don’t assume an optimization was wrong
- Verify if it was wrong with the profiler

129 GB/s → 84 GB/s



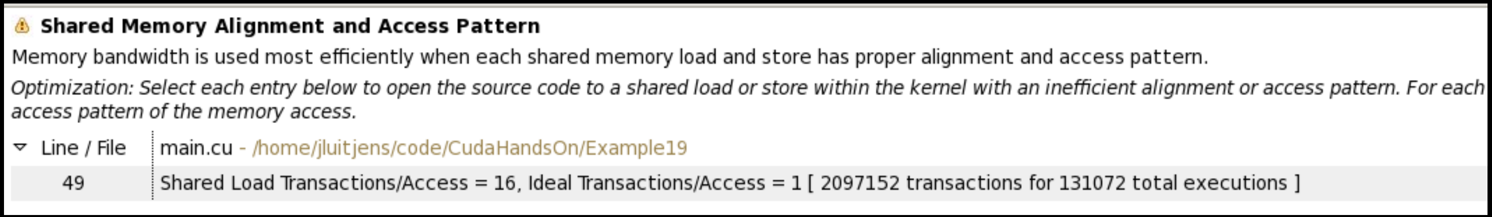
L1/Shared Memory

Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	2097152	1,351.979 GB/s	
Shared Stores	131072	84.499 GB/s	
Global Loads	131072	42.249 GB/s	
Global Stores	131072	42.249 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	2490368	1,520.977 GB/s	Idle Low Medium



gpuTranspose_kernel(int, int, float const *, float*)

Start	547.303 ms (5
End	547.716 ms (5
Duration	413.872 µs
Grid Size	[64,64,1]
Block Size	[32,32,1]
Registers/Thread	10
Shared Memory/Block	4 KiB
▿ Efficiency	
Global Load Efficiency	100%
Global Store Efficiency	100%
Shared Efficiency	⚠ 5.9%
Warp Execution Efficiency	100%
Non-Predicated Warp Execution Efficien	97.1%
▿ Occupancy	
Achieved	86.7%
Theoretical	100%
▿ Shared Memory Configuration	
Shared Memory Requested	48 KiB
Shared Memory Executed	48 KiB



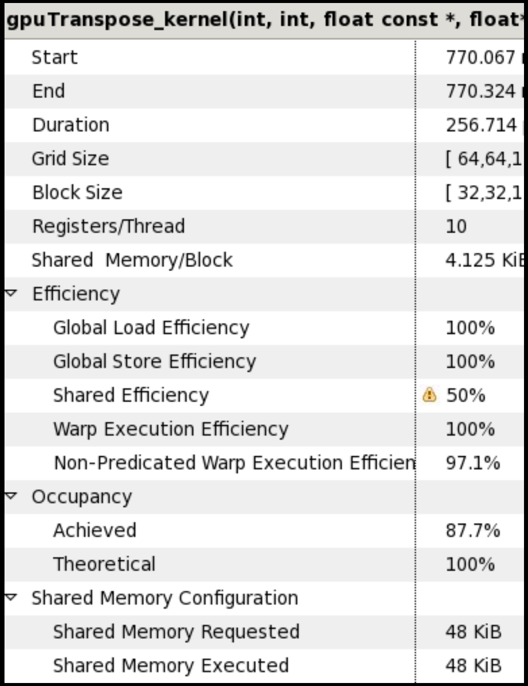
⚠ **Shared Memory Alignment and Access Pattern**

Memory bandwidth is used most efficiently when each shared memory load and store has proper alignment and access pattern.

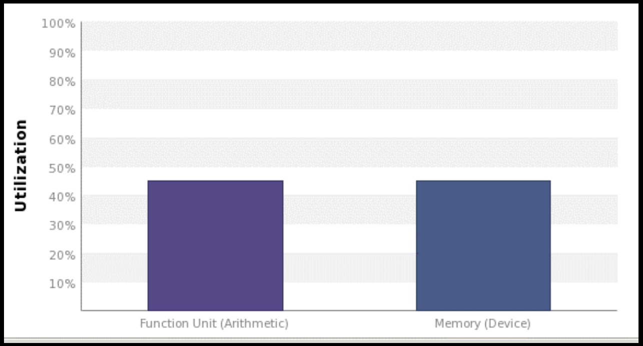
Optimization: Select each entry below to open the source code to a shared load or store within the kernel with an inefficient alignment or access pattern. For each access pattern of the memory access.

▿ Line / File	main.cu - /home/jluitjens/code/CudaHandsOn/Example19
49	Shared Load Transactions/Access = 16, Ideal Transactions/Access = 1 [2097152 transactions for 131072 total executions]

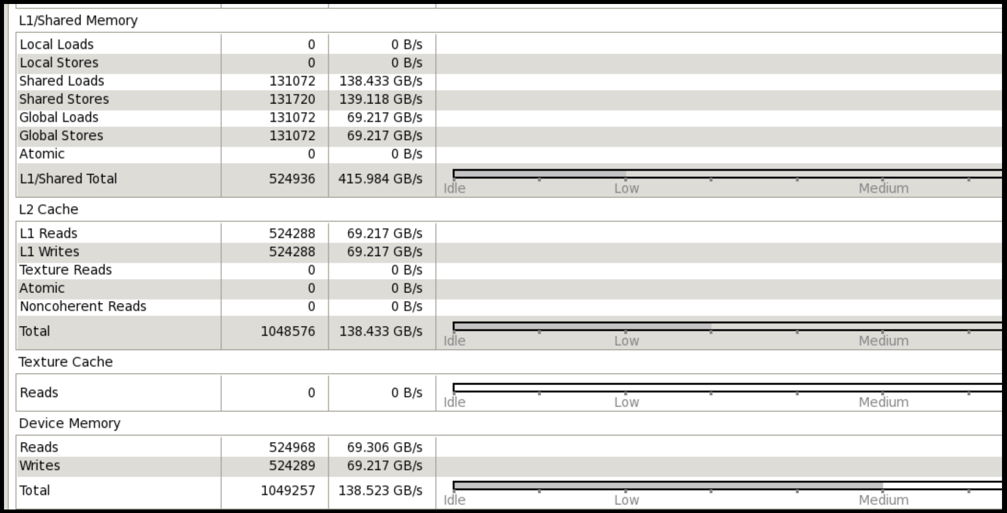
Performance Analysis



gpuTranspose_kernel(int, int, float const *, float*	
Start	770.067
End	770.324
Duration	256.714
Grid Size	[64,64,1
Block Size	[32,32,1
Registers/Thread	10
Shared Memory/Block	4.125 KiE
▿ Efficiency	
Global Load Efficiency	100%
Global Store Efficiency	100%
Shared Efficiency	⚠ 50%
Warp Execution Efficiency	100%
Non-Predicated Warp Execution Efficien	97.1%
▿ Occupancy	
Achieved	87.7%
Theoretical	100%
▿ Shared Memory Configuration	
Shared Memory Requested	48 KiB
Shared Memory Executed	48 KiB



84 GB/s → 137 GB/s



L1/Shared Memory		
Local Loads	0	0 B/s
Local Stores	0	0 B/s
Shared Loads	131072	138.433 GB/s
Shared Stores	131720	139.118 GB/s
Global Loads	131072	69.217 GB/s
Global Stores	131072	69.217 GB/s
Atomic	0	0 B/s
L1/Shared Total	524936	415.984 GB/s
L2 Cache		
L1 Reads	524288	69.217 GB/s
L1 Writes	524288	69.217 GB/s
Texture Reads	0	0 B/s
Atomic	0	0 B/s
Noncoherent Reads	0	0 B/s
Total	1048576	138.433 GB/s
Texture Cache		
Reads	0	0 B/s
Device Memory		
Reads	524968	69.306 GB/s
Writes	524289	69.217 GB/s
Total	1049257	138.523 GB/s

Floating Point

- Results of floating-point computations will slightly differ on some GPUs because of:
    - Different compiler outputs, instruction sets
    - Use of extended precision for intermediate results
- There are various options to force strict single precision on the host

ATOMICS

The Problem

- **How do you do global communication?**
- **Finish a grid and start a new one**
-
-

Global Communication

- Finish a kernel and start a new one
- All writes from all threads complete before a kernel finishes
  - If kernel invocations are synchronous.
  - This is not the case with Fermi and later!

```
step1<<<grid1,blk1>>>(...);
// The system ensures that all
      // writes from step1
complete.
step2<<<grid2,blk2>>>(...);
```

Global Communication

- Would need to decompose kernels into before and after parts

Race Conditions

- Or, write to a predefined memory location
- Race condition! Updates can be lost

Race Conditions

```
threadId:0
threadId:1917
     // vector[0] was equal to 0
vector[0] += 5;              vector[0] += 1;
...                          ...
a = vector[0];               a = vector[0];
```

- What is the value of a in thread 0?
- What is the value of a in thread 1917?

Race Conditions

- Thread 0 could have finished execution before 1917 started
- Or the other way around
- Or both are executing at the same time

Race Conditions

- Answer: not defined by the programming model, can be arbitrary

Atomics

- CUDA provides atomic operations to deal with this problem

Atomics

- An atomic operation guarantees that only a single thread has access to a piece of memory while an operation completes
- The name atomic comes from the fact that it is uninterruptable
- No dropped data, but ordering is still arbitrary
- Different types of atomic instructions
- `atomic{Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor}`
- More types in Fermi and later

Example: Histogram

```
// Determine frequency of colors in a picture
// colors have already been converted into ints
// Each thread looks at one pixel and increments
// a counter atomically
__global__ void histogram(int* color,
                          int* buckets)
{
  int i = threadIdx.x
        + blockDim.x * blockIdx.x;
  int c = colors[i];
  atomicAdd(&buckets[c], 1);
}
```

Example: Workqueue

```
// For algorithms where the amount of work per
   item
// is highly non-uniform, it often makes sense for
// to continuously grab work from a queue
__global__
void workq(int* work_q, int* q_counter,
           int* output, int queue_max)
{
  int i = threadIdx.x
        + blockDim.x * blockIdx.x;
  int q_index =
    atomicInc(q_counter, queue_max);
  int result = do_work(work_q[q_index]);
  output[i] = result;
}
```

Atomics

- Atomics are slower than normal load/store
- You can have the whole machine queuing on a single location in memory
- Atomics unavailable on older (really old now) architectures!

Atomics

- `atomicAdd` returns the previous value at a certain address
- Useful for grabbing variable amounts of data from a list

Example: Global Min/Max (Naive)

```
// If you require the maximum across all threads
// in a grid, you could do it with a single global
// maximum value, but it will be VERY slow
__global__
void global_max(int* values, int* gl_max)
{
  int i = threadIdx.x
        + blockDim.x * blockIdx.x;
  int val = values[i];
  atomicMax(gl_max,val);
}
```

Example: Global Min/Max (Better)

```
// introduce intermediate maximum results, so that
// most threads do not try to update the global
  max
__global__
void global_max(int* values, int* max,
                int *reg_max,
                int num_regions)
{
  // i and val as before …
  int region = i % num_regions;
  if(atomicMax(&reg_max[region],val) <
  val)
  {
    atomicMax(max,val);
  }
```

Global Min/Max

- Single value causes serial bottleneck
- Create hierarchy of values for more parallelism
- Performance will still be slow, so use judiciously
- See next lecture for even better version!

Summary

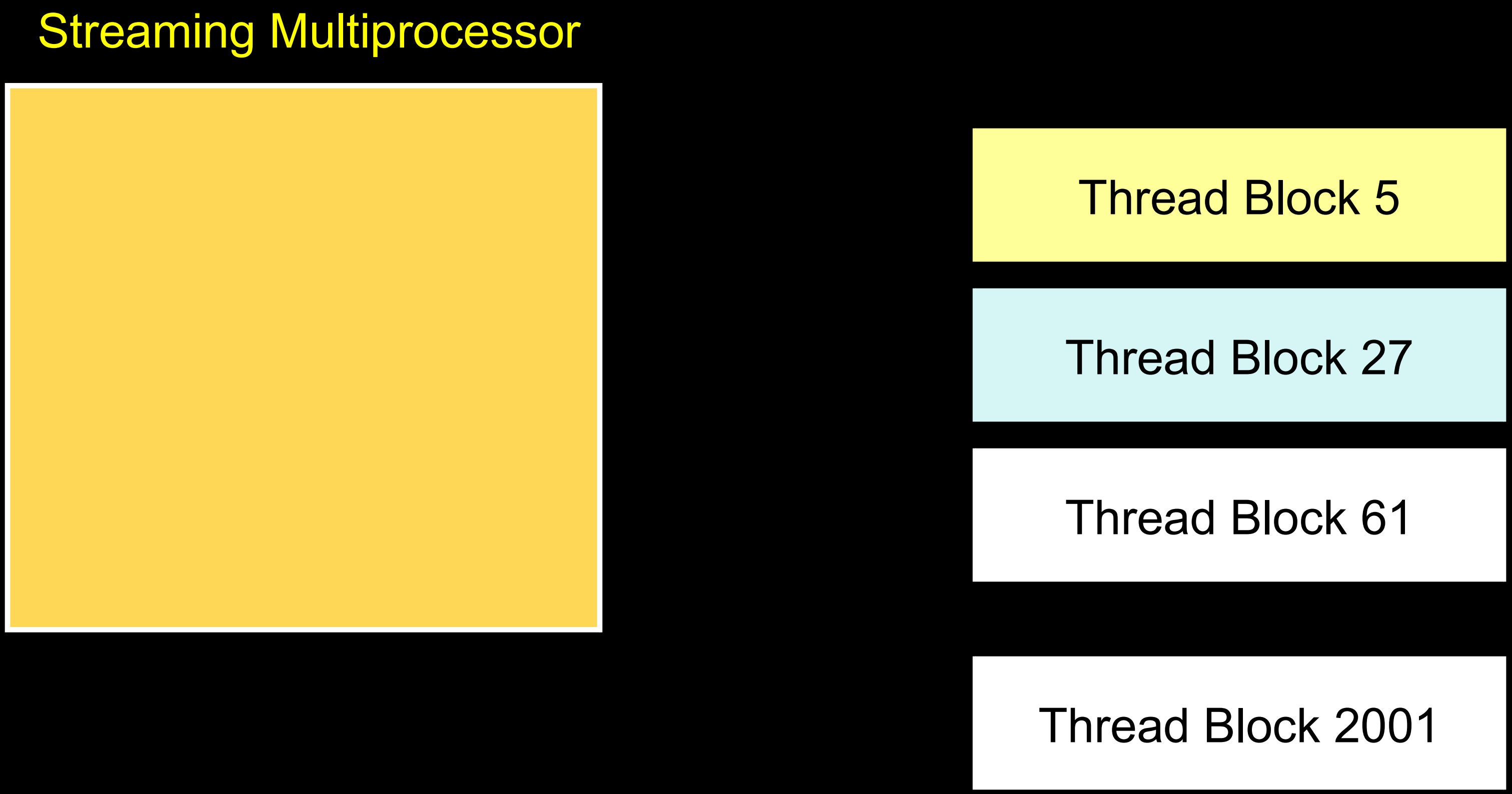
- Can’t use normal load/store for inter-thread communication because of race conditions
-
- Use atomic instructions for sparse and/or unpredictable global communication
- See next lectures for shared memory and scan for other communication patterns
- Decompose data (very limited use of single global sum/max/min/etc.) for more parallelism
-

SM EXECUTION & DIVERGENCE

How an SM executes threads

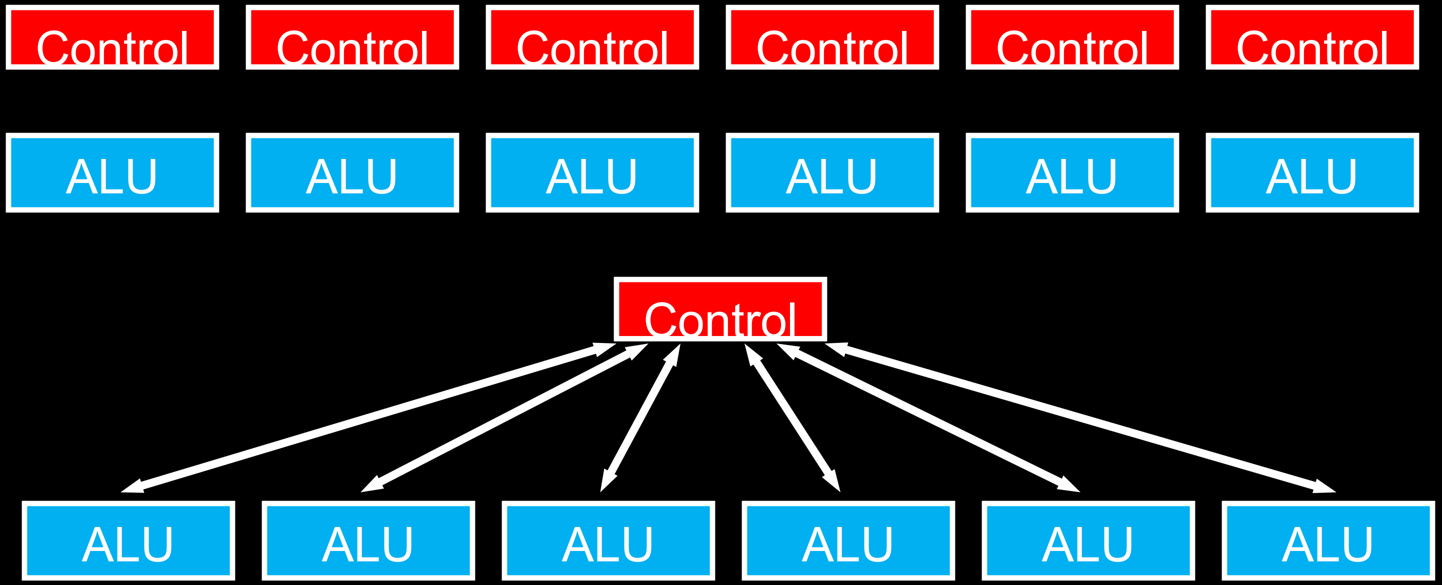
- Overview of how a Stream Multiprocessor works
- SIMT Execution
- Divergence

Scheduling Blocks onto SMs



- HW Schedules thread blocks onto available SMs
- No guarantee of ordering among thread blocks
- HW will schedule thread blocks as soon as a previous thread block finishes

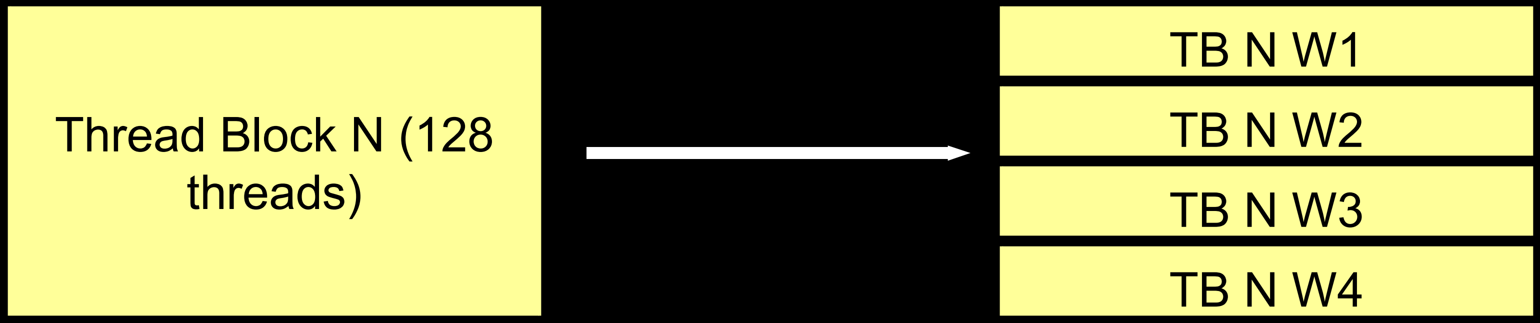
Warps



- A **warp** = 32 threads launched together
- Usually, execute together as well

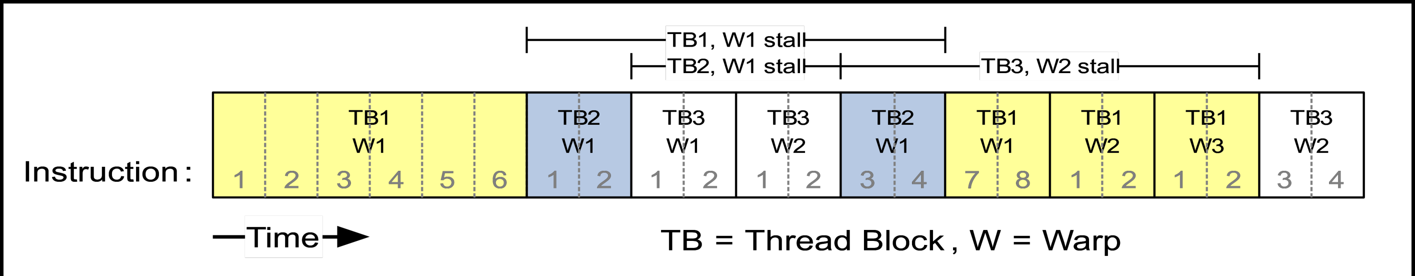
Mapping of Thread Blocks

- Each thread block is mapped to one or more warps
- The hardware schedules each warp independently



Thread Scheduling Example

- SM implements zero-overhead warp scheduling
- At any time, only one of the warps is executed by SM *
- Warps whose next instruction has its inputs ready for consumption are eligible for execution
- Eligible Warps are selected for execution on a prioritized scheduling policy
- All threads in a warp execute the same instruction when selected

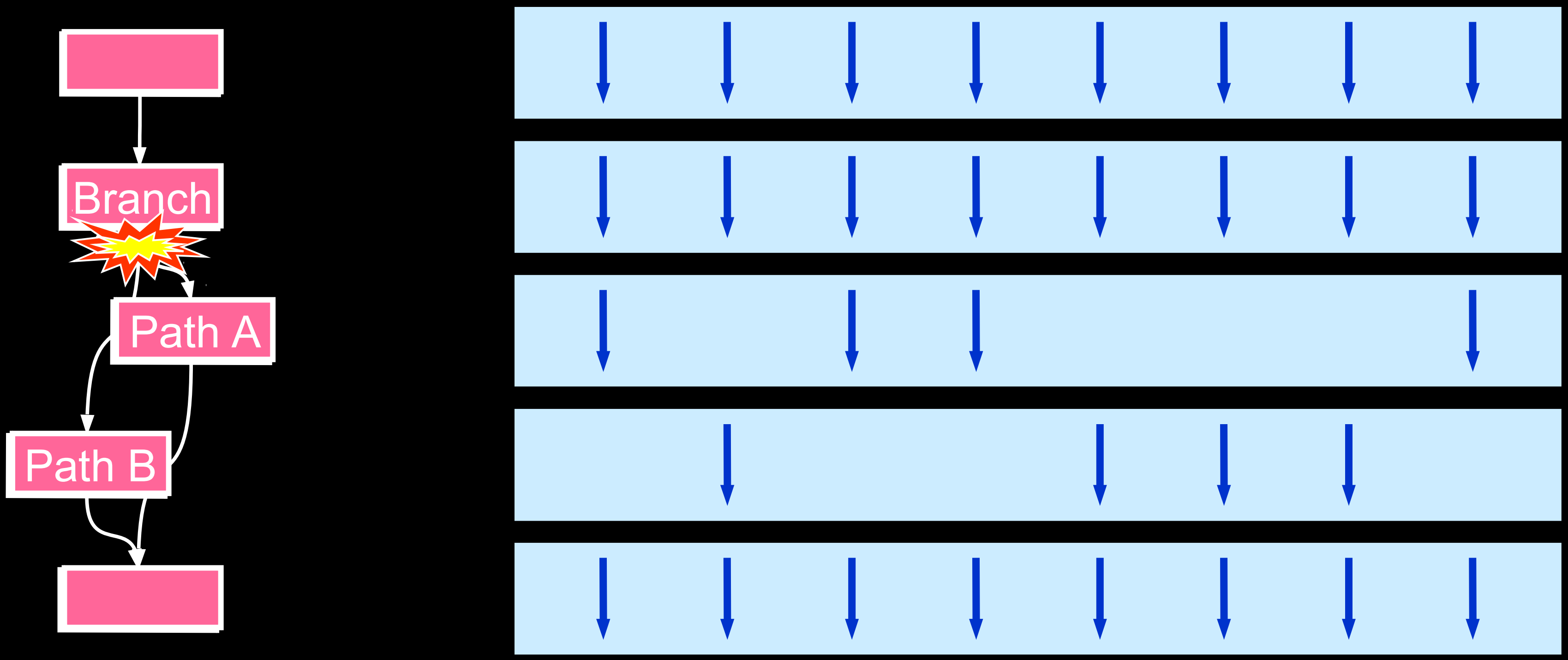


Control Flow Divergence

- What happens if you have the following code?

```
if(foo(threadIdx.x))
{
  do_A();
}
else
{
  do_B();
}
```

Control Flow Divergence



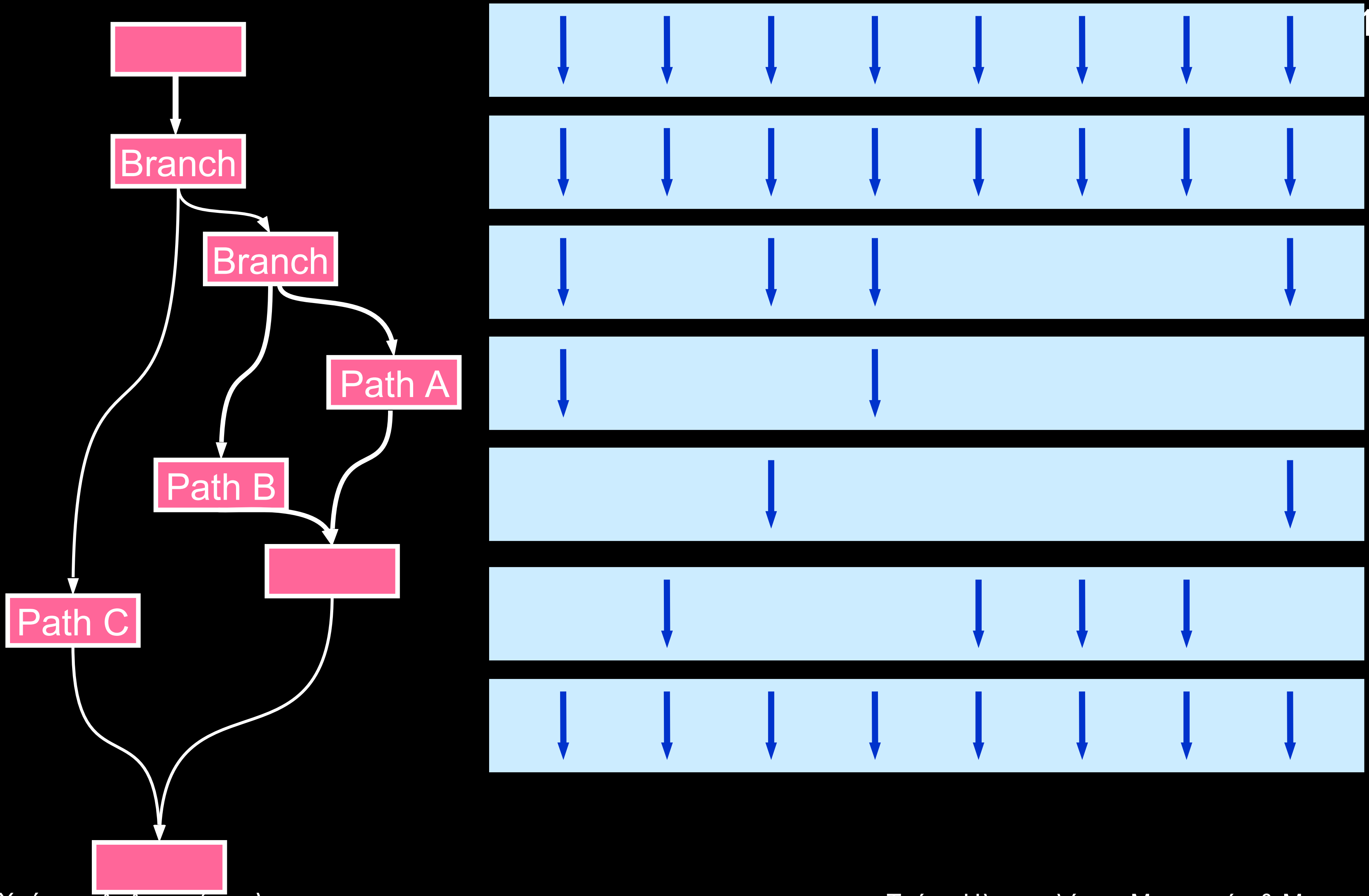
From Fung et al. MICRO '07

Control Flow Divergence

- Nested branches are handled as well

```
if(foo(threadIdx.x))
{
  if(bar(threadIdx.x))
    do_A();
  else
    do_B();
}
else
  do_C();
```

Control Flow Divergence



Control Flow Divergence

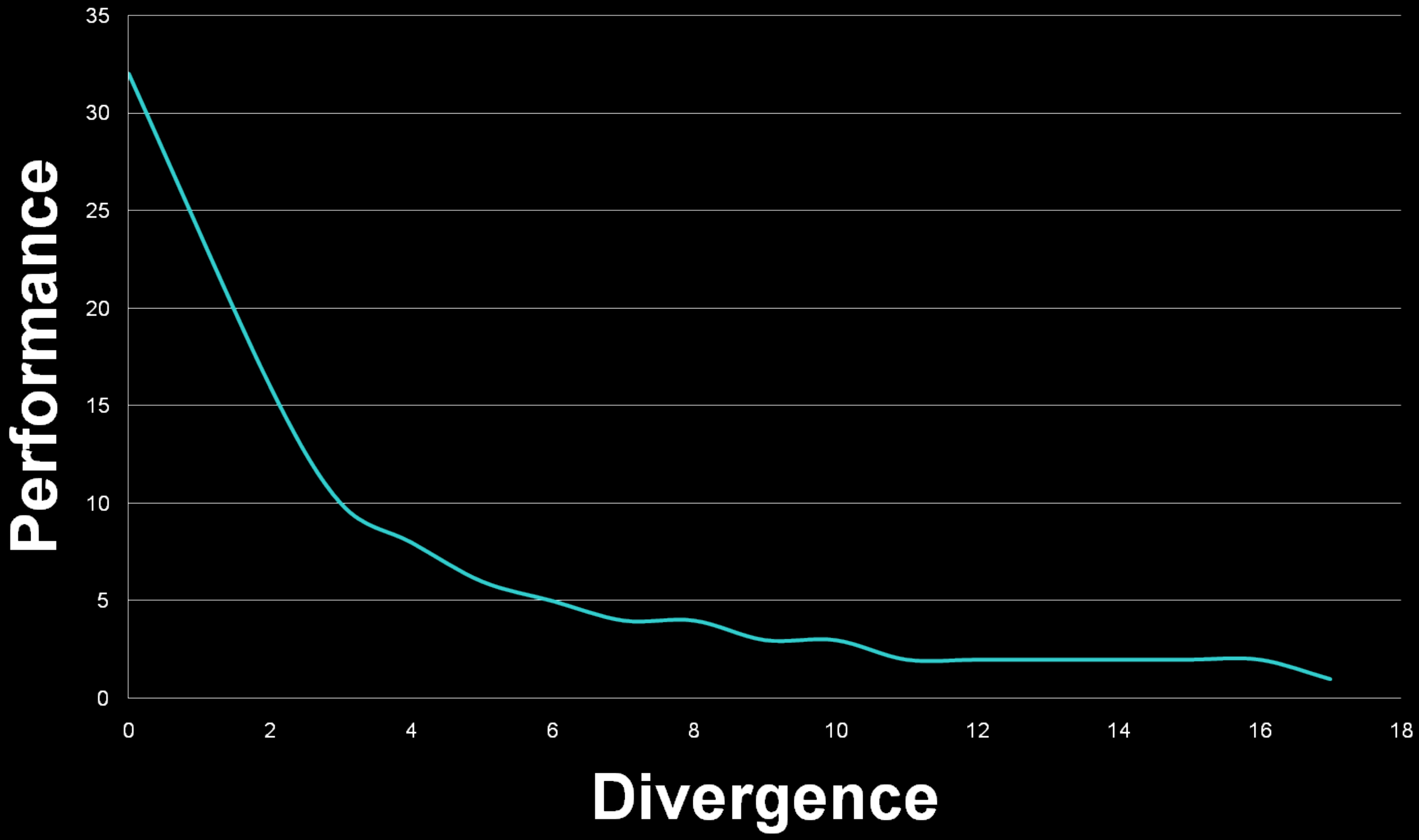
- You don’t have to worry about divergence for correctness (*)
- You might have to think about it for performance
- Depends on your branch conditions

Control Flow Divergence

- Performance drops off with the degree of divergence

```
switch(threadIdx.x % N)
{
  case 0:
    ...
  case 1:
    ...
}
```

Divergence


35
30
25
20
15
10
5
0
0
2
4
6
8
10
12
14
16
18
Performance
Divergence