并行与分布式计算

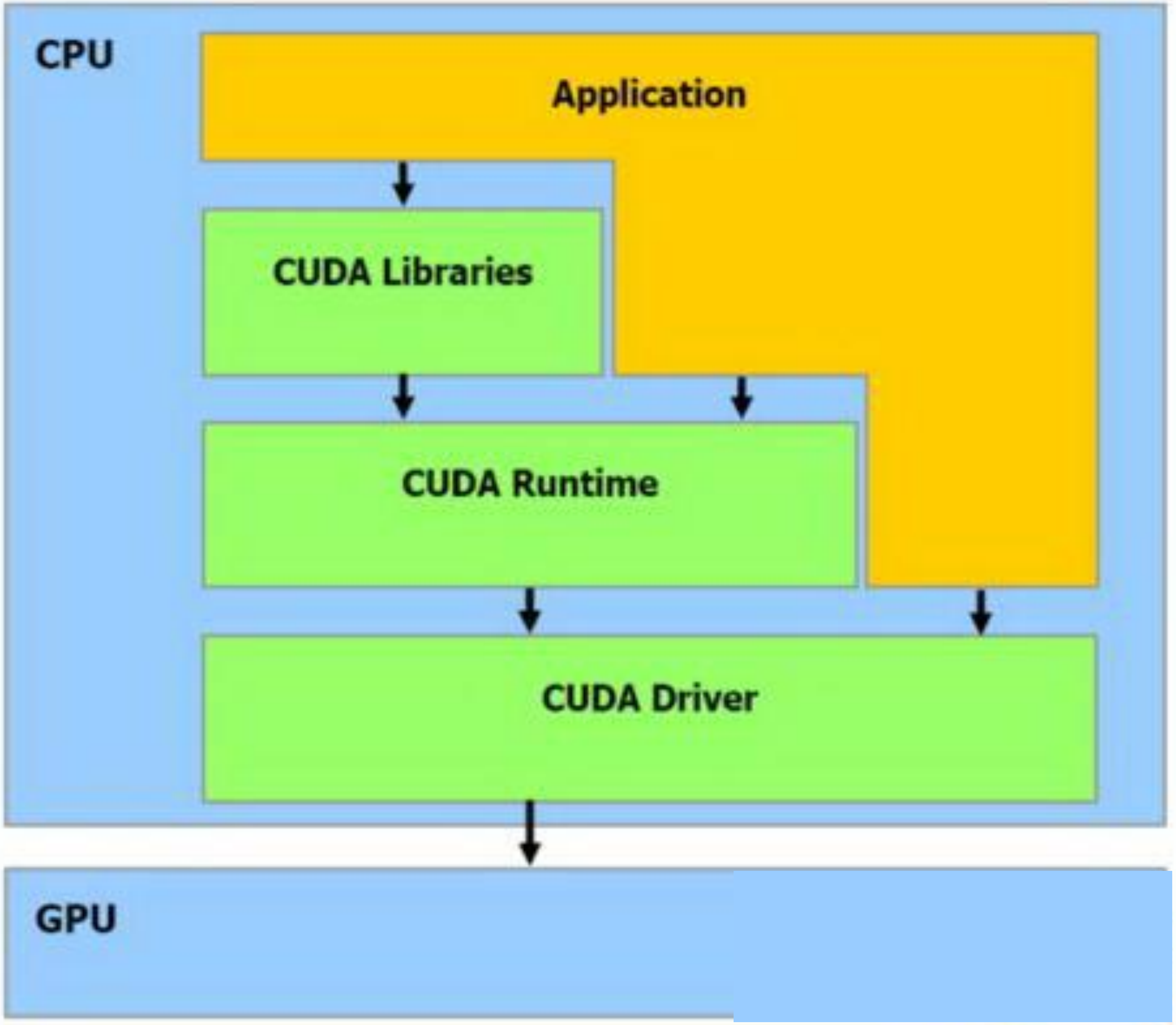
Ying Liu, Prof., Ph.D

School of Computer Science and Technology
University of Chinese Academy of Sciences
Data Mining and High Performance Computing Lab

CUDA Programming Model

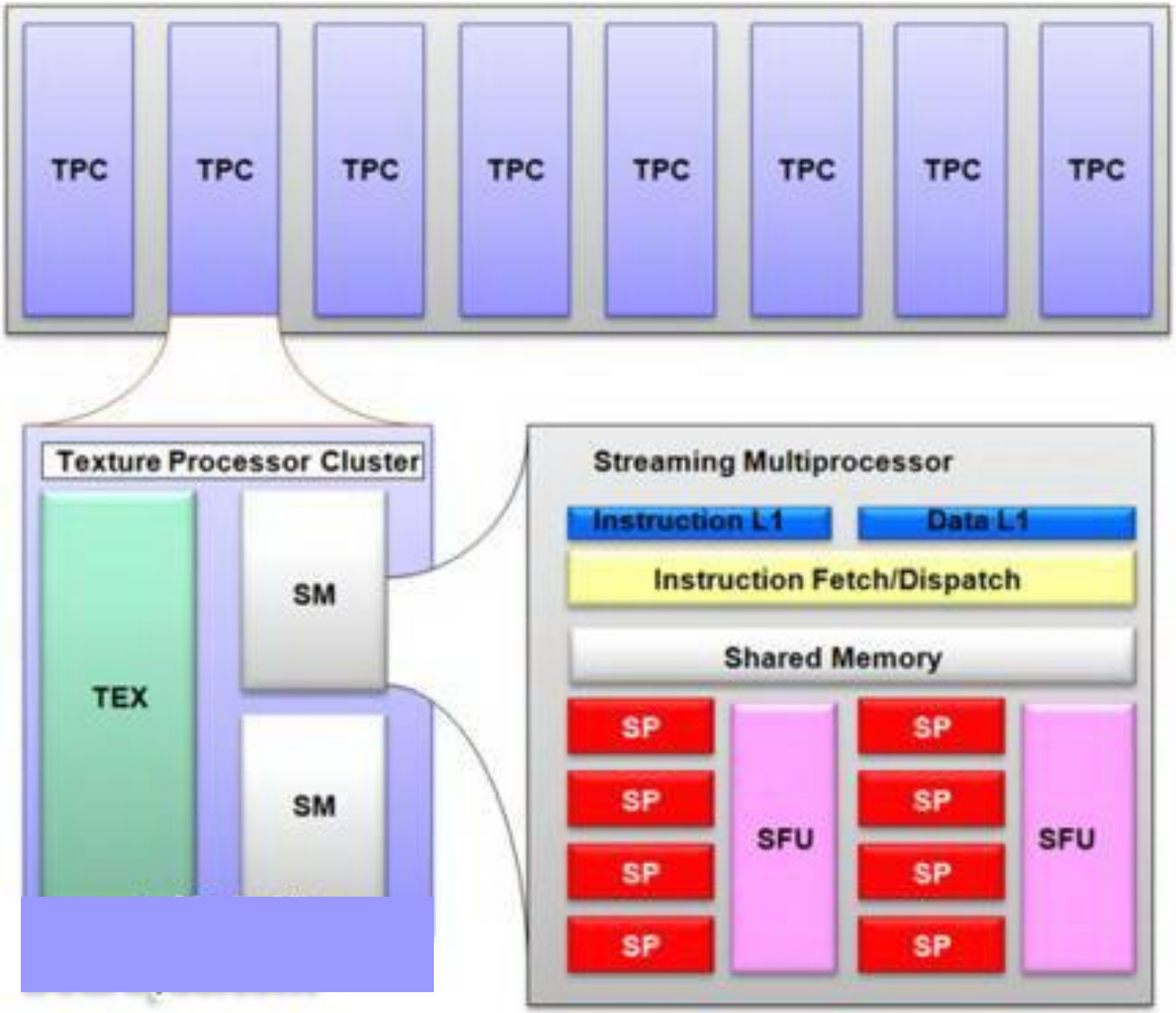
- Compute Unified Device Architecture (CUDA)
- Execution model: kernels, threads, blocks, and grids
- CUDA Basics
- A simple example: matrix multiplication
- CUDA Toolkit: libraries, compiler, debugger, emulator, etc.

CUDA



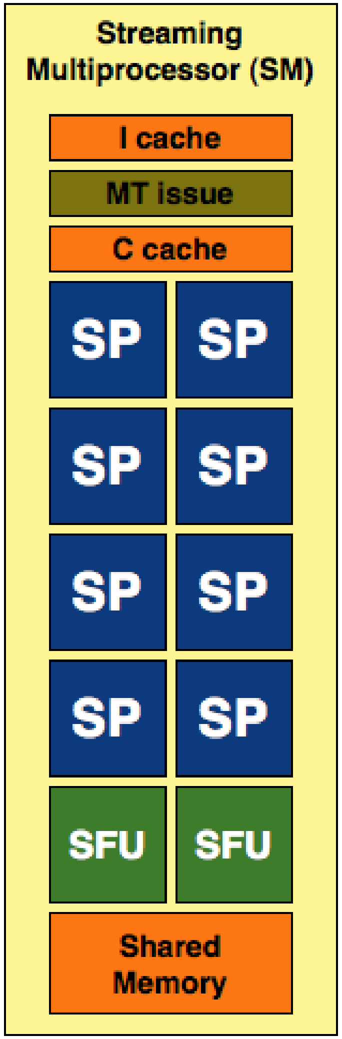
CPU
Application
CUDA Libraries
CUDA Runtime
CUDA Driver
GPU

G80 CUDA Mode



Streaming Multiprocessor (SM)

- An array of SPs
  - 8 streaming processors
  - 2 Special Function Units (SFU)
    - Transcendental operations (e.g. sin, cos) and interpolation
  - A 16KB read/write shared memory
    - Not a cache, but a software-managed data store
  - Multithreading issuing unit
    - Dispatch instructions
  - Instruction cache
  - Constant cache

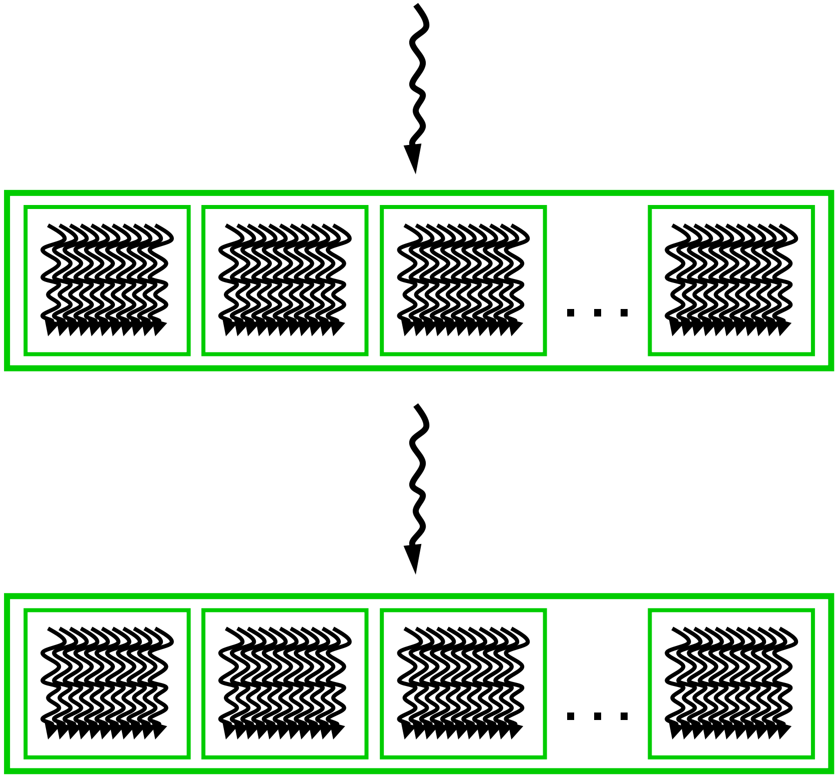


CUDA Device

- A compute device
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
  - Is typically a GPU but can also be another type of parallel processing device
- Kernel — Data-parallel portions of an application which run on many threads

CUDA

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



Serial Code (host)

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

Serial Code (host)

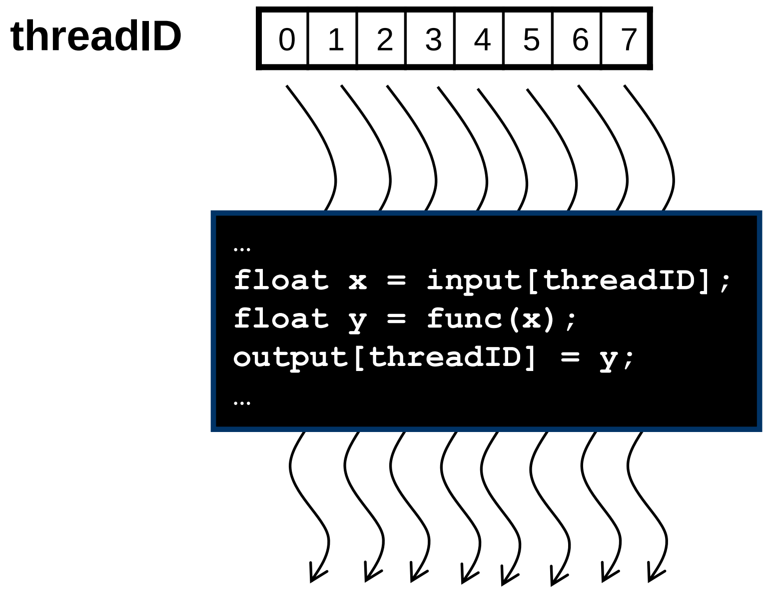
Parallel Kernel (device)
KernelB<<< nBlk, nTid >>>(args);

CUDA Programming Model

- Compute Unified Device Architecture (CUDA)
- Execution model: kernels, threads, blocks, and grids
- CUDA Basics
- A simple example: matrix multiplication
- CUDA Toolkit: libraries, compiler, debugger, emulator, etc.

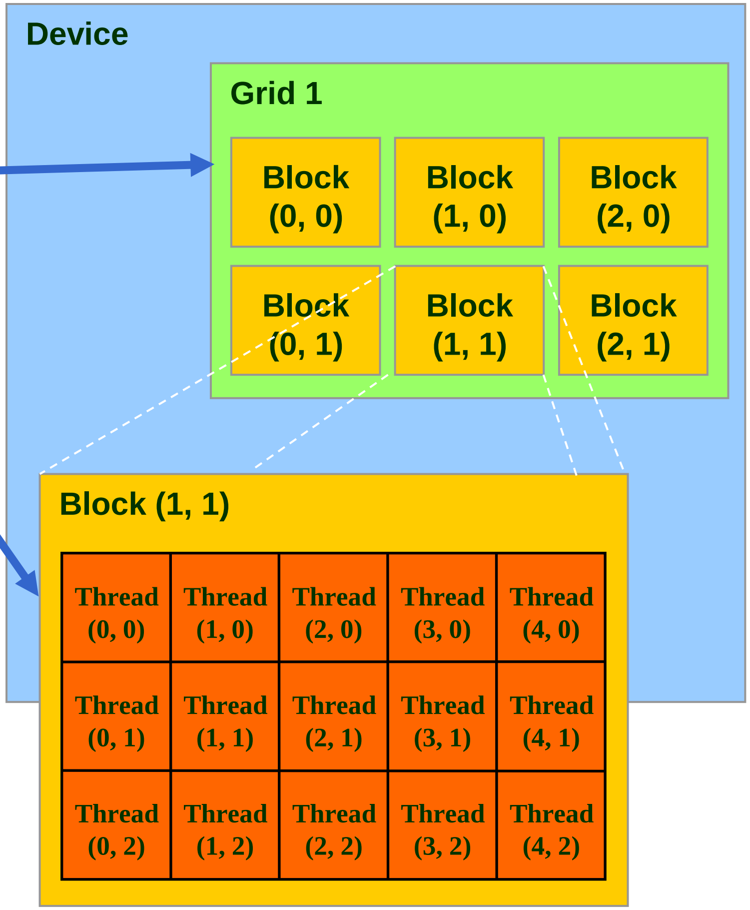
Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same code (SPMD)
  - Each thread has an ID that it uses to compute memory addresses and make control decisions



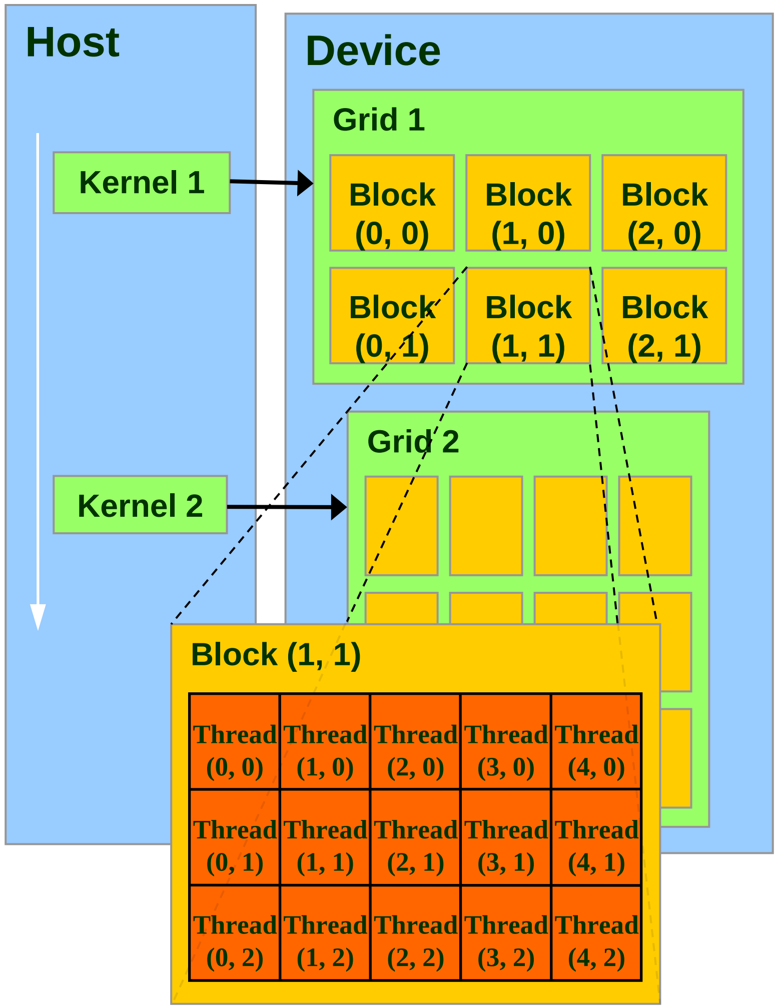
Block IDs and Thread IDs

- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplify memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …

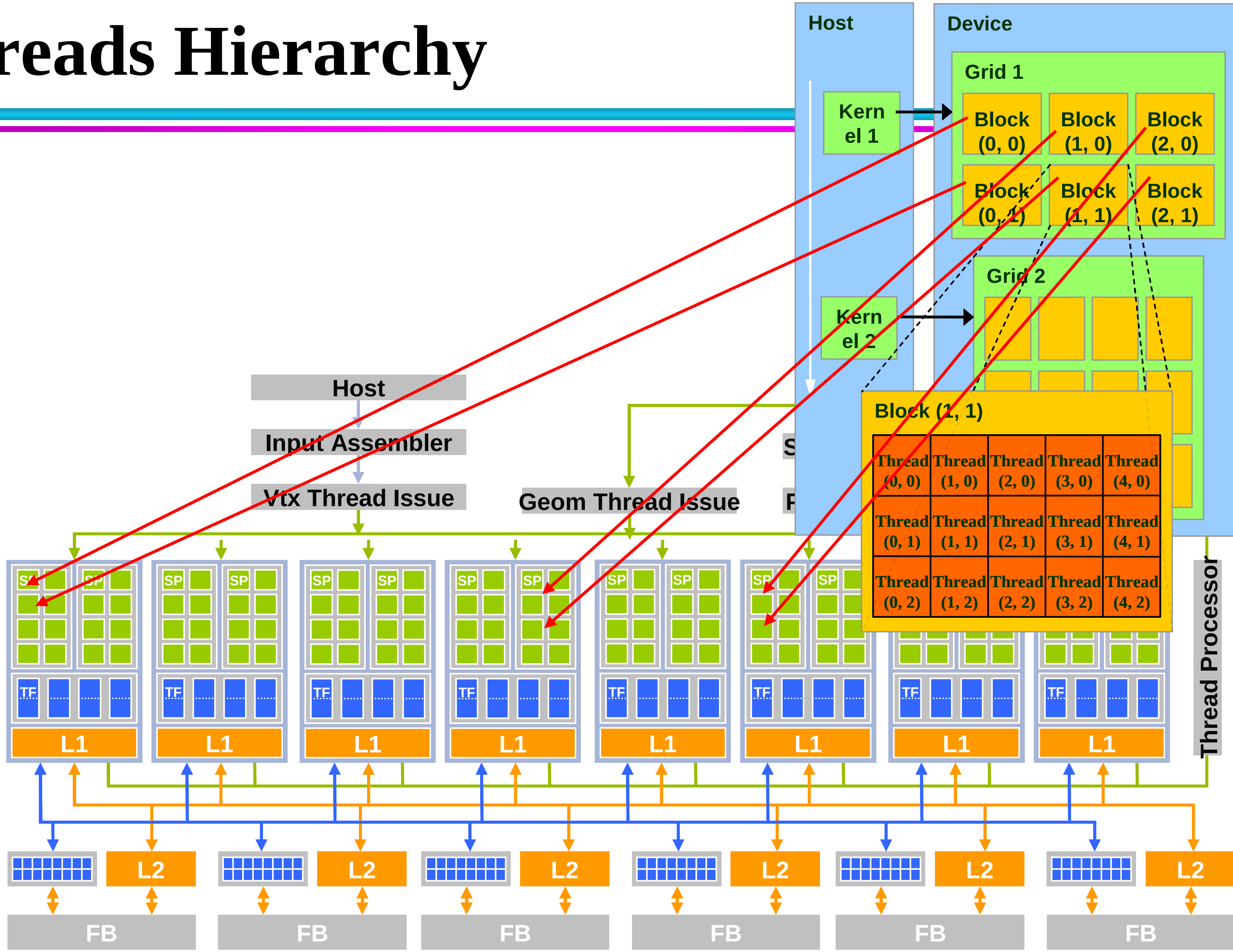


Threads Hierarchy

- Thread: parallel execution
- Thread block
  - Cooperative Thread Array (CTA)
  - Synchronization between threads
  - Share data in shared memory
  - 1D, 2D, 3D
  - Max 512 threads
- Grid
  - A group of thread block
  - 1D, 2D, 3D
  - Share data in global memory
  - Dynamically scheduled at runtime
- Kernel
  - The part of code running on threads

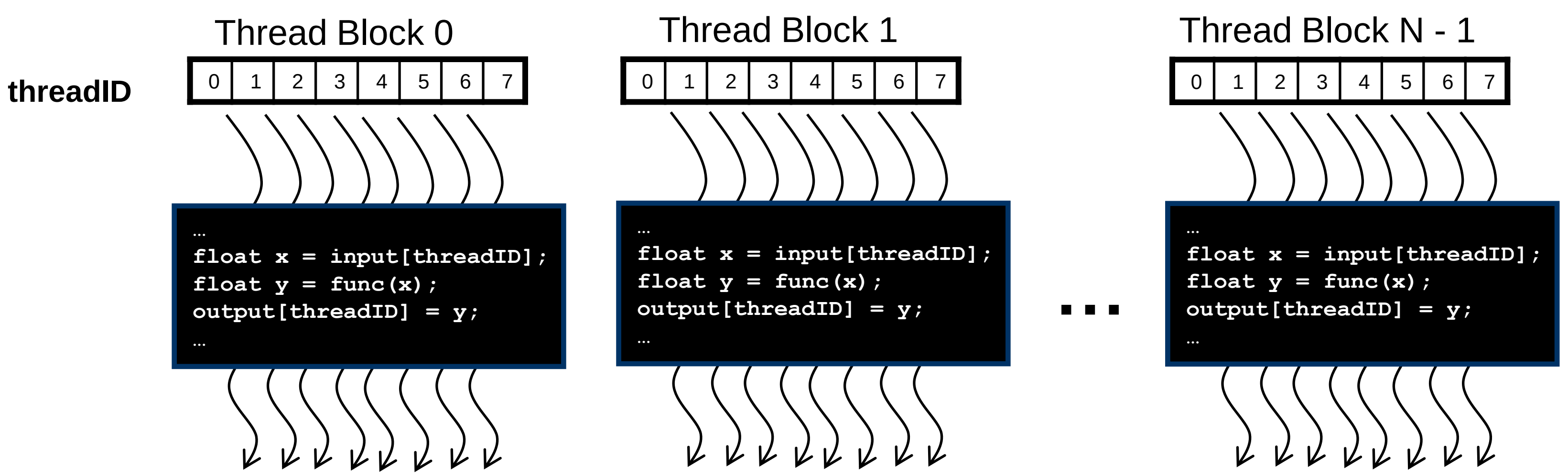


Threads Hierarchy

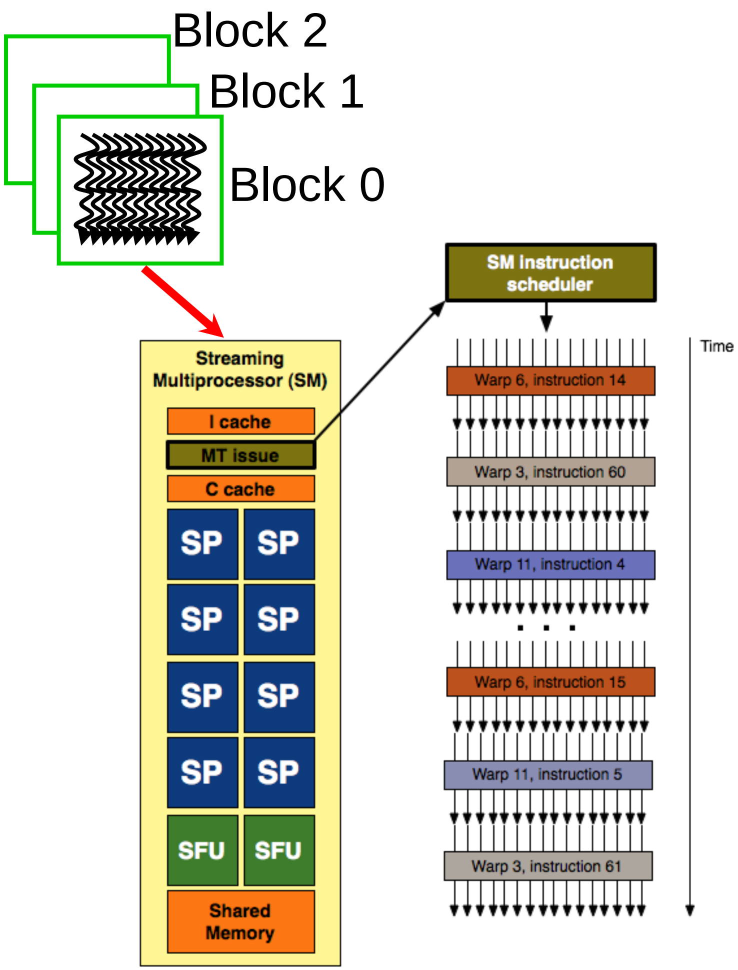


Thread Blocks

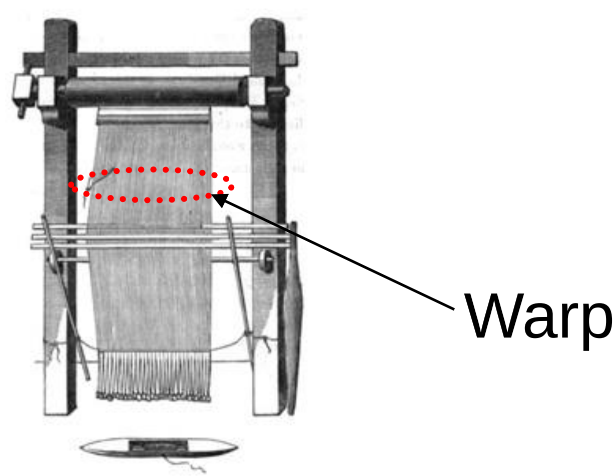
- Divide monolithic thread array into multiple blocks
  - Threads within a block cooperate via *shared memory, atomic operations* and *barrier synchronization*
  - Threads in different blocks cannot cooperate



Thread Execution



- A warp of 32 threads physically running on SM
  - Sharing instructions
  - 4 cycles for 1 warp instruction
  - Dynamically scheduled by SM
    - Executed when operands ready



CUDA Programming Model

- Compute Unified Device Architecture (CUDA)
- Execution model: kernels, threads, blocks, and grids
- CUDA Basics
- A simple example: matrix multiplication
- CUDA Toolkit: libraries, compiler, debugger, emulator, etc.

CUDA Extends C

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

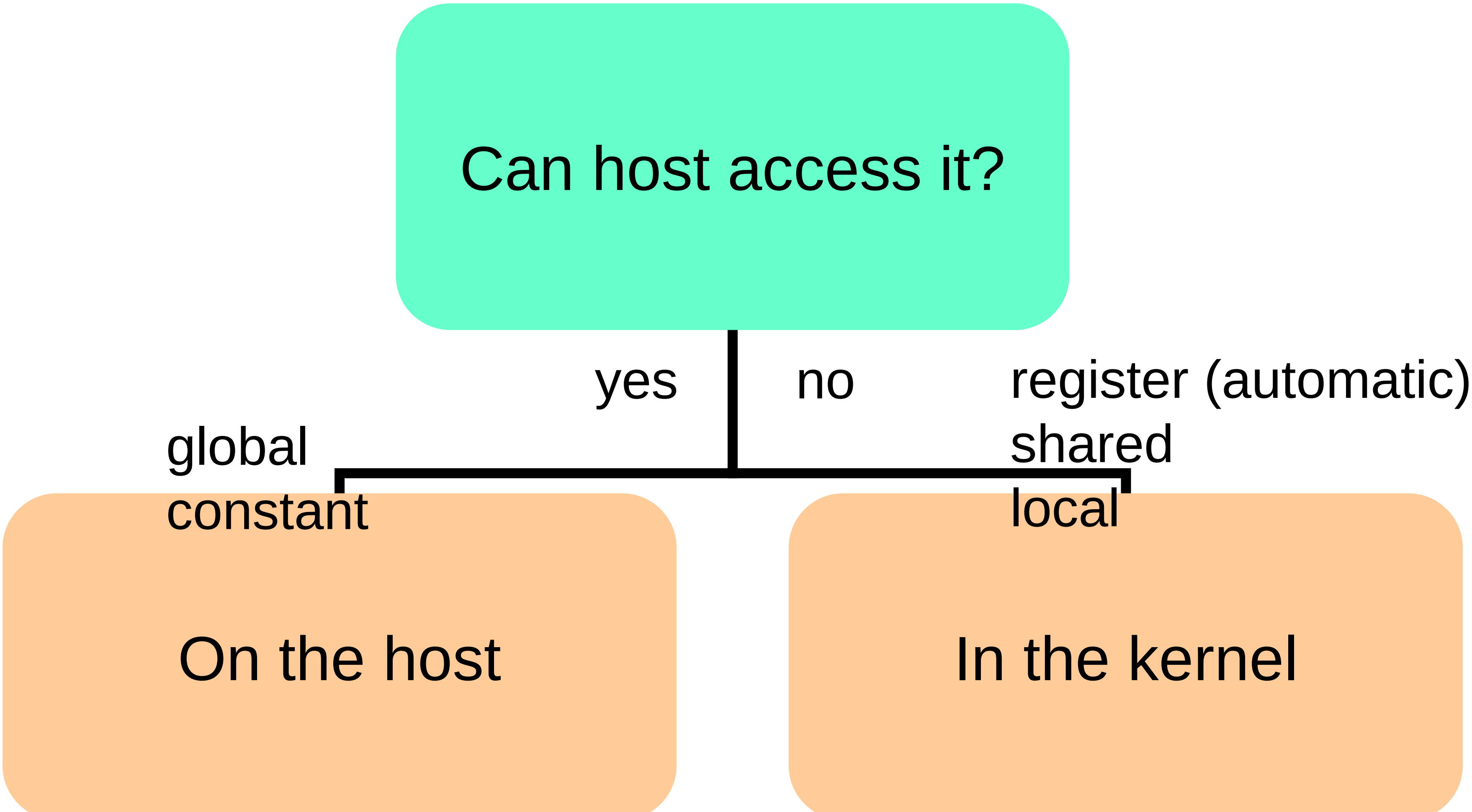
```
__device__ float filter[N];
__global__ void convolve (float *image)  {

   __shared__ float region[M];
   ...

   region[threadIdx] = image[i];

   __syncthreads()
   ...

  image[j] = result;
}

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

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

CUDA Variable Type Qualifiers

Variable declaration		Memory	Scope	Lifetime
`__device__ __local__`	`int LocalVar;`	local	thread	thread
`__device__ __shared__`	`int SharedVar;`	shared	block	block
`__device__`	`int GlobalVar;`	global	grid	application
`__device__ __constant__`	`int ConstantVar;`	constant	grid	application

- __device__ is optional when used with __local__, __shared__, or __constant__
- Automatic variables without any qualifier reside in a register
  - Except arrays that reside in local memory

Where to Declare Variables?



CUDA Variable Declarations

- __device__
  - Resides in the global memory
  - Has the lifetime of an application
  - Is accessible from all the threads within the grid
  - Is accessible from the host through the runtime library
  - Address of a _device_ variable can only be used in device code
- __constant__
  - Resides in constant memory
  - Has the lifetime of an application
  - Is accessible from all the threads within the grid
  - Is accessible from the host through the runtime library
  - Cannot be assigned to from the device, only from the host through the runtime library

CUDA Variable Declarations (Cont.)

- __shared__
  - Resides in the shared memory of a thread block
  - Has the lifetime of the block
  - Is only accessible from all the threads within the block
  - Cannot have initialization as part of declaration
  - Address of a _device_ variable can only be used in device code
- Automatic variable
  - Declared with no qualifier
  - Resides in a register or local memory
  - Has the lifetime of the thread
  - Thread private

Variable Type Restrictions

- In compute capability 1.x, pointers are restricted to only point to memory allocated or declared in the global memory if the compiler is not able to resolve whether they point to the shared memory or the global memory

- In compute capability 2.x, pointers are supported without any restriction

- CUDA does not support function pointers in the device code part

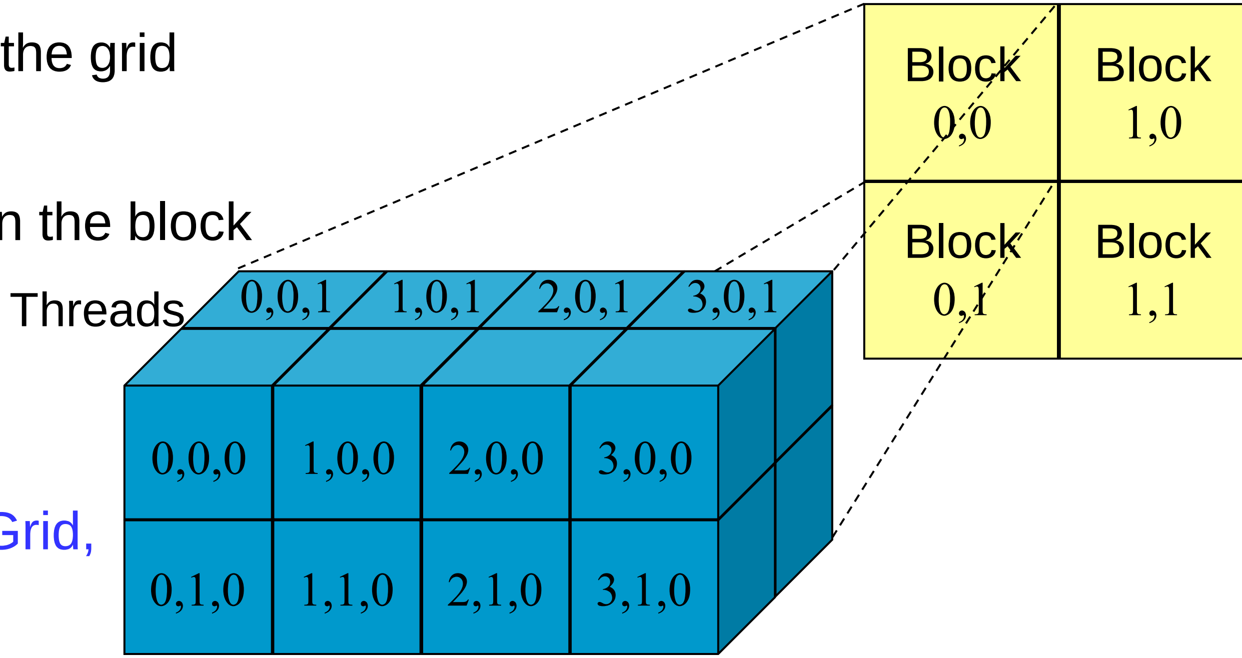
Built-in Vector Types

- Built-in
  - int1, int2, int3, int4, float1, float2, float3, float4, …
  - Define by a constructor function make_<type name>
    - int4 make_int4 (int x, int y, int z, int w)
    - int4 iv(1, 2, 3, 4)
      - iv.x = 1, iv.y = 2, iv.z = 3, iv.w = 4

Built-in dim3 Type

- dim3 gridDim
  - Dimensions of the grid in blocks (gridDim.z unused)
- dim3 blockDim
  - Dimensions of the block in threads
- dim3 blockIdx
  - Block index within the grid
- dim3 threadIdx
  - Thread index within the block

```
dim3 dimGrid(2, 2)
dim3 dimBlock(4, 2, 2)
kernelFunction<<< dimGrid, dimBlock>>>(…)
```



CUDA Function Declarations

	Executed on the:	Only callable from the:
__device__ float DeviceFunc()	device	device
__global__ void KernelFunc()	device	host
__host__ float HostFunc()	host	host

- __global__ defines a kernel function
  - Must return void
- __device__ and __host__ can be used together

CUDA Function Declarations (Cont.)

- __device__ functions cannot have their address taken (CUDA 1.x)
- For functions executed on the device:
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments
- CUDA 2.0+ supports recursion with restriction
  - **__global__** functions do not support recursion
  - **__device__** functions only support recursion in device code compiled for devices

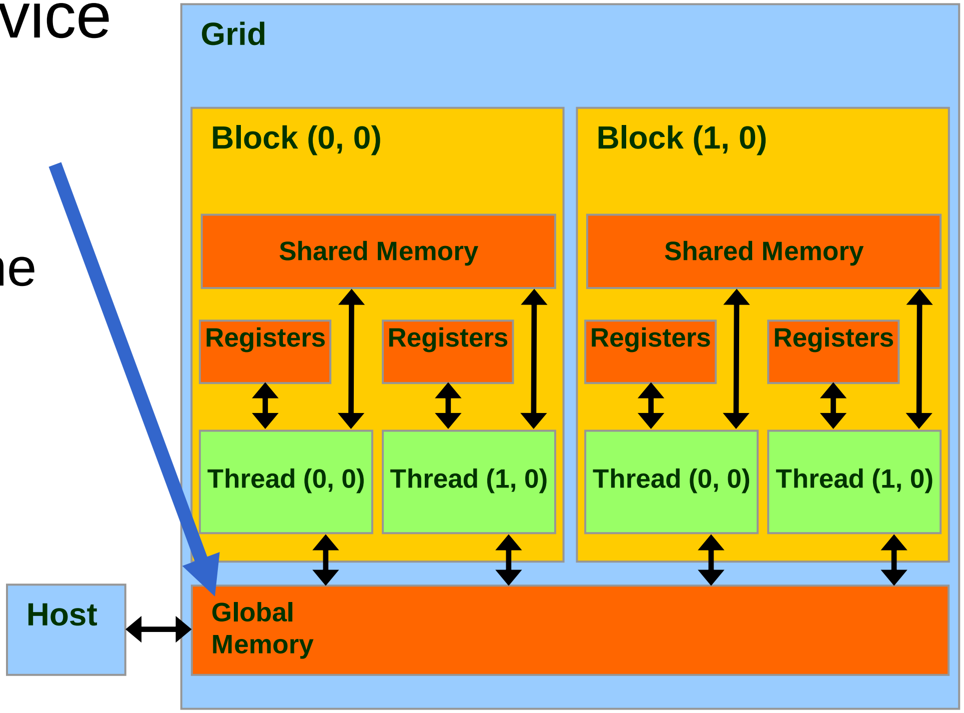
Calling a Kernel Function – Thread Creation

- A kernel function must be called with an execution configuration:

```
__global__ void KernelFunc(...);
…
dim3   DimGrid(100, 50);              // 5000 thread blocks
dim3   DimBlock(4, 8, 8);             // 256 threads per block
size_t SharedMemBytes = 64;           // 64 bytes of shared
    memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes
    >>>(...);
```

CUDA Device Memory Allocation

- cudaMalloc()
  - Allocates object in the device Global Memory
  - Requires two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** allocated object
- cudaFree()
  - Frees object from device Global Memory
    - Pointer to freed object



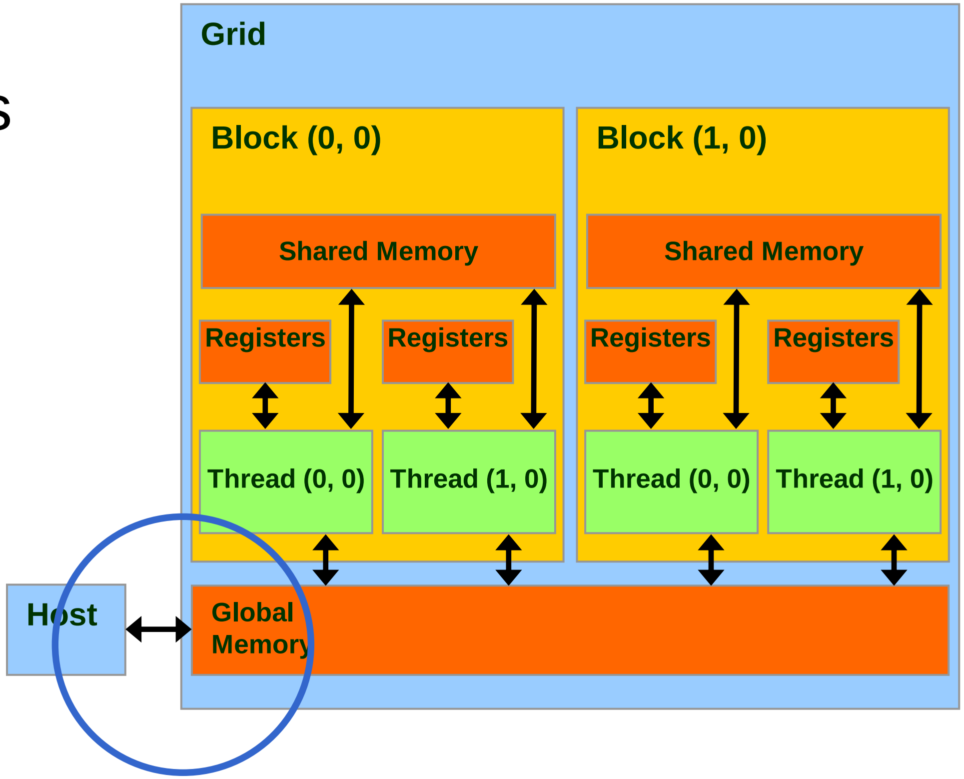
CUDA Device Memory Allocation (Cont.)

- Example code:
  - Allocate a 64 * 64 single precision float array
  - Attach the allocated storage to Md.elements
  - “d” is often used to indicate a device data structure

```
TILE_WIDTH = 64;
__device__ Matrix Md;
int size = TILE_WIDTH * TILE_WIDTH * sizeof(float);


cudaMalloc((void**)&Md.elements, size);

cudaFree(Md.elements);
```

CUDA Host-Device Data Transfer

- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
- Asynchronous transfer



CUDA Host-Device Data Transfer (Cont.)

- Example code :
  - Transfer a 64 * 64 single precision float array
  - M is in host memory and Md is in device memory
  - cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost

```
cudaMemcpy(Md.elements, M.elements, size,
cudaMemcpyHostToDevice);

cudaMemcpy(M.elements, Md.elements, size,
cudaMemcpyDeviceToHost);
```

Threads Synchronization

- void __syncthreads();
  - Barrier
  - Synchronize all the threads within a block
  - Avoid race condition

```
__shared__ float scratch[256];
scratch[threadID] = begin[threadID];
__syncthreads();
int left = scratch[threadID -1];
```

Wait here till all the threads within this block reach this line

Dead-Lock with __syncthreads

- Dead-lock if
  - Some threads have val larger than threshold
  - And others not

```
__global__ void compute(...)
{
    // do some computation for val
    if( val > threshold )
            return;

    __syncthreads();
     // work with val & store it
     return;
}
```

CUDA Programming Model

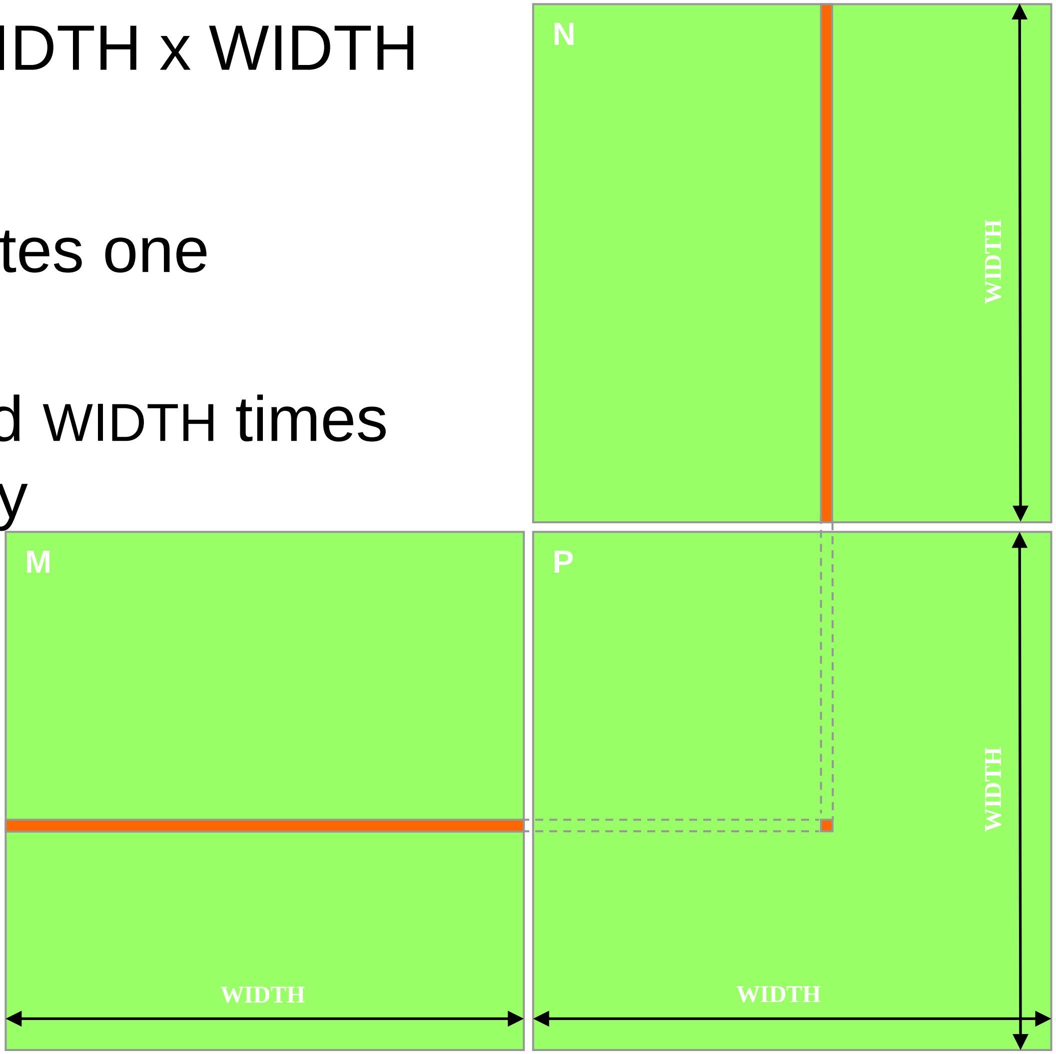
- Compute Unified Device Architecture (CUDA)
- Execution model: kernels, threads, blocks, and grids
- CUDA Basics
- A simple example: matrix multiplication
- CUDA Toolkit: libraries, compiler, debugger, emulator, etc.

A Simple Example: Matrix Multiplication

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

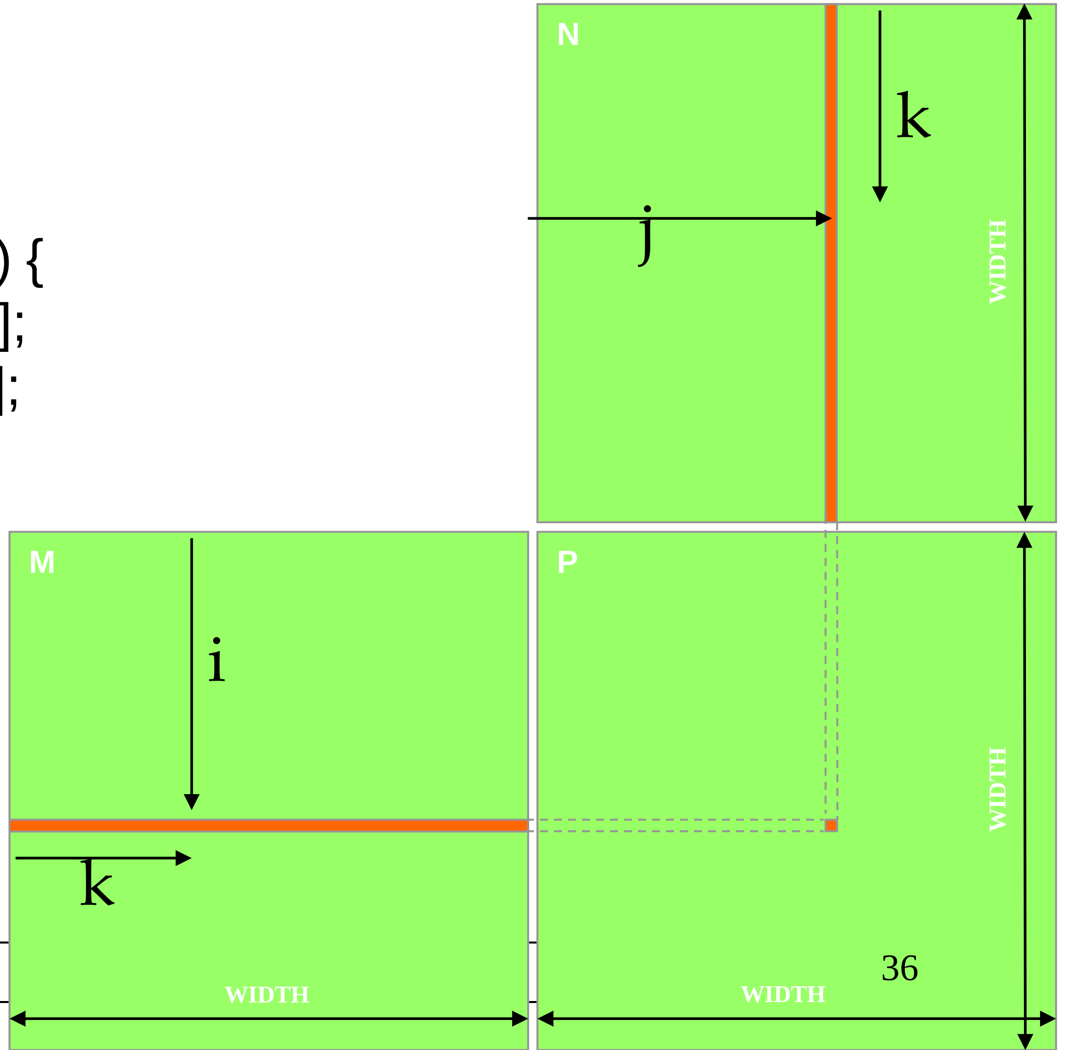
Square Matrix Multiplication

- 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



Step 1: A Simple Host Version in C

```
// Matrix multiplication on the (CPU) host
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);
   __device__ 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)

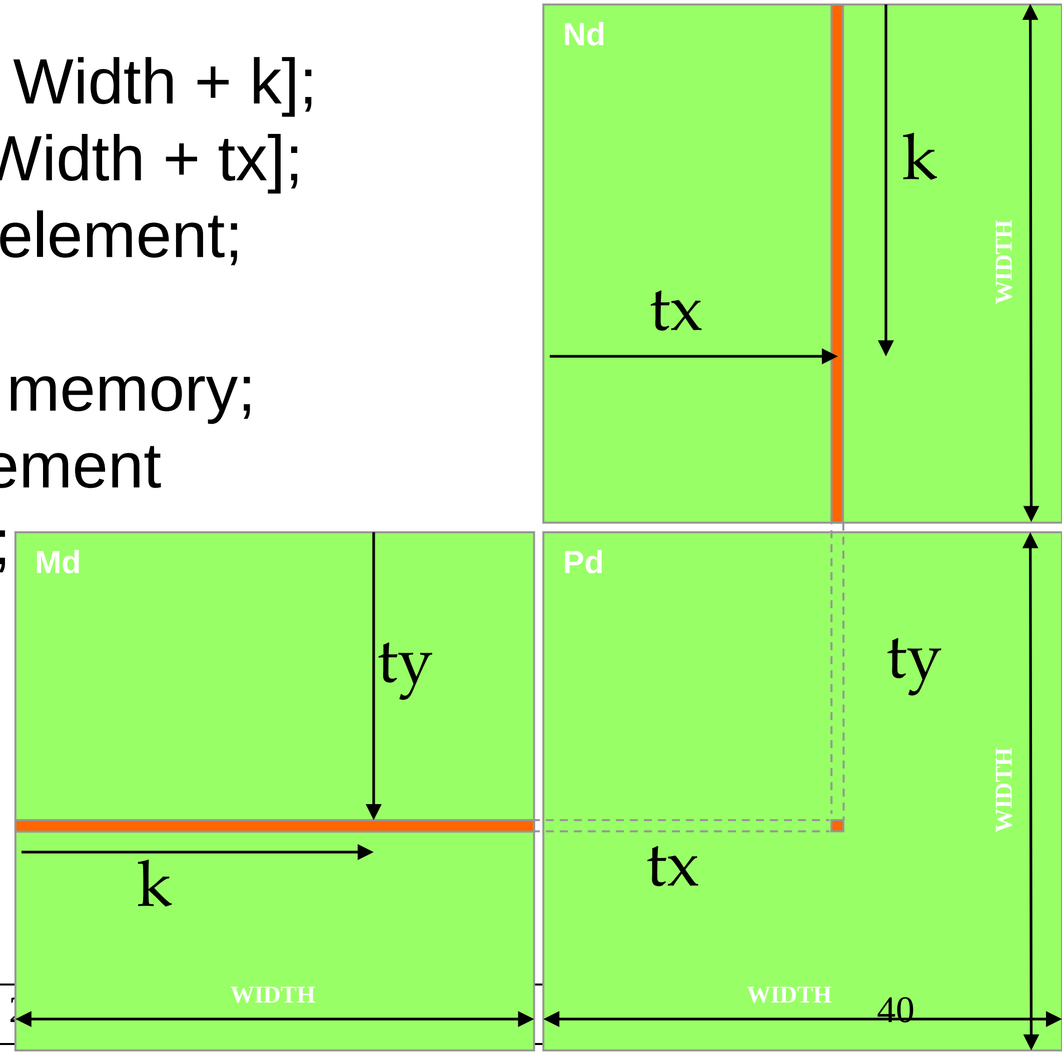
```
2.  // Kernel invocation code – to be shown later
    …

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)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // 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[ty * Width + k];
    float Nelement = Nd[k * Width + tx];
    Pvalue += Melement * Nelement;
}
// Write the matrix to device memory;
// each thread writes one element
Pd[ty * Width + tx] = Pvalue;
}
```

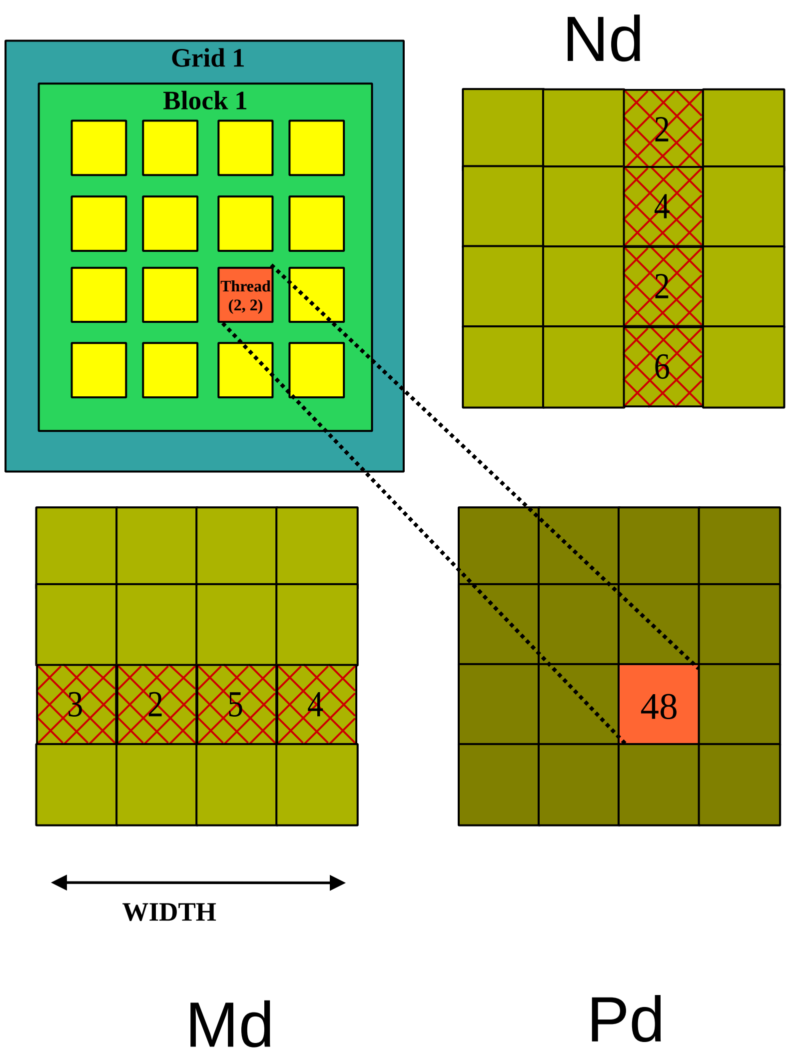


Step 5: Kernel Invocation (Host-side Code)

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

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

Only One Thread Block Used

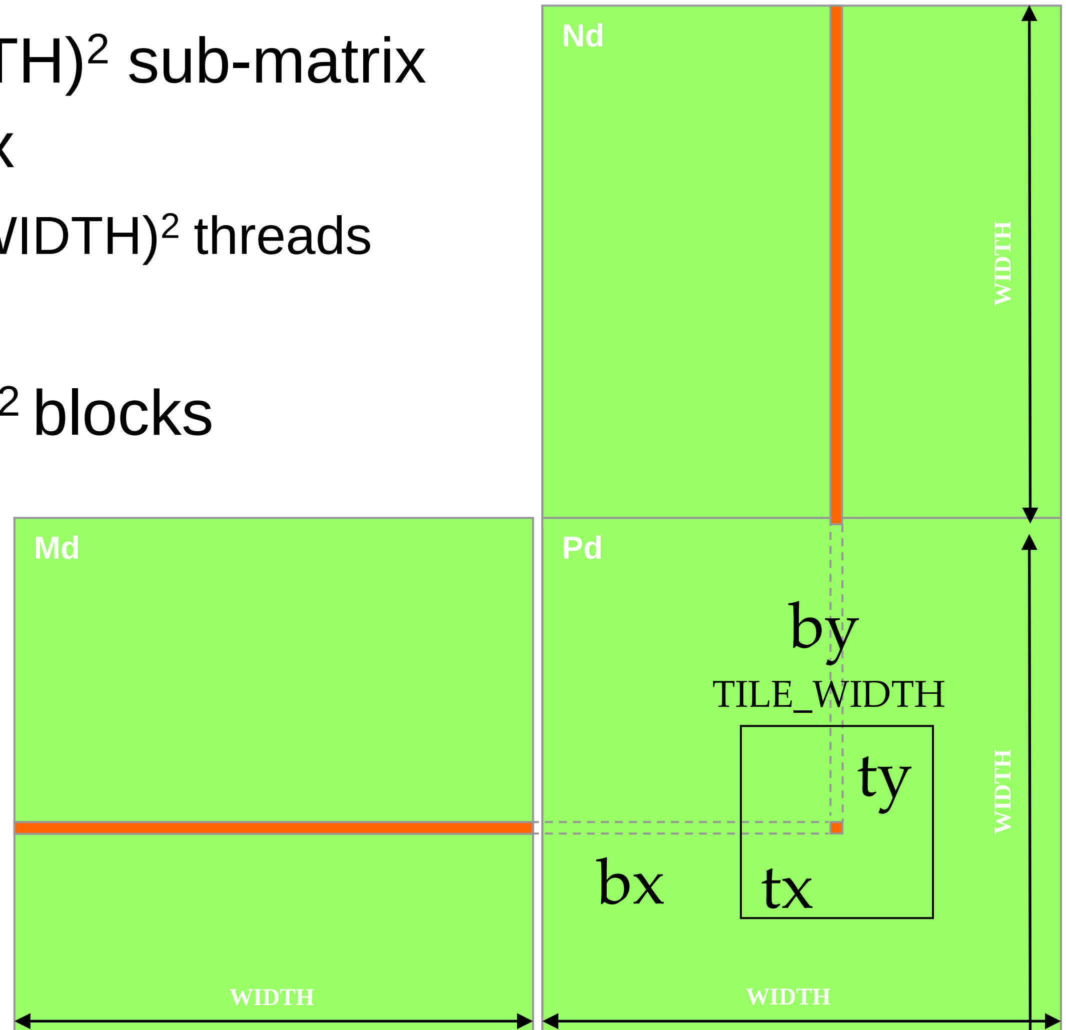
- Only one block of threads compute matrix Pd
  - Each thread computes one element of Pd
- Each thread
  - Loads a row of matrix Md
  - Loads a column of matrix Nd
  - Perform one multiply and addition for each pair of Md and Nd elements
  - Compute to off-chip memory access ratio close to 1:1 (not very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



Step 6: Handling Arbitrary Sized Square Matrices

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

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



CUDA Programming Model

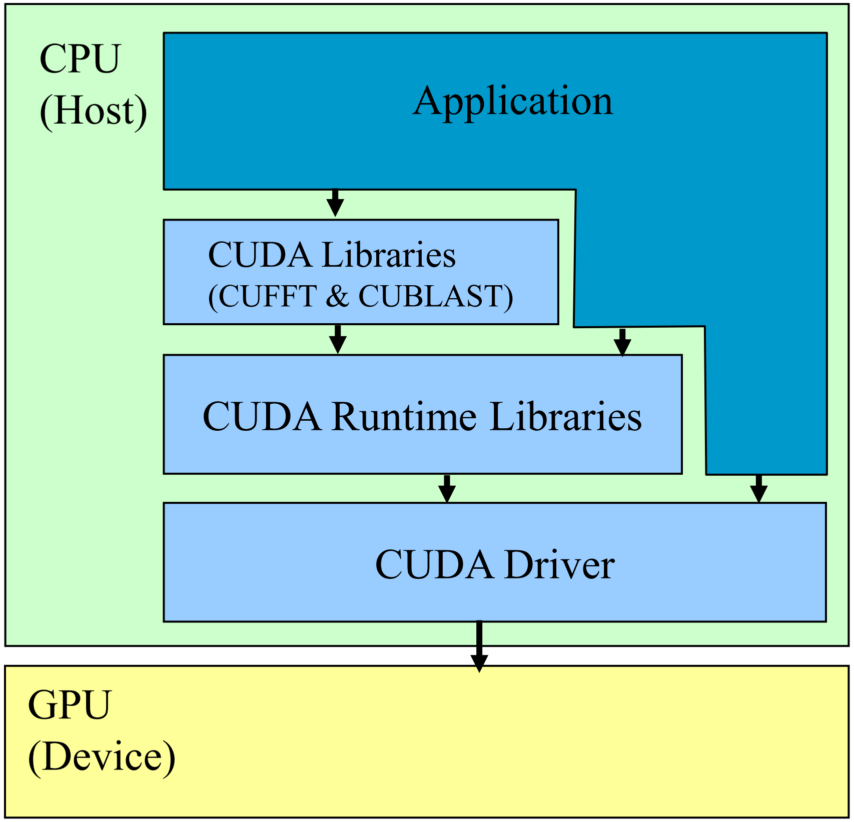
- Compute Unified Device Architecture (CUDA)
- Execution model: kernels, threads, blocks, and grids
- CUDA basics
- A simple example: matrix multiplication
- CUDA toolkit: libraries, compiler, debugger, emulator, etc.

Application Programming Interface

- It consists of:
  - Language extensions
    - To target portions of the code for execution on the device
  - A runtime library can be categorized into:
    - A common component providing built-in vector types and a subset of the C runtime library in both host and device codes
    - A host component to control and access one or more devices from the host
    - A device component providing device-specific functions

CUDA Libraries

- CUBLAS
  - BLAS implementation
- CUFFT
  - FFT implementation
- CUDPP
  - Data parallel primitives
    - Reduction
    - Scan
    - Sort
- CURAND
  - generation of high-quality pseudorandom numbers
- CUSPARSE
  - subroutines for sparse matrices



CUBLAS

- An implementation of BLAS (Basic Linear Algebra Subprograms) on top of the CUDA driver
  - Allows access to the computational resources of NVIDIA GPUs
  - Self-contained APIs
  - Programmers not necessary to interact with the CUDA driver directly
- How to use the library
  - Allocate matrices or vectors on device
  - Data padding
  - Call CUBLAS APIs
  - Deliver the result to the host

Using CUBLAS

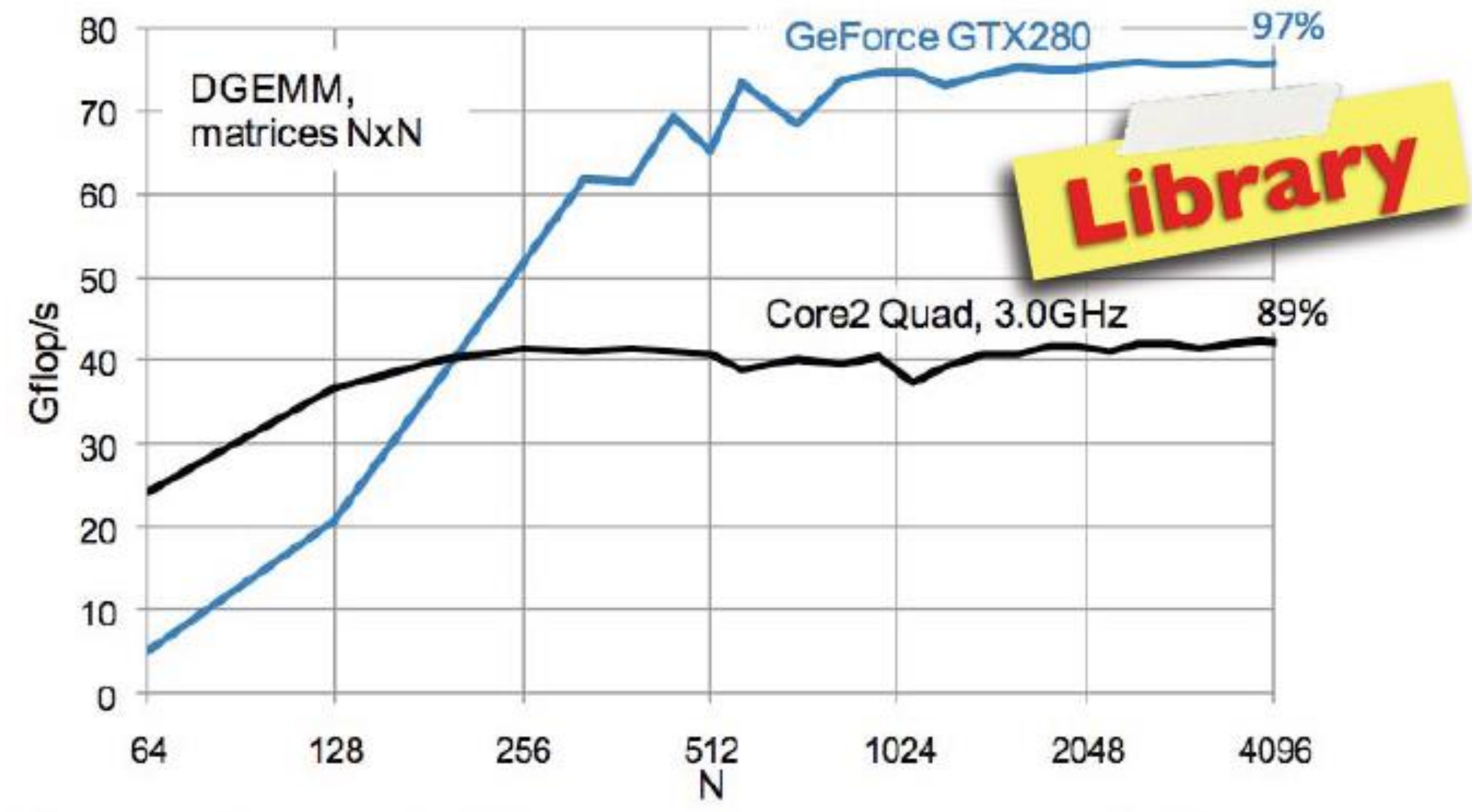
- Interface to CUBLAS library is in cublas.h
- Function naming convention
  - cublas + BLAS name
  - Eg., cublasSGEMM
- Error handling
  - CUBLAS core functions do not return error
    - CUBLAS provides function to retrieve last error recorded
  - CUBLAS helper functions do return error

CUBLAS Helper Functions

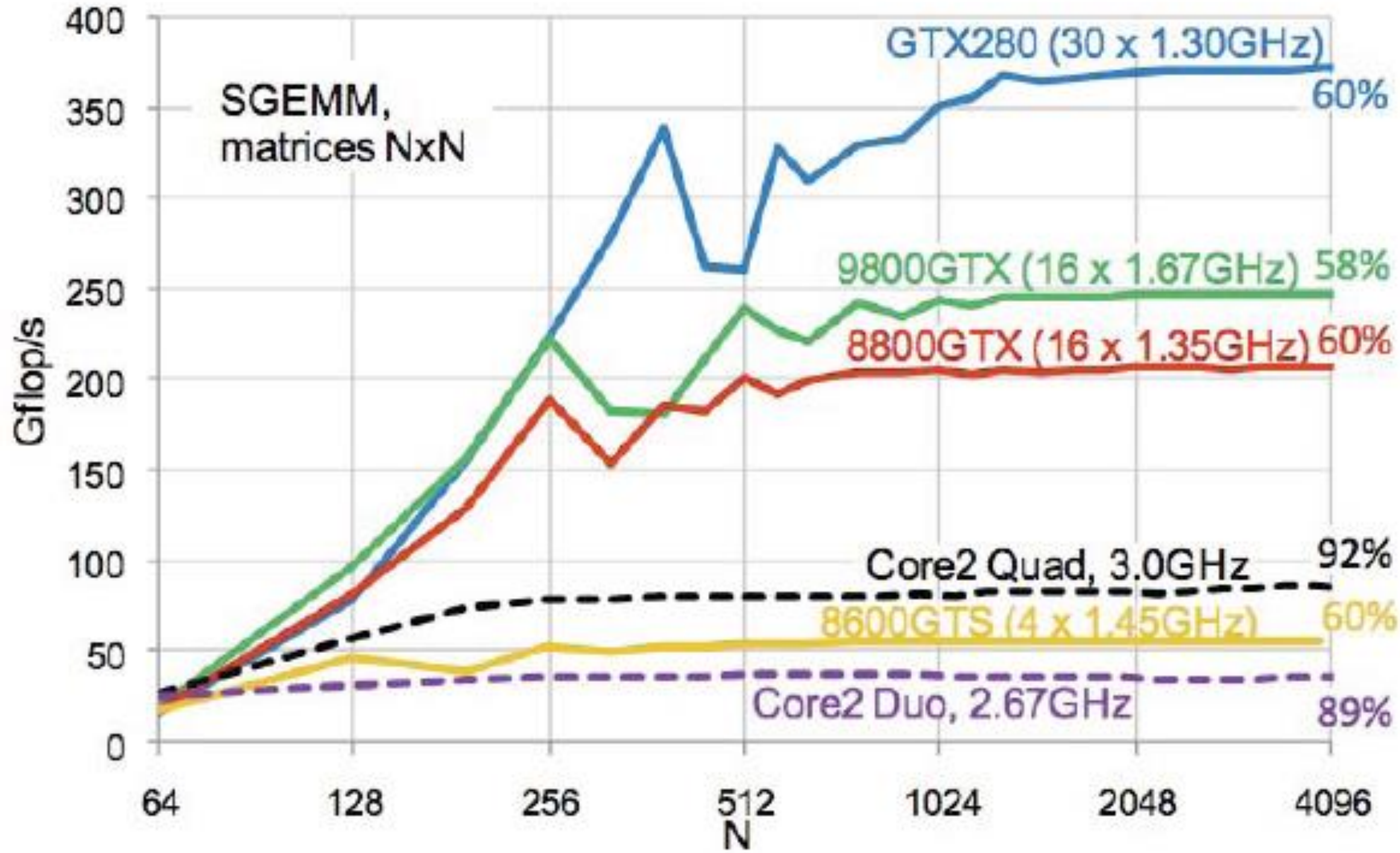
- cublasInit()
  - Initializes CUBLAS library
- cublasShutdown()
  - Releases resources used by CUBLAS library
- cublasGetError()
  - Returns last error from CUBLAS core function (+ resets)
- cublasAlloc()
  - Wrapper around cudaMalloc() to allocate space for array
- cublasFree()
  - destroys object in GPU memory
- cublas[Set|Get][Vector|Matrix]()
  - Copies array elements between CPU and GPU memory
  - Accommodates non-unit strides

sgemmExample.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "cublas.h"

int main(void)
{
  float *a_h, *b_h, *c_h;
  float *a_d, *b_d, *c_d;
  float alpha = 1.0f, beta = 0.0f;
  int N = 2048, n2 = N*N;
  int nBytes, i;

  nBytes = n2*sizeof(float);

  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  c_h = (float *)malloc(nBytes);

  for (i=0; i < n2; i++) {
    a_h[i] = rand() / (float) RAND_MAX;
   b_h[i] = rand() / (float) RAND_MAX;
  }

  cublasInit();

  cublasAlloc(n2, sizeof(float), (void **)&a_d);
  cublasAlloc(n2, sizeof(float), (void **)&b_d);
  cublasAlloc(n2, sizeof(float), (void **)&c_d);

  cublasSetVector(n2, sizeof(float), a_h, 1, a_d, 1);
  cublasSetVector(n2, sizeof(float), b_h, 1, b_d, 1);

  cublasSgemm('n', 'n', N, N, N, alpha, a_d, N,
                          b_d, N, beta, c_d, N);

  cublasGetVector(n2, sizeof(float), c_d, 1, c_h, 1);

  free(a_h); free(b_h); free(c_h);
  cublasFree(a_d); cublasFree(b_d);
  cublasFree(c_d);

  cublasShutdown();
  return 0;
}
```



Rates in double precision matrix-matrix multiply on a GPU and a CPU



Rates in single precision matrix-matrix multiply on a GPU and a CPU

CUFFT

- Efficient data parallel implementation of Fast Fourier Transform (FFT)
- CUFFT is the CUDA FFT library
  - Provides a simple interface for computing parallel FFT on an NVIDIA GPU
  - Allows users to leverage the floating-point power and parallelism of the GPU without having to develop a custom, GPU-based FFT implementation
- Supported features:
  - 1D, 2D and 3D transforms of complex to complex (C2C), real to complex (R2C) and complex to real (C2R)
  - Batched execution for doing multiple 1D transforms in parallel
  - 1D transform size up to 16M elements
  - 2D and 3D transform sizes in the range [2, 16384]
  - In-place and out-of-place transforms for real and complex data

Transform Types

- Library supports real and complex transforms
  - CUFFT_C2C, CUFFT_C2R, CUFFT_R2C
- Directions
  - CUFFT_FORWARD (-1) and CUFFT_INVERSE (1)
    - According to sign of the complex exponential term
- Real and imaginary parts of complex input and output arrays are interleaved
  - cufftComplex type is defined for this

CUFFT Types and Definitions

- cufftHandle
  - Type used to store and access CUFFT plans
- cufftResults
  - Enumeration of API function return values
- cufftReal
  - single-precision, real datatype
- cufftComplex
  - single-precision, complex datatype
- Real and complex transforms
  - CUFFT_C2C, CUFFT_C2R, CUFFT_R2C
- Directions
  - CUFFT_FORWARD, CUFFT_INVERSE

CUFFT Example

```
#include <stdio.h>
#include <math.h>
#include "cufft.h"

int main(int argc, char *argv[])
{
  cufftComplex *a_h, *a_d;
  cufftHandle plan;
  int N = 1024, batchSize = 10;
  int i, nBytes;
  double maxError;

  nBytes = sizeof(cufftComplex)*N*batchSize;
  a_h = (cufftComplex *)malloc(nBytes);

  for (i=0; i < N*batchSize; i++) {
    a_h[i].x = sinf(i);
    a_h[i].y = cosf(i);
  }

  cudaMalloc((void **)&a_d, nBytes);
  cudaMemcpy(a_d, a_h, nBytes,
             cudaMemcpyHostToDevice);

  cufftPlan1d(&plan, N, CUFFT_C2C, batchSize);

  cufftExecC2C(plan, a_d, a_d, CUFFT_FORWARD);
  cufftExecC2C(plan, a_d, a_d, CUFFT_INVERSE);

  cudaMemcpy(a_h, a_d, nBytes,
             cudaMemcpyDeviceToHost);

  // check error - normalize
  for (maxError = 0.0, i=0; i < N*batchSize; i++) {
    maxError = max(fabs(a_h[i].x/N-sinf(i)), maxError);
    maxError = max(fabs(a_h[i].y/N-cosf(i)), maxError);
  }

  printf("Max fft error = %g\n", maxError);

  cufftDestroy(plan);
  free(a_h); cudaFree(a_d);

  return 0;
}
```

Common Runtime Component: Mathematical Functions

- pow, sqrt, cbrt, hypot
- exp, exp2, expm1
- log, log2, log10, log1p
- sin, cos, tan, asin, acos, atan, atan2
- sinh, cosh, tanh, asinh, acosh, atanh
- ceil, floor, trunc, round
- Etc.
  - When executed on the host, a given function uses the C runtime implementation if available
  - These functions are only supported for scalar types, not vector types

Device Runtime Component: Mathematical Functions

- Some mathematical functions (e.g. `sin(x)`) have a less accurate, but faster device-only version (e.g. `__sin(x)`)
  - `__pow`
  - `__log`, `__log2`, `__log10`
  - `__exp`
  - `__sin`, `__cos`, `__tan`
- Use flag "-use_fast_math" in compilation

Device Runtime Component: Synchronization Function

- void __syncthreads();
- Synchronizes all threads in a block
- Once all threads have reached this point, execution resumes normally
- Used to avoid race condition
- Allowed in conditional constructs only if the conditional is uniform across the entire thread block

GPU Atomic Operations

- Allow multiple threads to perform concurrent read-modify-write operations on 32-bit integer in global memory without conflicts (hardware with compute capability 1.1)
  - Associative operations on signed/unsigned int
  - add, sub, min, max, ...
  - and, or, xor
  - Increment, decrement
  - Exchange, compare and swap

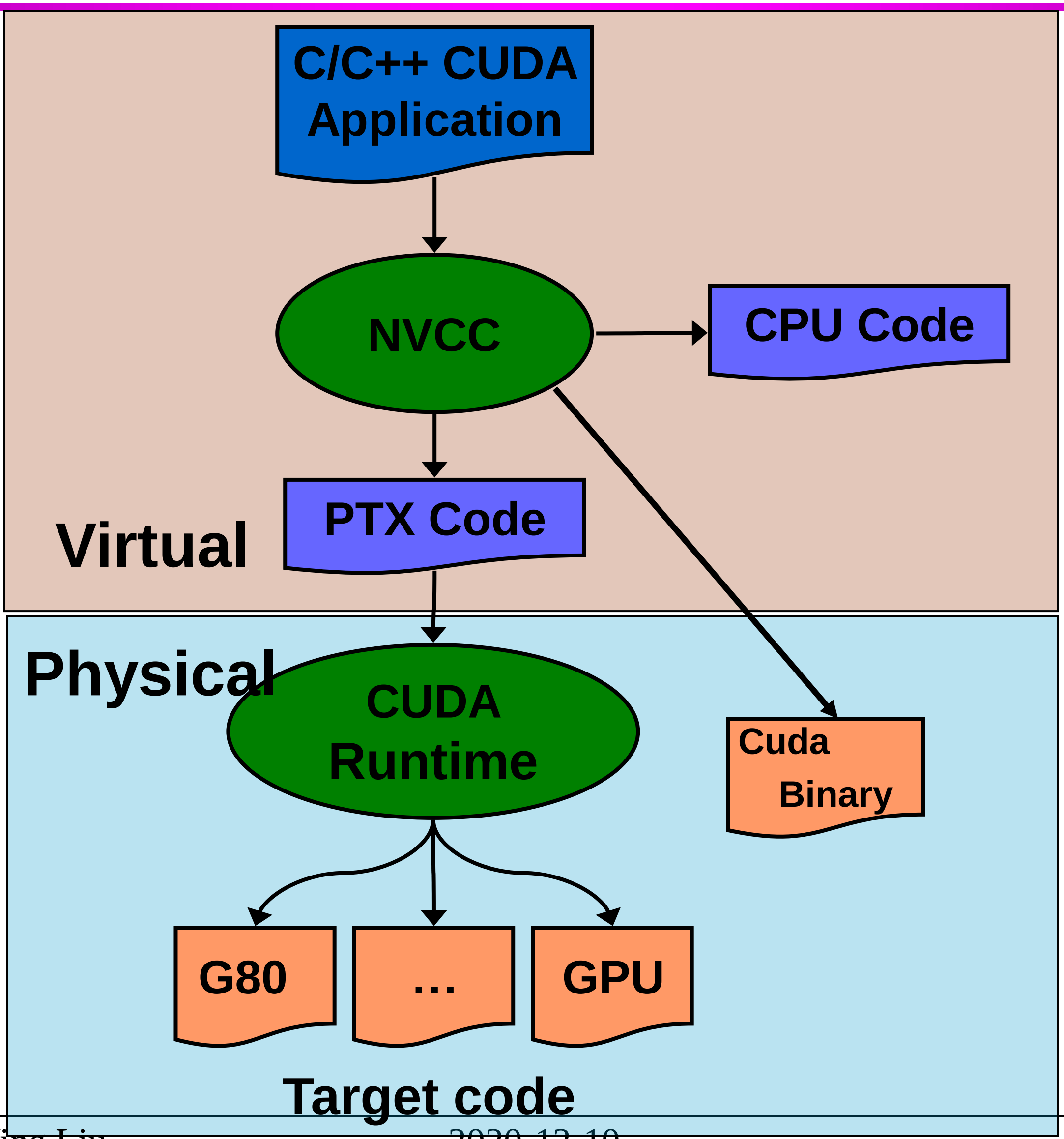
GPU Atomic Operations

- Useful in sorting, reduction operations and building data structures in parallel
- Add the option "-arch sm_30" to the nvcc command line
- Hardware with compute capability 1.2 support atomic operations in shared memory
- Hardware with compute capability 2.x supports atomic operations on 32-bit float

Compilation

- Any source file containing CUDA language extensions must be compiled with NVCC
- NVCC is a compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++,...
- NVCC outputs:
  - C code (host CPU Code)
    - Must then be compiled with the rest of the application using another tool
  - PTX
    - Object code directly
    - Or, PTX source, interpreted at runtime

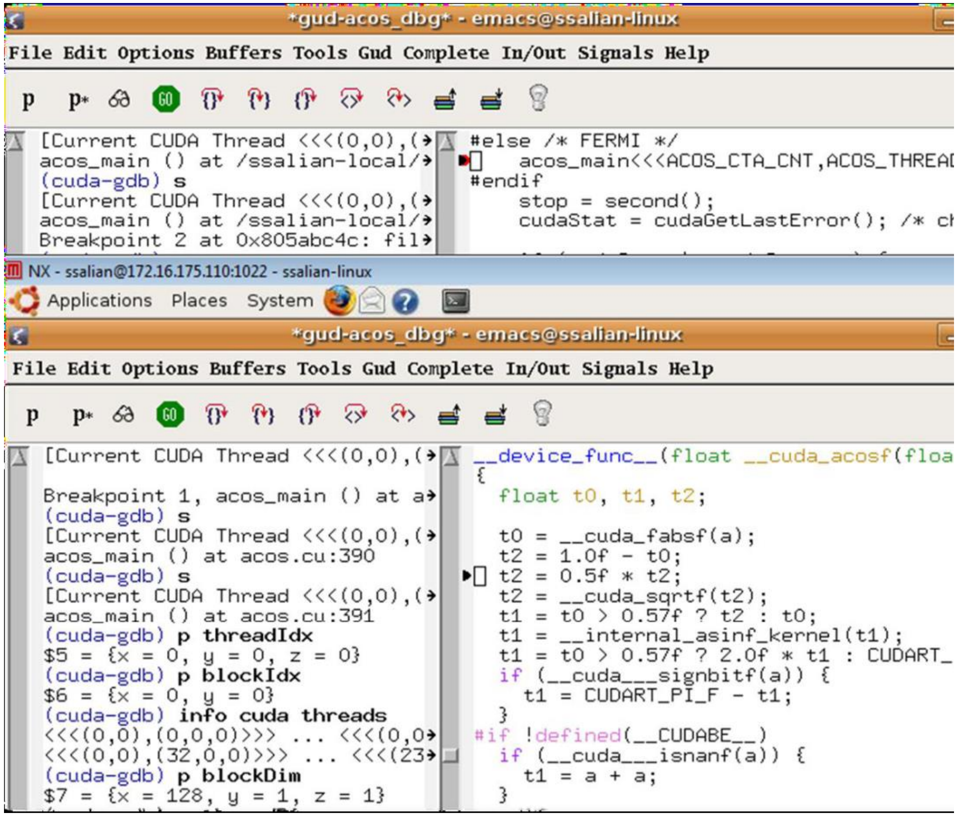
Compiling a CUDA Program



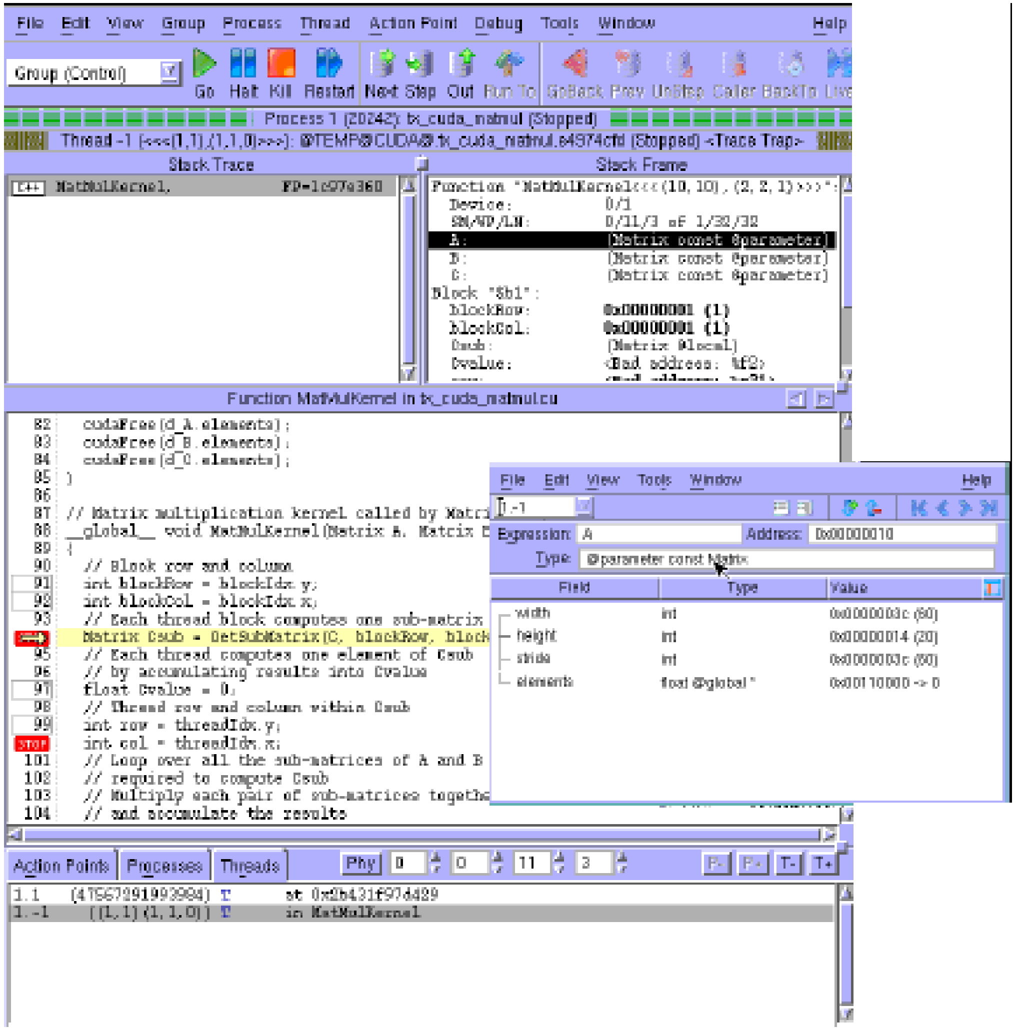
NVIDIA cuda-gdb

- CUDA debugging integrated into GDB on Linux
  - Supported on 32bit and 64bit systems
  - Seamlessly debug both the host/CPU and device/GPU code
  - Set break points on any source line or symbol name
  - Access and print all CUDA memory allocs, local, global, constant and shared vars

(Included in the CUDA Toolkit)



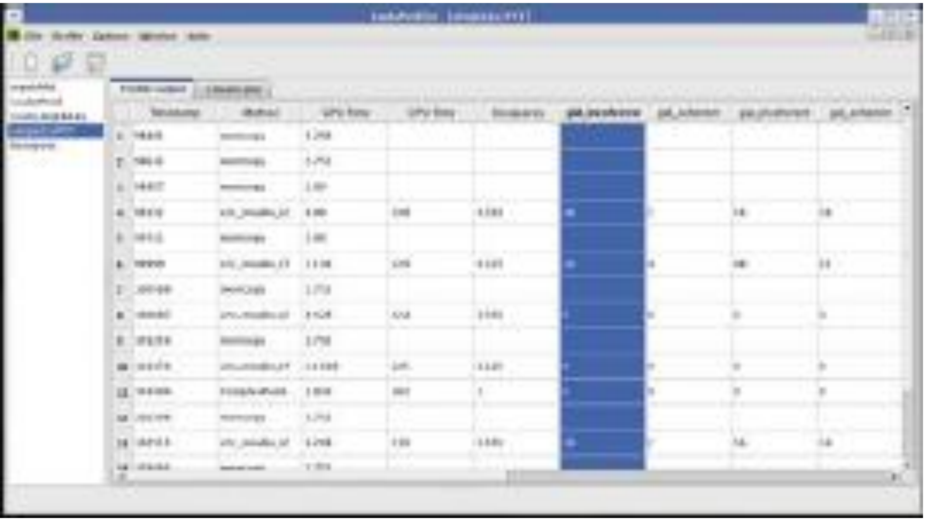
Allinea DDT debugger

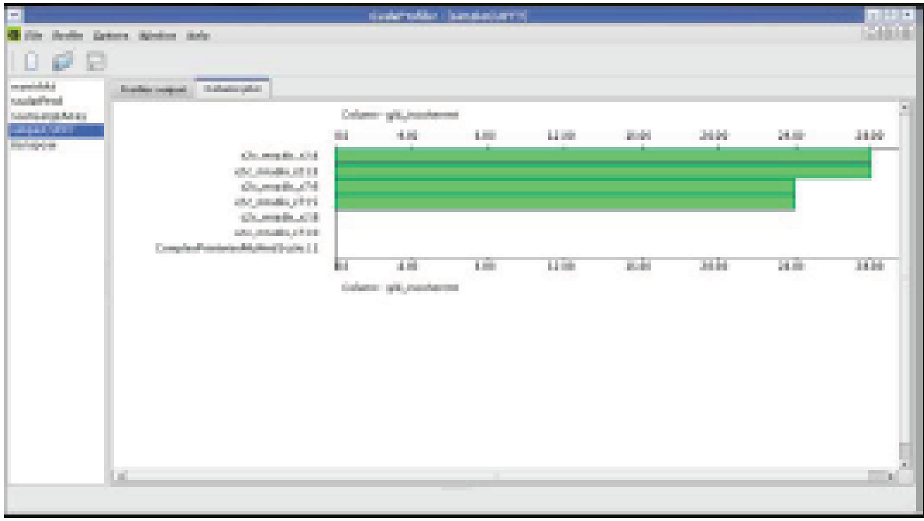


- CUDA SDK 3.0 with DDT 2.6
  - Released June 2010
  - Fermi and Tesla support
  - cuda-memcheck support for memory errors
  - Combined MPI and CUDA support
  - Stop on kernel launch feature
  - Kernel thread control, evaluation and breakpoints
  - Identify thread counts, ranges and CPU/GPU threads easily
- SDK 3.1 in beta with DDT 2.6.1
- SDK 3.2
  - Coming soon: multiple GPU device support

CUDA Profiler

- Motivation
  - Identify performance bottlenecks in multi-kernel applications
  - Quantify the benefit of optimizing a single kernel
- Access to hardware performance counters





Profiler

- Values represent events within a thread warp
- Only targets one multi-processor
  - Values will not correspond to the total number of warps launched for a particular kernel
  - Launch enough thread blocks to ensure the target multi-processor is given a consistent percentage of the total work
- Values are best used to identify the relative performance differences between unoptimized and optimized codes
  - Try to reduce the magnitudes of gld/gst_incoherent, divergent_branch and warp serialize

Profiler

- Measures
  - Kernel execution
    - branch, divergent_branch, warp_serialize, instructions, cta_launched…
  - Memory transfer
    - gld_incoherent, gld_coherent, gst_incoherent, gst_coherent, local load, local store…
- Profiler counters: 4

CUDA Profiler

- Example

timestamp=[2155.302] method=[_Z10fhaar1dwtdiPf]
gputime=[7.808] cputime=[74.730] occupancy=[1.000]

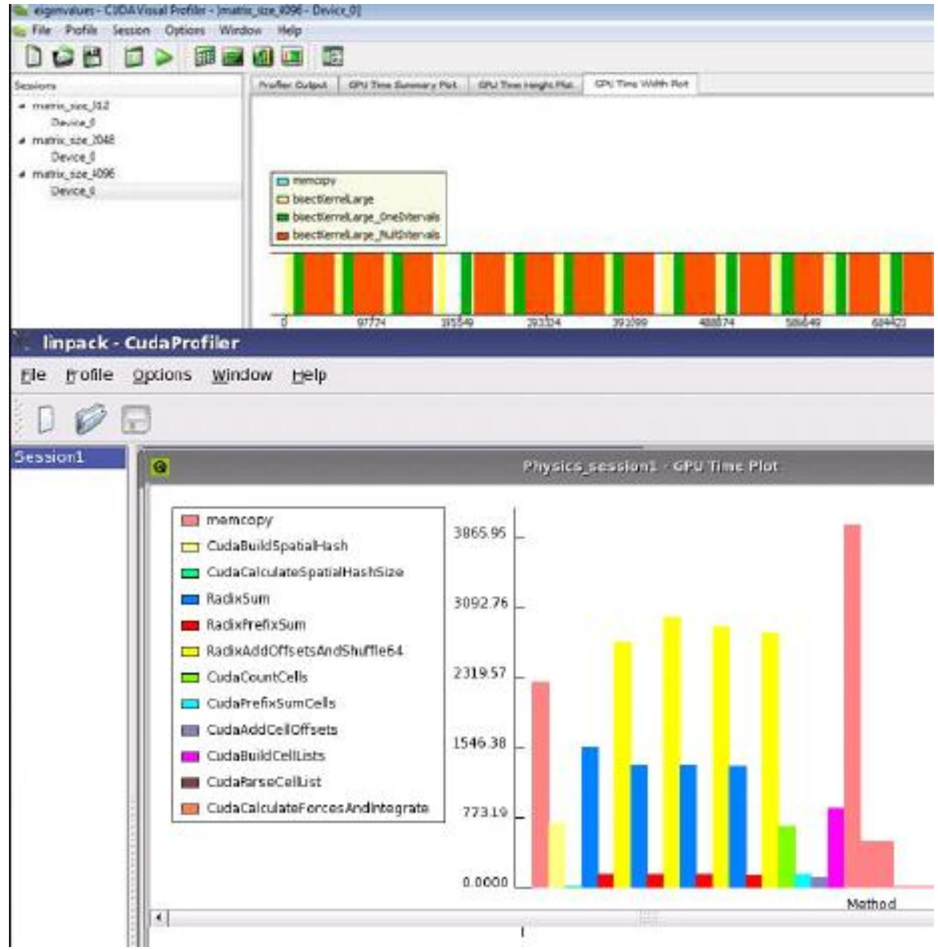
timestamp=[2421.886] method=[memcopy] gputime=[4.864]
cputime=[238.159]

timestamp=[2706.140] method=[_Z10ihaar1dwtdiPf]
gputime=[7.296] cputime=[59.295] occupancy=[1.000]

timestamp=[2876.413] method=[memcopy] gputime=[4.608]
cputime=[224.679]

NVIDIA Visual Profiler

- Analyze GPU HW performance signals, kernel occupancy, instruction throughput, and more
- Highly configurable tables and graphical views
- Save/load profiler sessions or export to CSV for later analysis
- Compare results visually across multiple sessions to see improvements
- Windows, Linux and Mac OS X OpenCL support on Windows and Linux

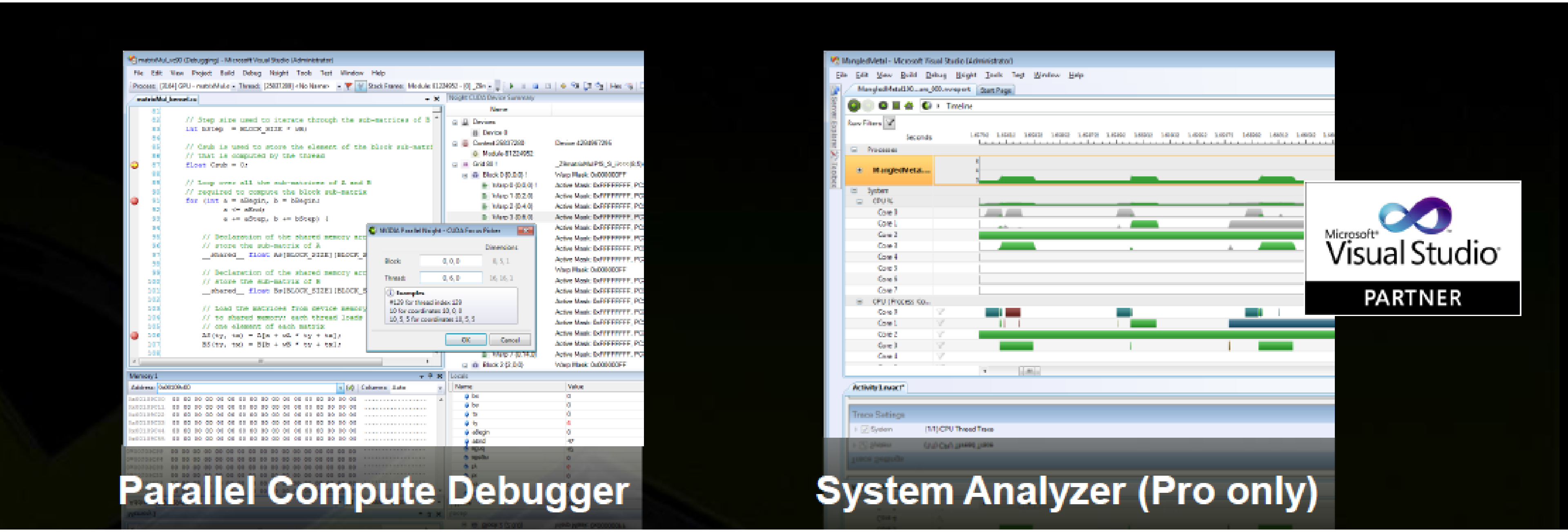


(Included in the CUDA Toolkit)

Parallel Nsight Visual Studio Edition

- Development environment for heterogeneous platform (GPU+CPU)
- Compile your code with Debug flag
- Use the Visual Studio interface to debug your GPU code
  - Explore memory during a live session
  - Immediately view live variables
  - Set data breakpoints on memory area
  - …

Parallel Nsight Visual Studio Edition



Time Function

clock_t clock( )

Return value: per-multiprocessor counter incremented every clock cycle

- Sample at the beginning and the end of a kernel
- Take the difference of the two samples
- Record the result

- Time by clock() is larger than the time actually spent on executing thread instructions

Time Function

- CUDA Event
  - An accurate timing
  - Implementation: asynchronously record *event* at any point in a program

Creation of CUDA Event:

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
```

Time Function

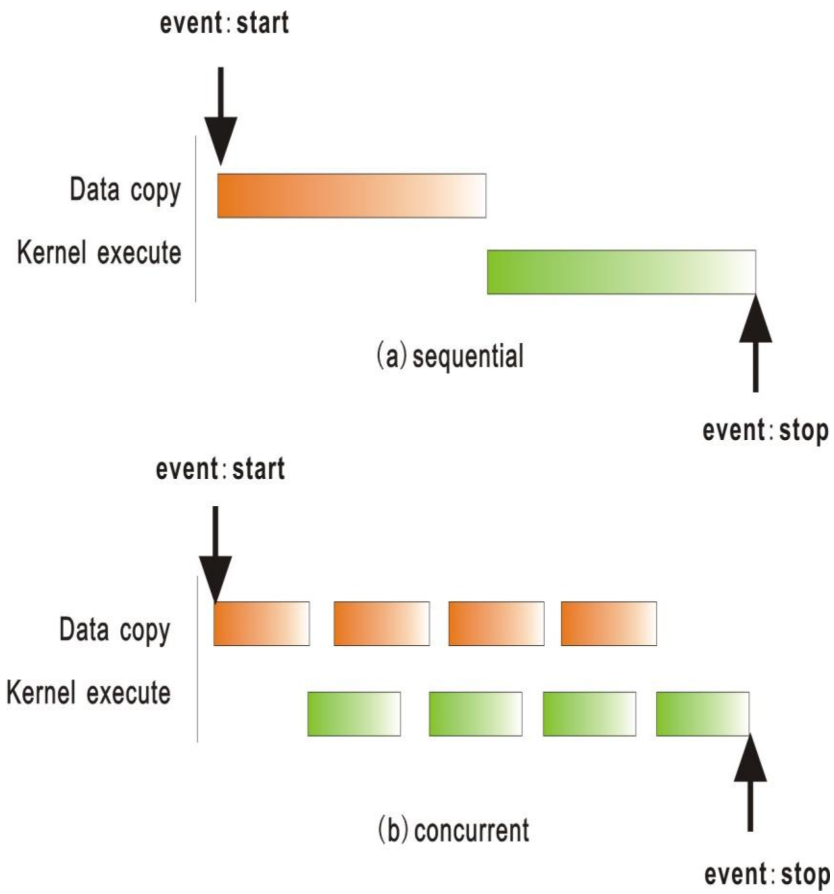
Timing using CUDA Event:

```
cudaEventRecord(start, 0);
myKernel<<<100, 512>>>(outputDev, inputDev);
…
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
```

Destruction of CUDA Event:

```
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Time Function




```
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start,0);
cudaMemcpyAsync(,,,stream[i]);
kernel<<<,,,stream[i]>>>()
cudaMemcpyAsync(,,,stream[i]);
cudaEventRecord(stop,0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsed
    Time, start, stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```