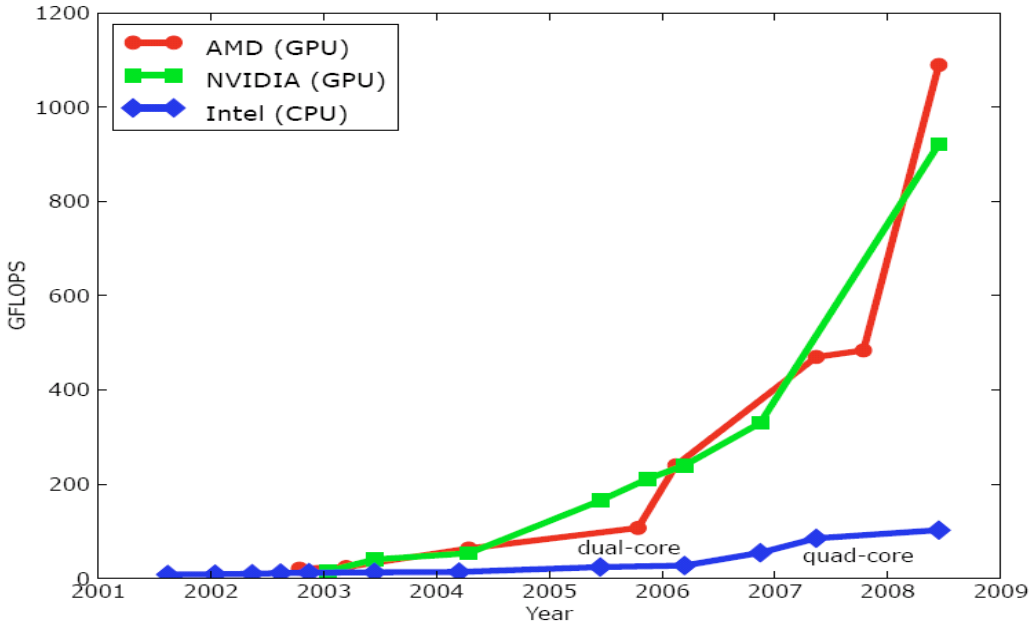
Programming Massively Parallel Processors

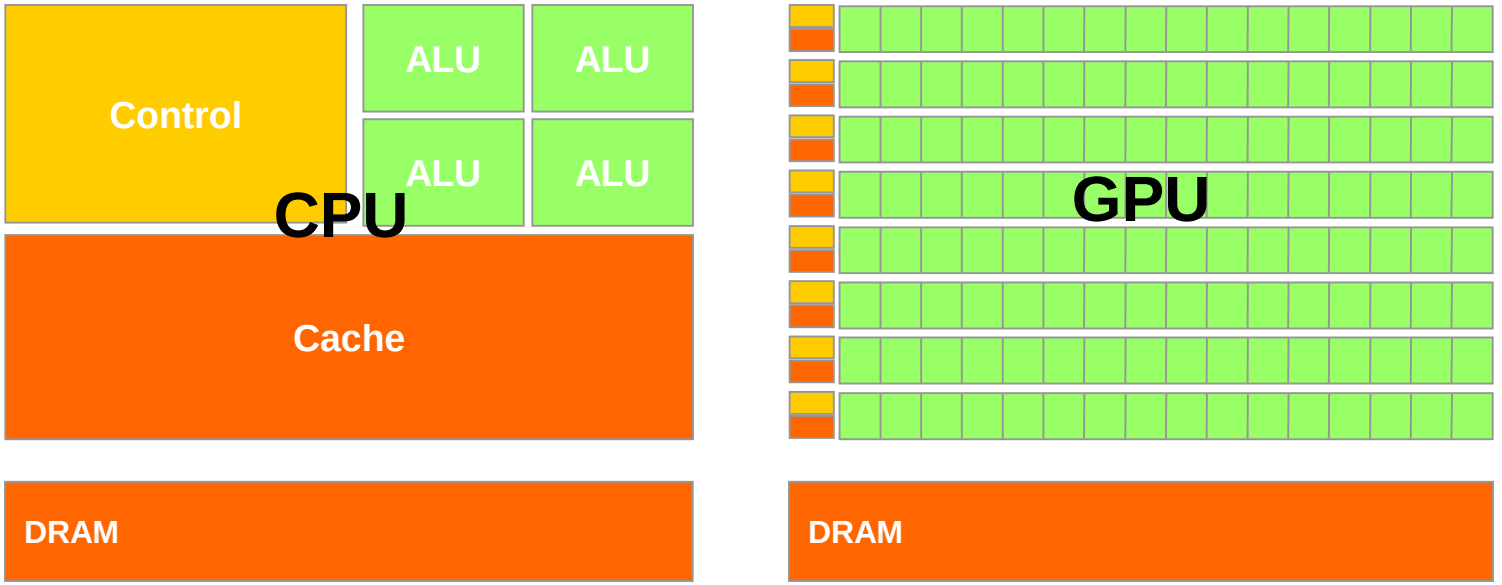
Why Massively Parallel Processor

- A quiet revolution and potential build-up
  - Calculation: 367 GFLOPS vs. 32 GFLOPS
  - Memory Bandwidth: 86.4 GB/s vs. 8.4 GB/s
  - Until last year, programmed through graphics API

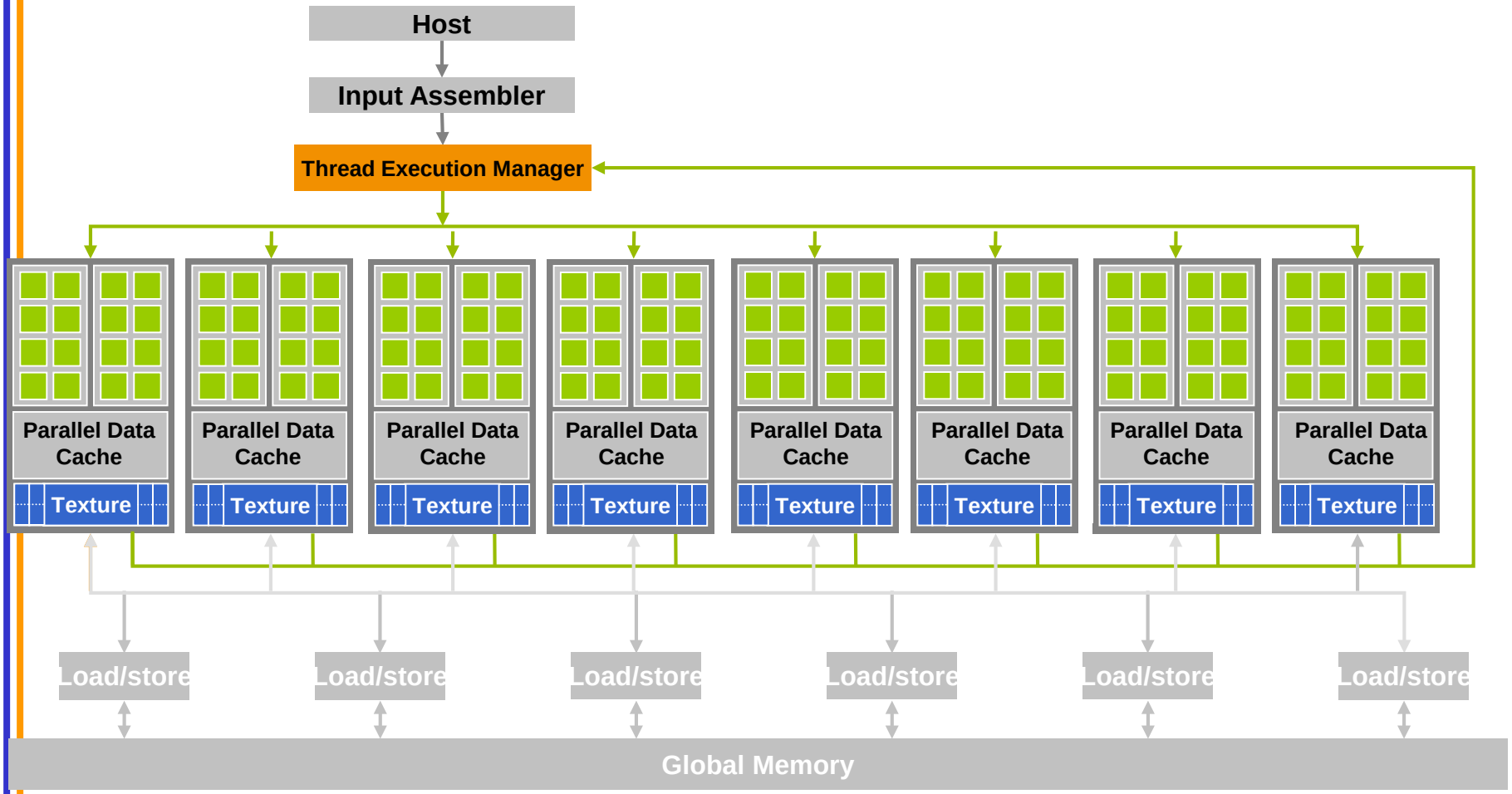


  - GPU in every PC and workstation – massive volume and potential impact

CPUs and GPUs have fundamentally different design philosophies



Architecture of a CUDA-capable GPU



GT200 Characteristics

- 1 TFLOPS peak performance (25-50 times of current high-end microprocessors)
- 265 GFLOPS sustained for apps such as VMD
- Massively parallel, 128 cores, 90W
- Massively threaded, sustains 1000s of threads per app
- 30-100 times speedup over high-end microprocessors on scientific and media applications: medical imaging, molecular dynamics

"I think they're right on the money, but the huge performance differential (currently 3 GPUs ~= 300 SGI Altix Itanium2s) will invite close scrutiny so I have to be careful what I say publically until I triple check those numbers."

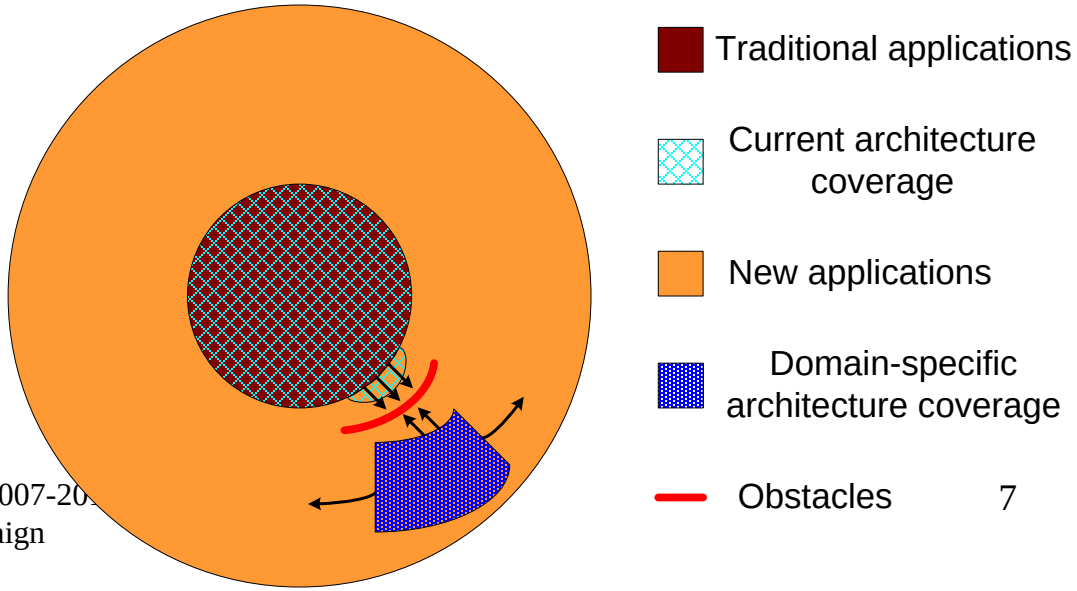
-John Stone, VMD group, Physics UIUC

Future Apps Reflect a Concurrent World

- Exciting applications in future mass computing market have been traditionally considered "supercomputing applications"
  - Molecular dynamics simulation, Video and audio coding and manipulation, 3D imaging and visualization, Consumer game physics, and virtual reality products
  - These "Super-apps" represent and model physical, concurrent world
- Various granularities of parallelism exist, but…
  - programming model must not hinder parallel implementation
  - data delivery needs careful management

Stretching Traditional Architectures

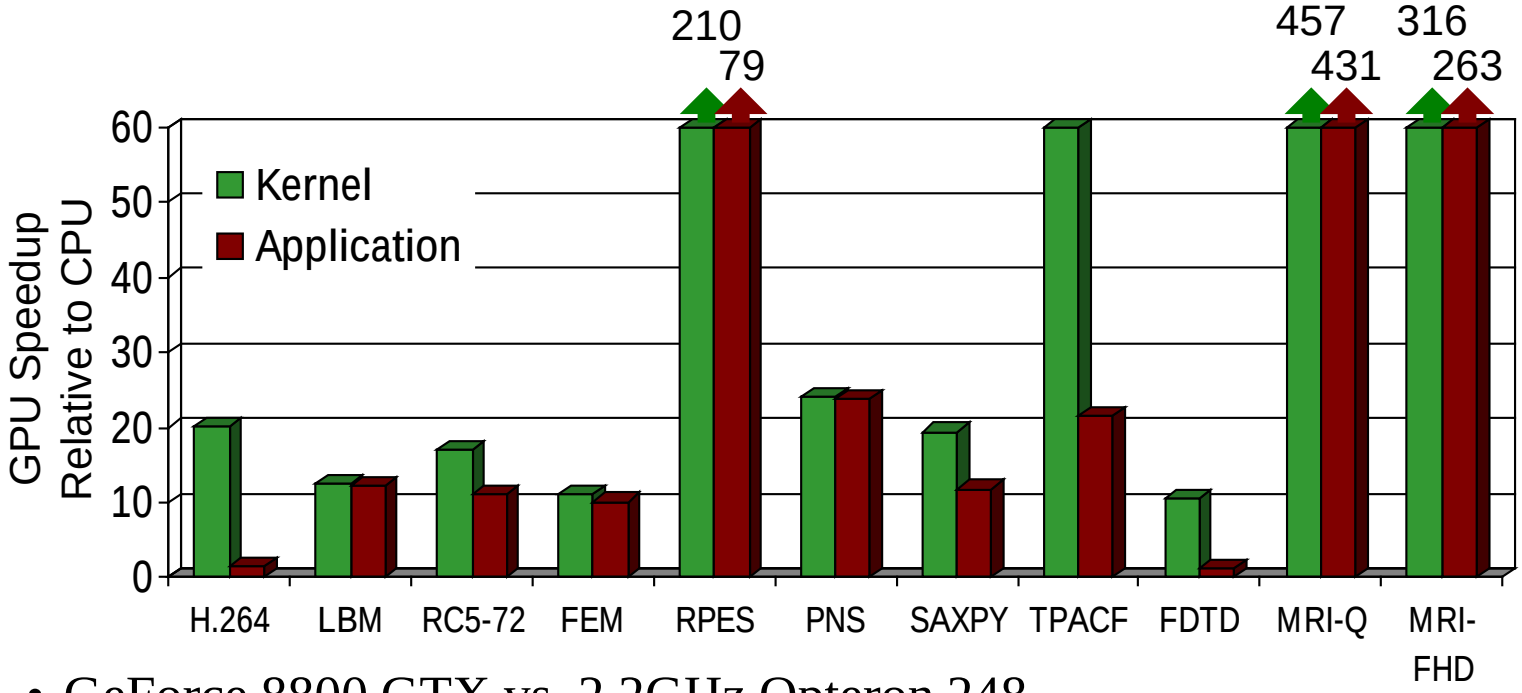
- Traditional parallel architectures cover some super-applications
  - DSP, GPU, network apps, Scientific
- The game is to grow mainstream architectures "out" or domain-specific architectures "in"
  - CUDA is latter



Samples of Previous Projects

Application	Description	Source	Kernel	% time
H.264	SPEC ‘06 version, change in guess vector	34,811	194	35%
LBM	SPEC ‘06 version, change to single precision and print fewer reports	1,481	285	>99%
RC5-72	Distributed.net RC5-72 challenge client code	1,979	218	>99%
FEM	Finite element modeling, simulation of 3D graded materials	1,874	146	99%
RPES	Rye Polynomial Equation Solver, quantum chem, 2-electron repulsion	1,104	281	99%
PNS	Petri Net simulation of a distributed system	322	160	>99%
SAXPY	Single-precision implementation of saxpy, used in Linpack’s Gaussian elim. routine	952	31	>99%
TRACF	Two Point Angular Correlation Function	536	98	96%
FDTD	Finite-Difference Time Domain analysis of 2D electromagnetic wave propagation	1,365	93	16%
MRI-Q	Computing a matrix Q, a scanner’s configuration in MRI reconstruction	490	33	>99%

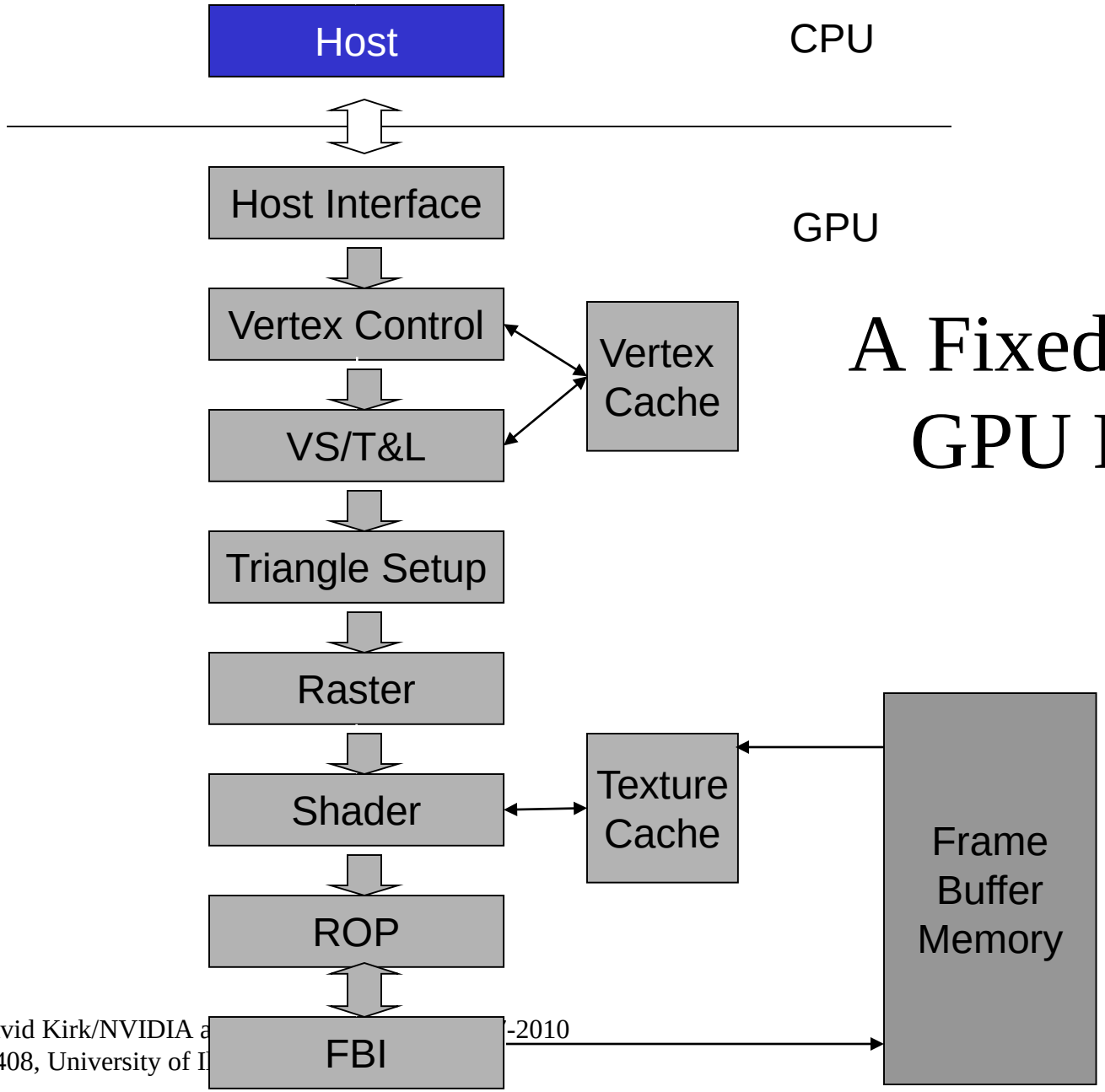
Speedup of Applications



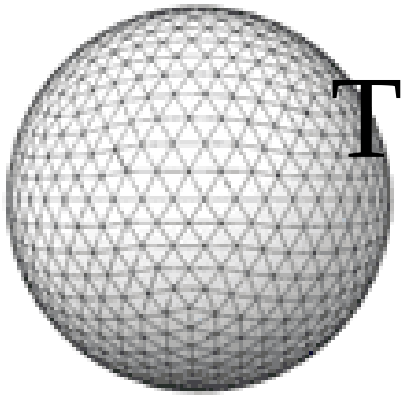
- GeForce 8800 GTX vs. 2.2GHz Opteron 248
- 10× speedup in a kernel is typical, as long as the kernel can occupy enough parallel threads
- 25× to 400× speedup if the function's data requirements and control flow suit the GPU and the application is optimized

GPU HISTORY

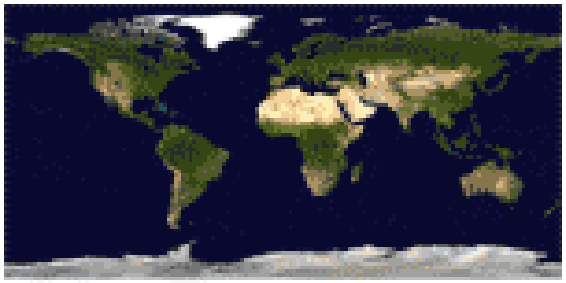
A Fixed Function GPU Pipeline



Texture Mapping Example

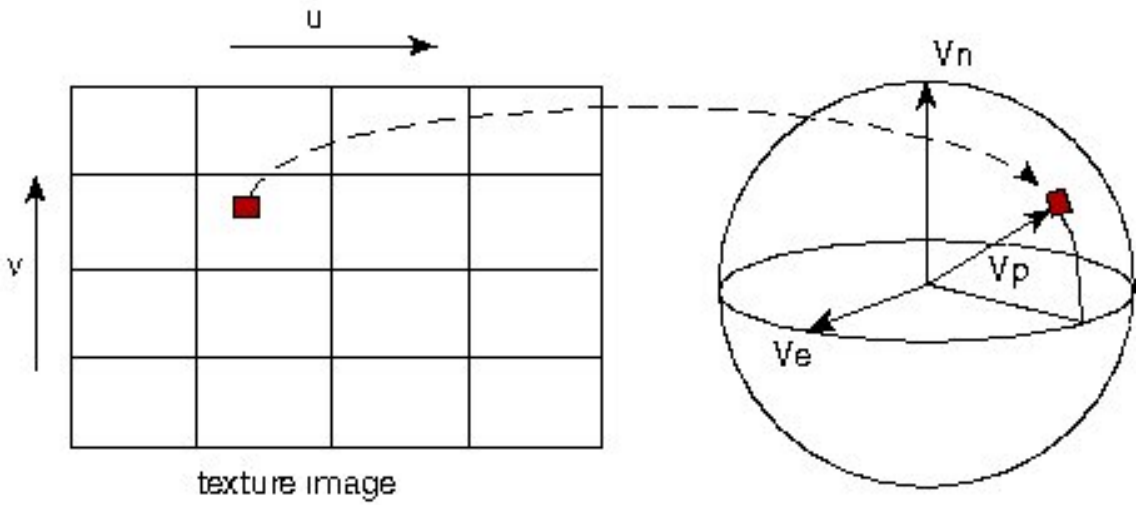


Sphere with no texture



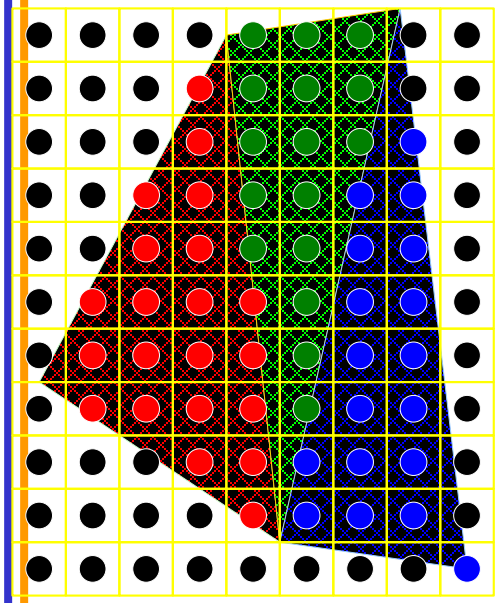
Texture image

Sphere with texture

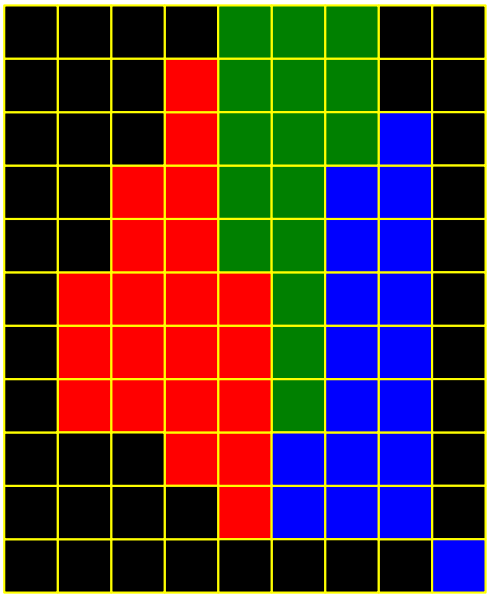


Texture mapping example: painting a world map texture image onto a globe object.

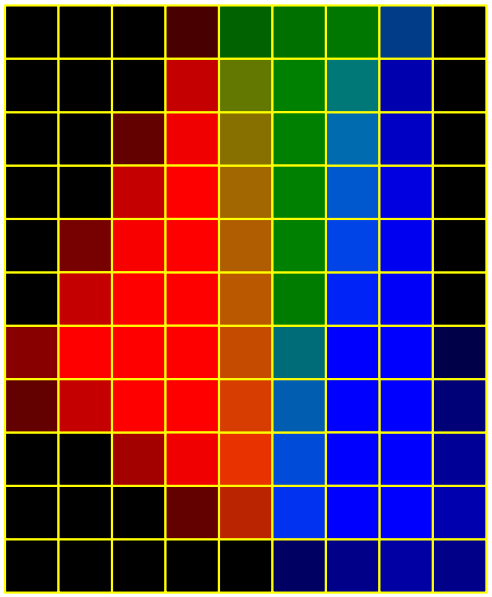
Anti-Aliasing Example



Triangle Geometry

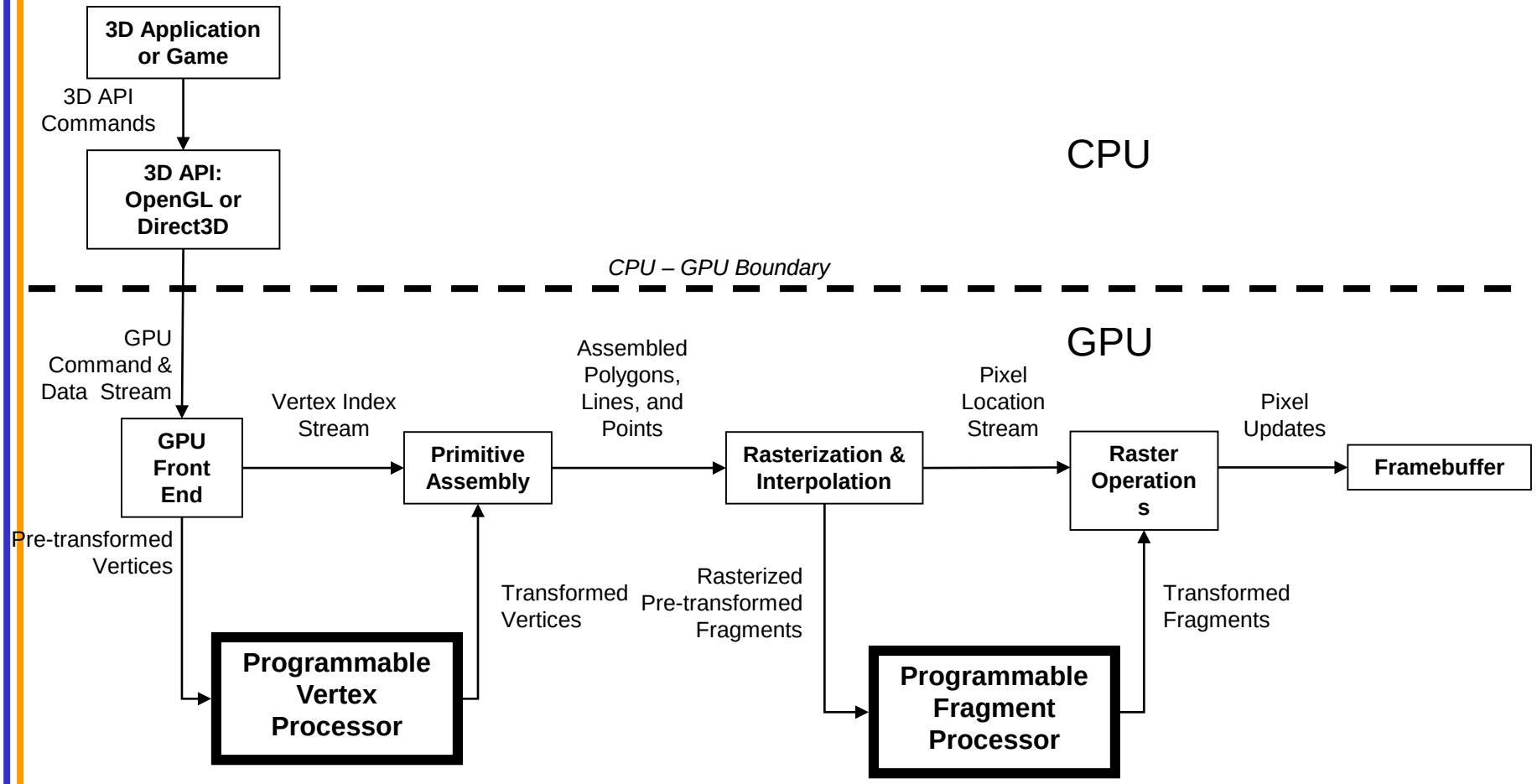


Aliased



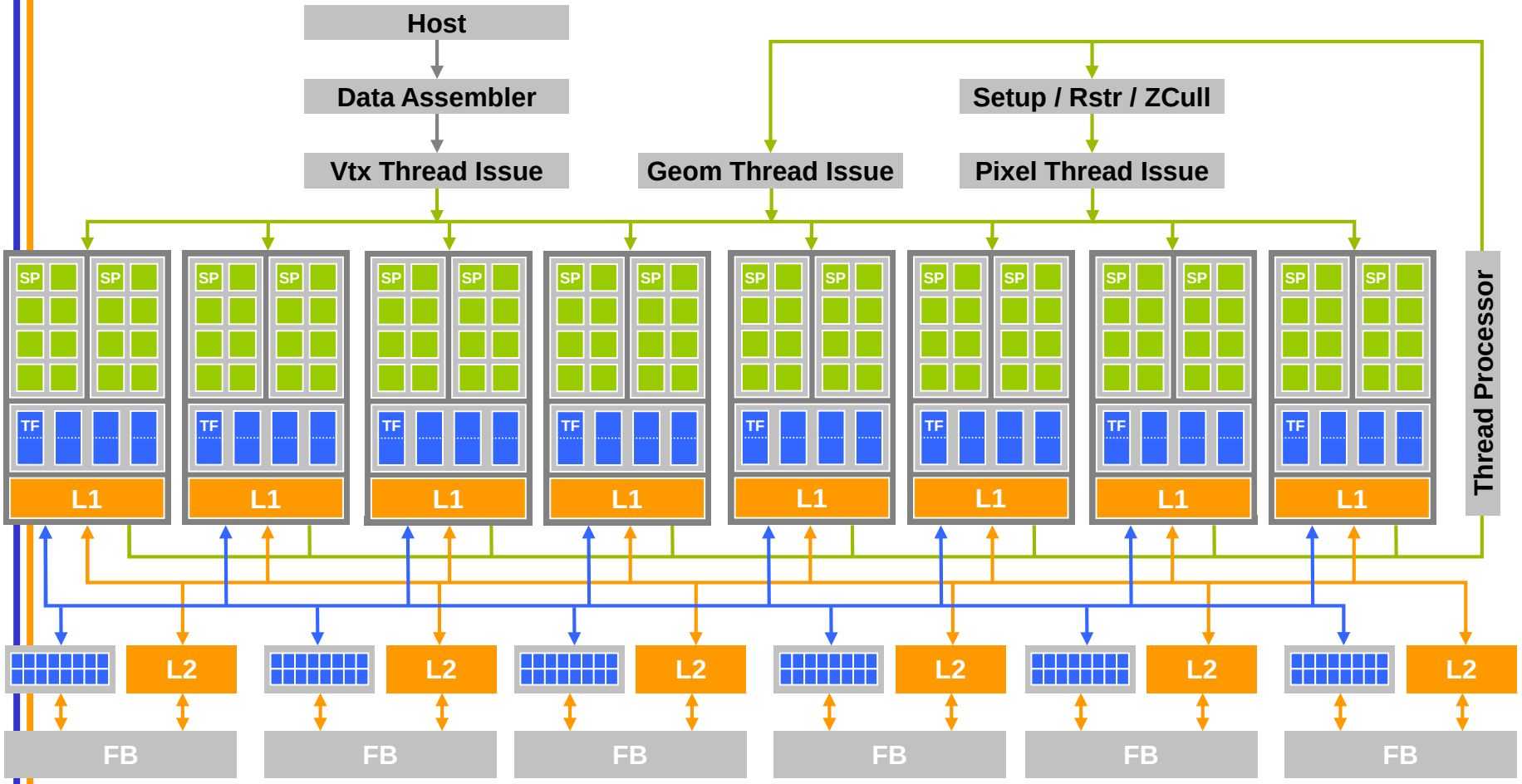
Anti-Aliased

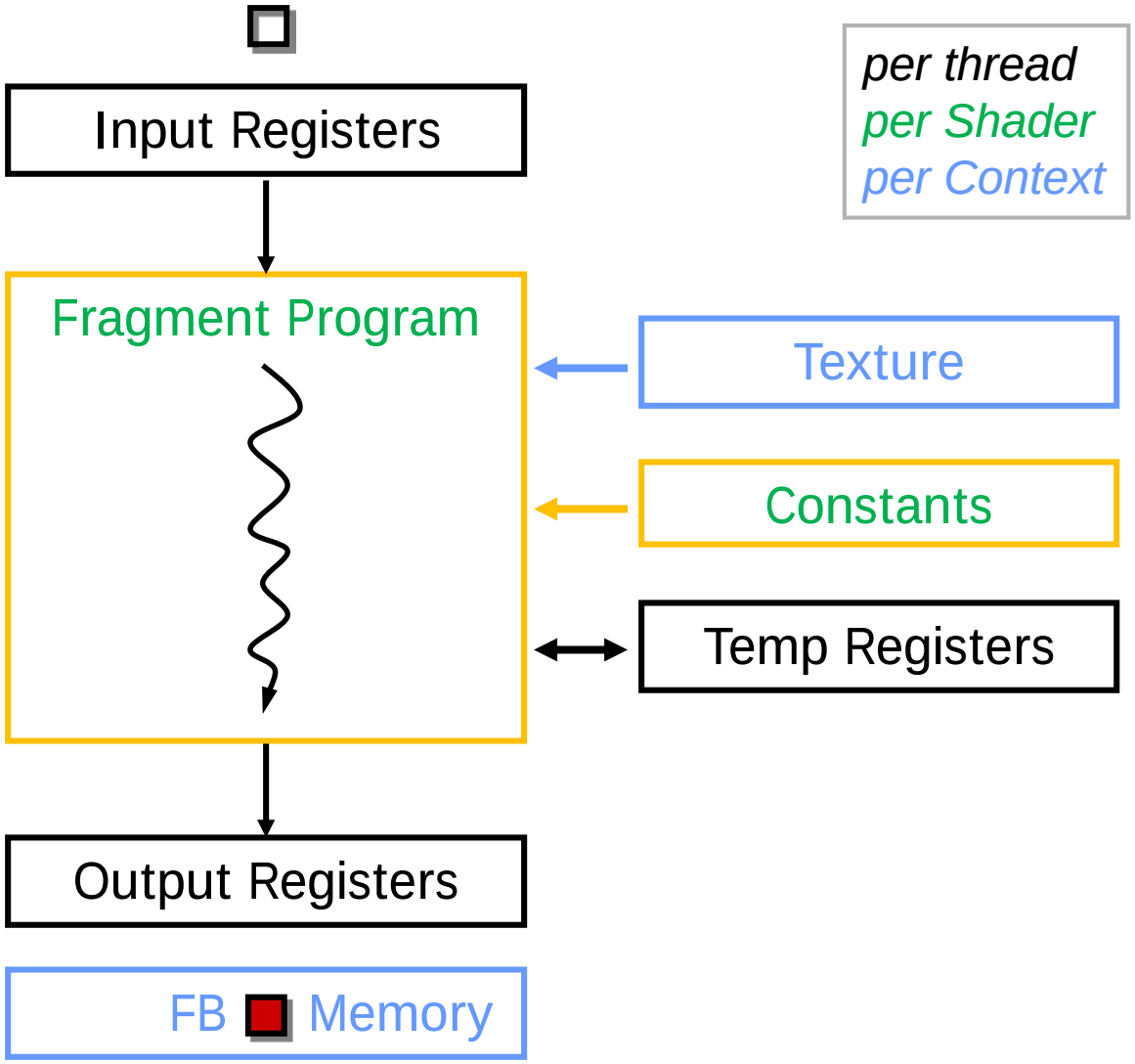
Programmable Vertex and Pixel Processors



An example of separate vertex processor and fragment processor in a programmable graphics pipeline

Unified Graphics Pipeline





The restricted input and output capabilities of a shader programming model.

CUDA PROGRAMMING MODEL

What is (Historical) GPGPU ?

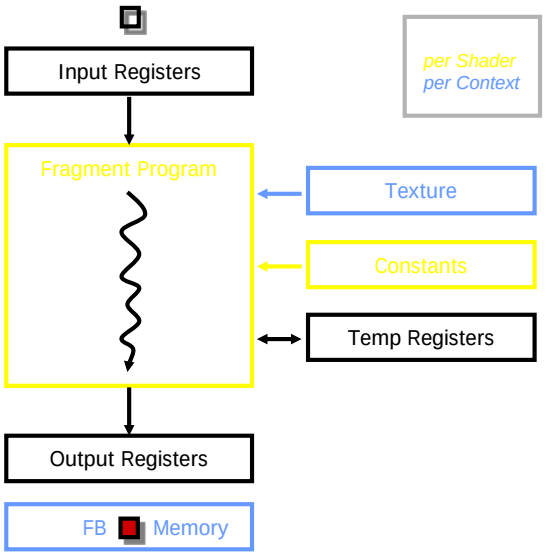
- General Purpose computation using GPU and graphics API in applications other than 3D graphics
  - GPU accelerates critical path of application

- Data parallel algorithms leverage GPU attributes
  - Large data arrays, streaming throughput
  - Fine-grain SIMD parallelism
  - Low-latency floating point (FP) computation

- Applications – see //GPGPU.org
  - Game effects (FX) physics, image processing
  - Physical modeling, computational engineering, matrix algebra, convolution, correlation, sorting

Previous GPGPU Constraints

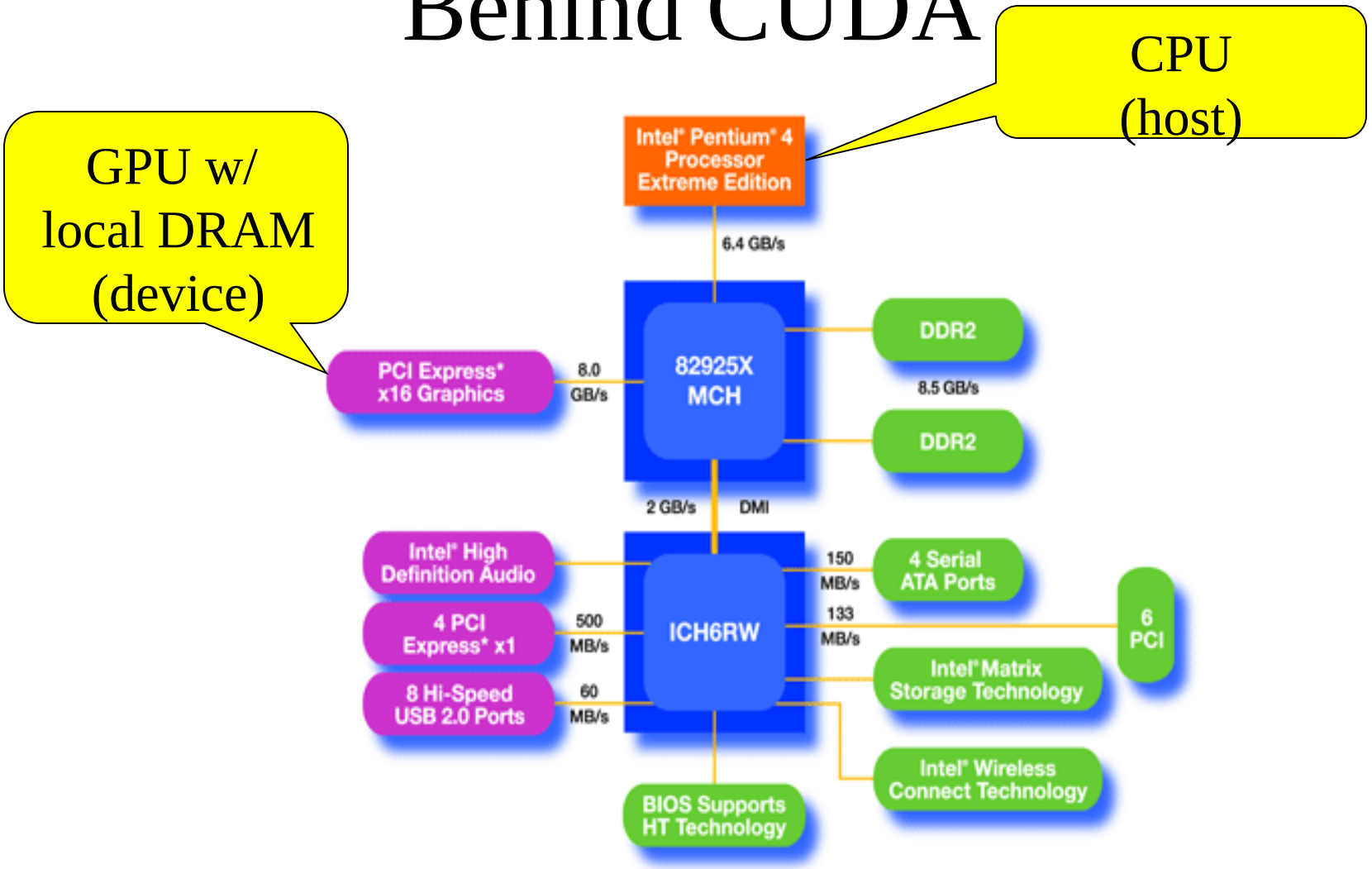
- Dealing with graphics API
  - Working with the corner cases of the graphics API
- Addressing modes
  - Limited texture size/dimension
- Shader capabilities
  - Limited outputs
- Instruction sets
  - Lack of Integer & bit ops
- Communication limited
  - Between pixels
  - Scatter a[i] = p



CUDA

- "Compute Unified Device Architecture"
- General purpose programming model
  - User kicks off batches of threads on the GPU
  - GPU = dedicated super-threaded, massively data parallel co-processor
- Targeted software stack
  - Compute oriented drivers, language, and tools
- Driver for loading computation programs into GPU
  - Standalone Driver - Optimized for computation
  - Interface designed for compute – graphics-free API
  - Data sharing with OpenGL buffer objects
  - Guaranteed maximum download & readback speeds
  - Explicit GPU memory management

An Example of Physical Reality Behind CUDA



Parallel Computing on a GPU

- 8-series GPUs deliver 25 to 200+ GFLOPS on compiled parallel C applications
  - Available in laptops, desktops, and clusters

GeForce 8800

- GPU parallelism is doubling every year
- Programming model scales transparently

Tesla D870

- Programmable in C with CUDA tools
- Multithreaded SPMD model uses application data parallelism and thread parallelism

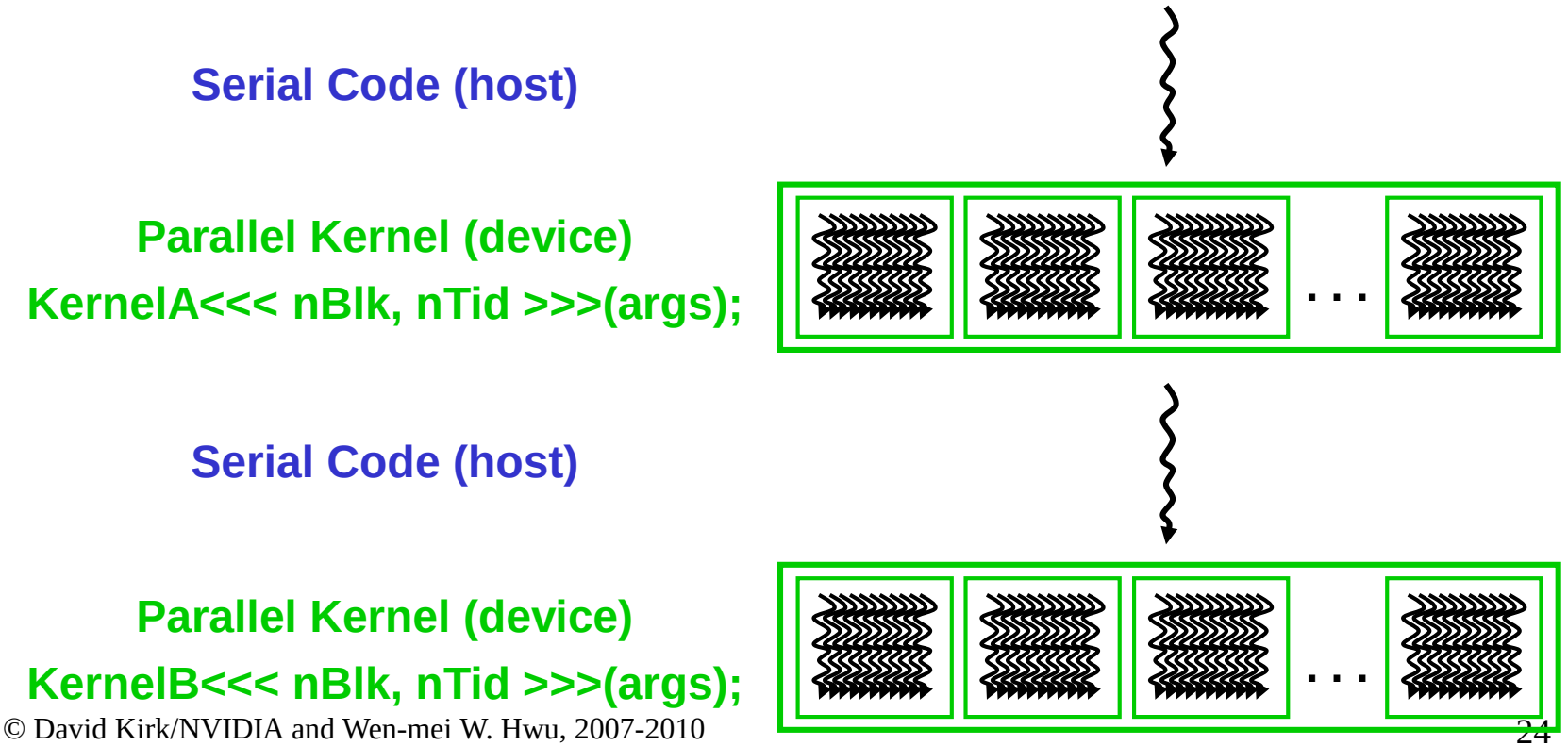
Tesla S870

Overview

- CUDA programming model – basic concepts and data types
- CUDA application programming interface - basic
- Simple examples to illustrate basic concepts and functionalities
- Performance features will be covered later

CUDA – C with no shader limitations!

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

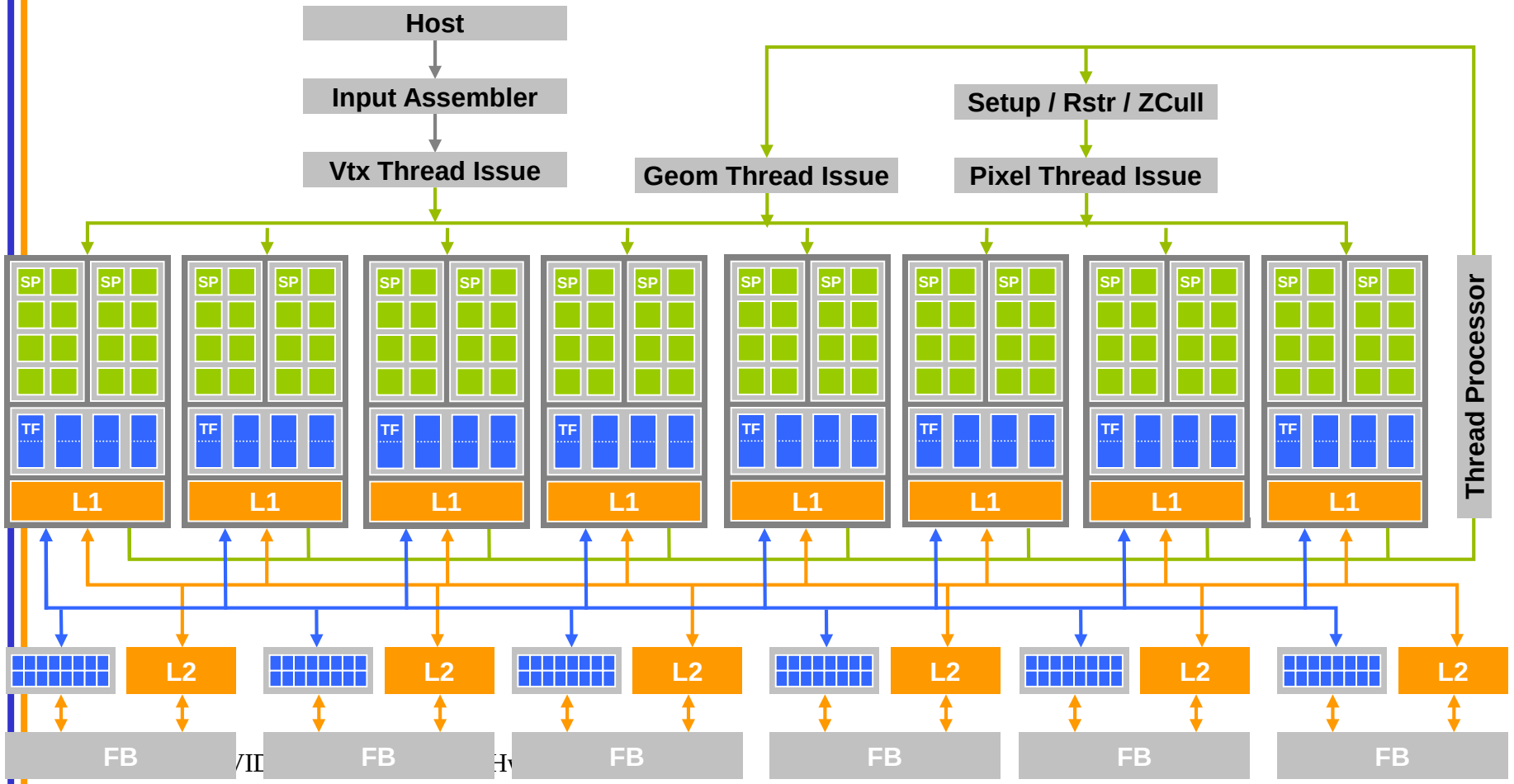


CUDA Devices and Threads

- 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
- Data-parallel portions of an application are expressed as device kernels which run on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

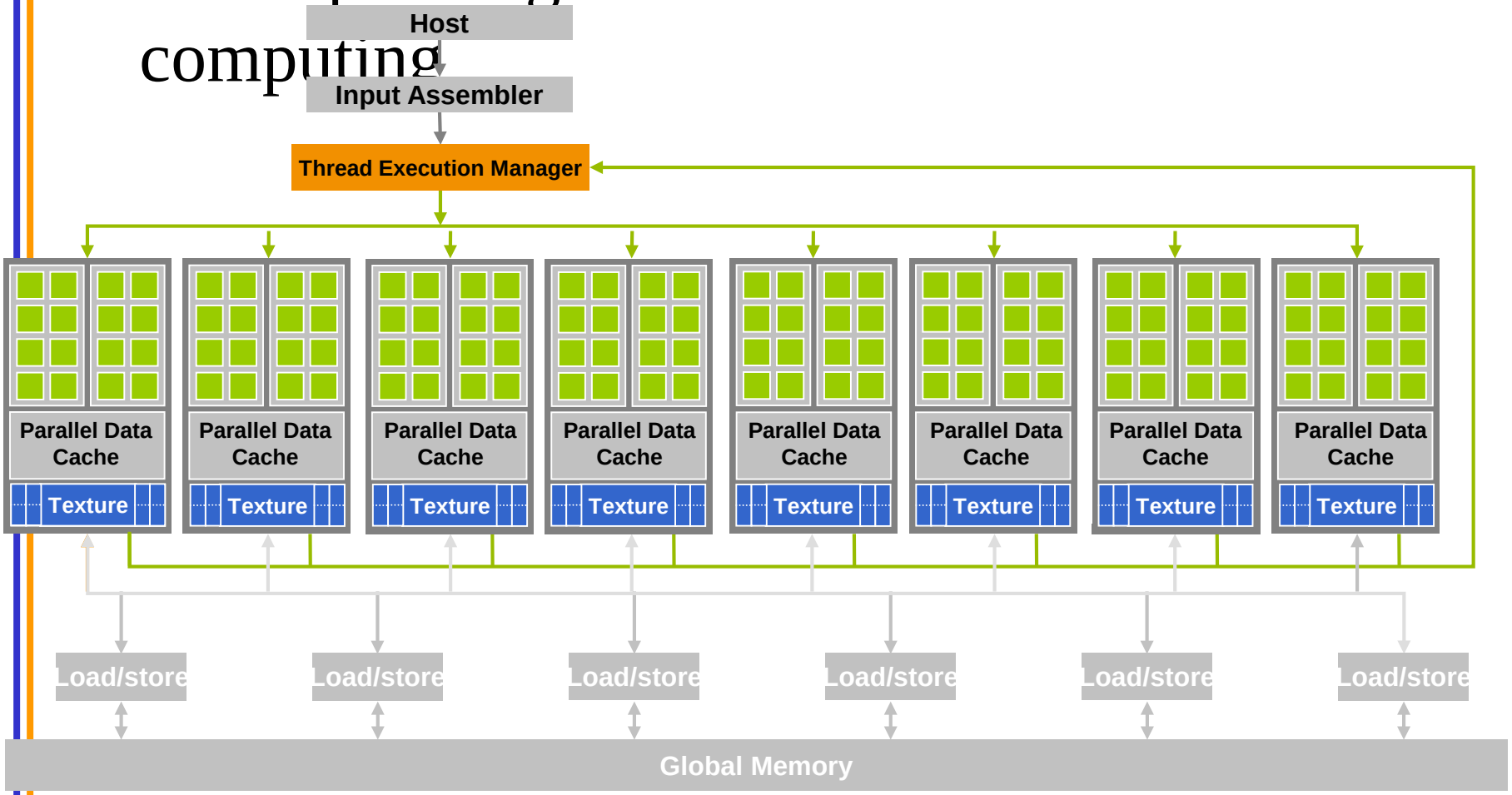
G80 – Graphics Mode

- The future of GPUs is programmable processing
- So – build the architecture around the processor



G80 CUDA mode – A **Device** Example

- Processors execute computing threads
- New operating mode/HW interface for computing

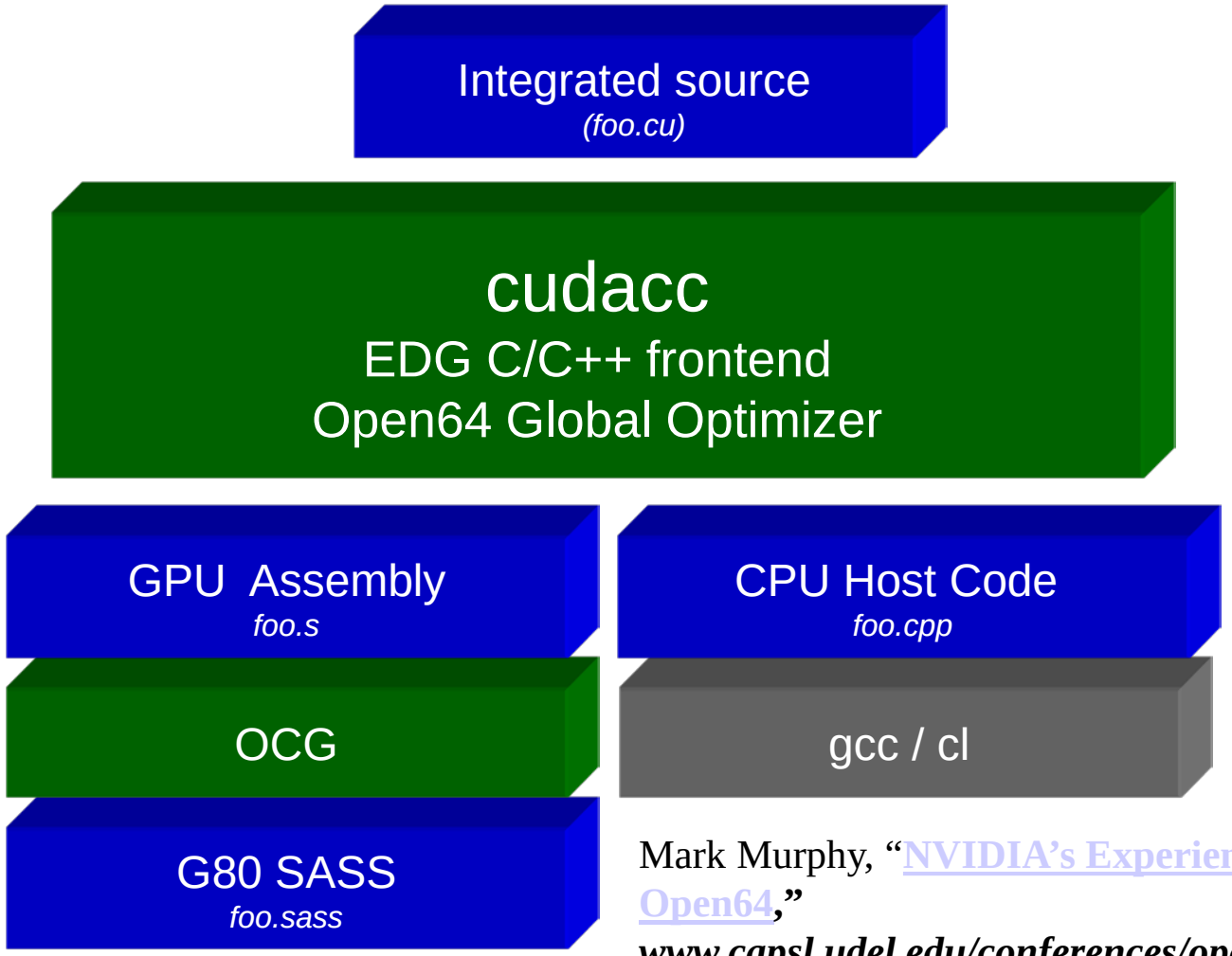


Extended C

- **Declspecs**
  - **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);
```

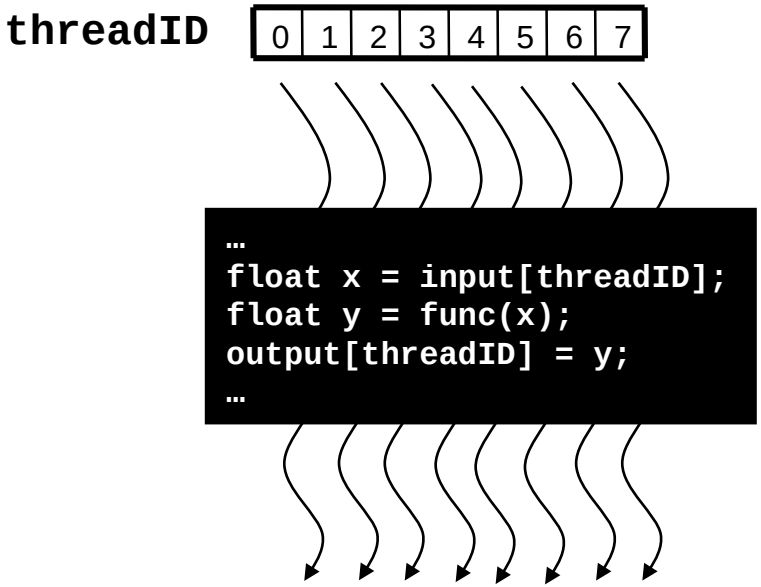
Extended C



Mark Murphy, "NVIDIA's Experience with Open64,"
www.capsl.udel.edu/conferences/open64/2008/Papers/101.doc

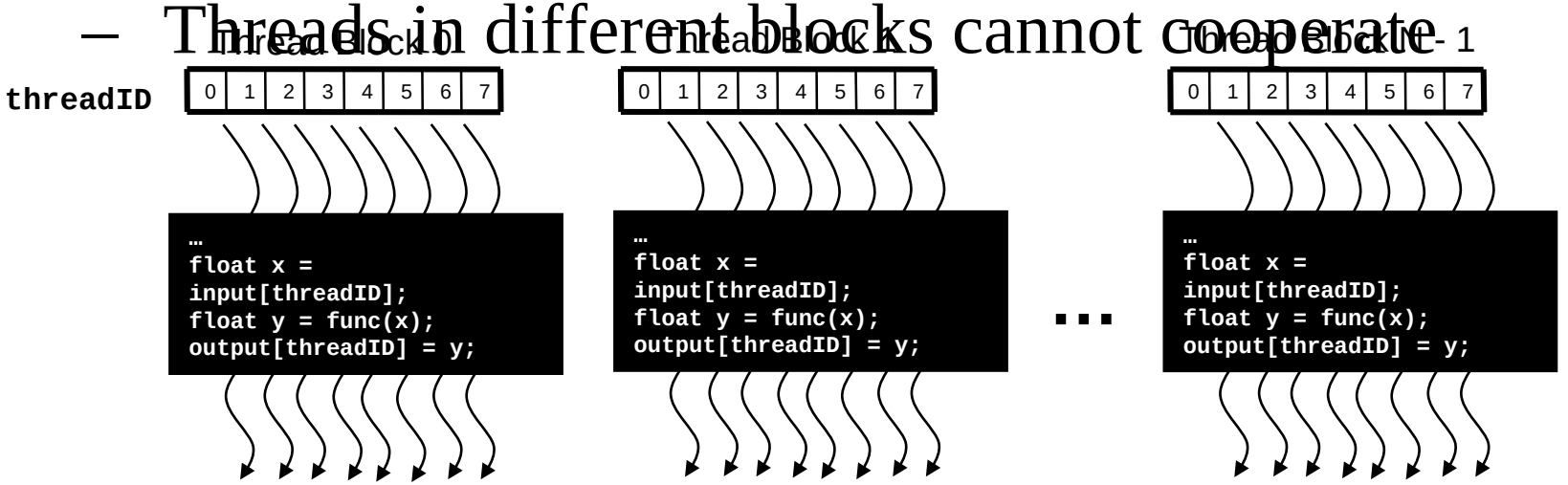
Arrays of Parallel Threads

- 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



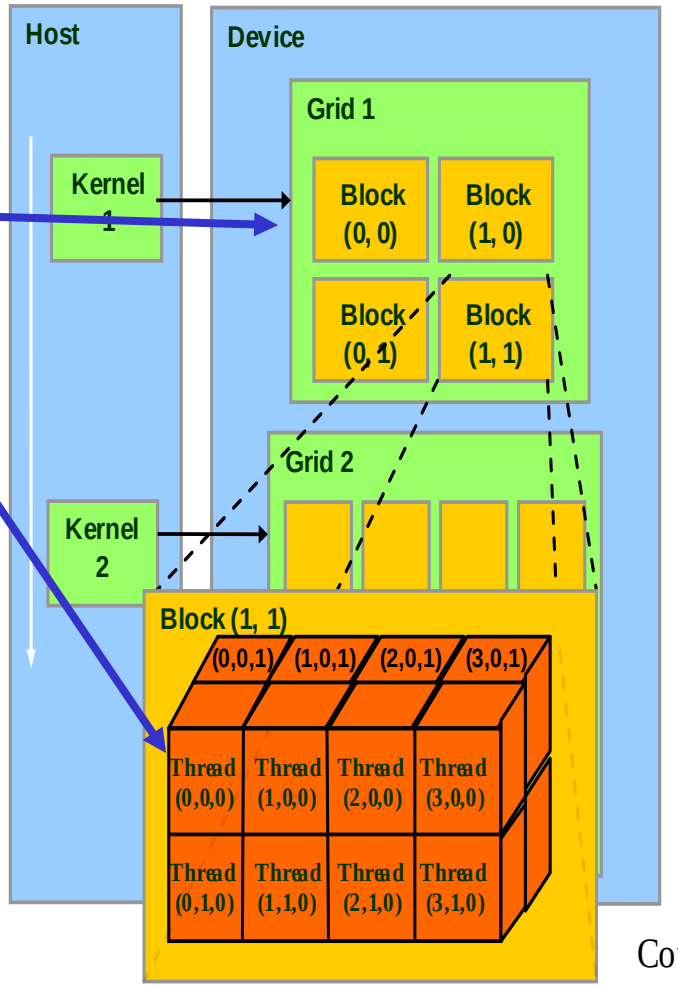
Thread Blocks: Scalable Cooperation

- 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



Block IDs and Thread IDs

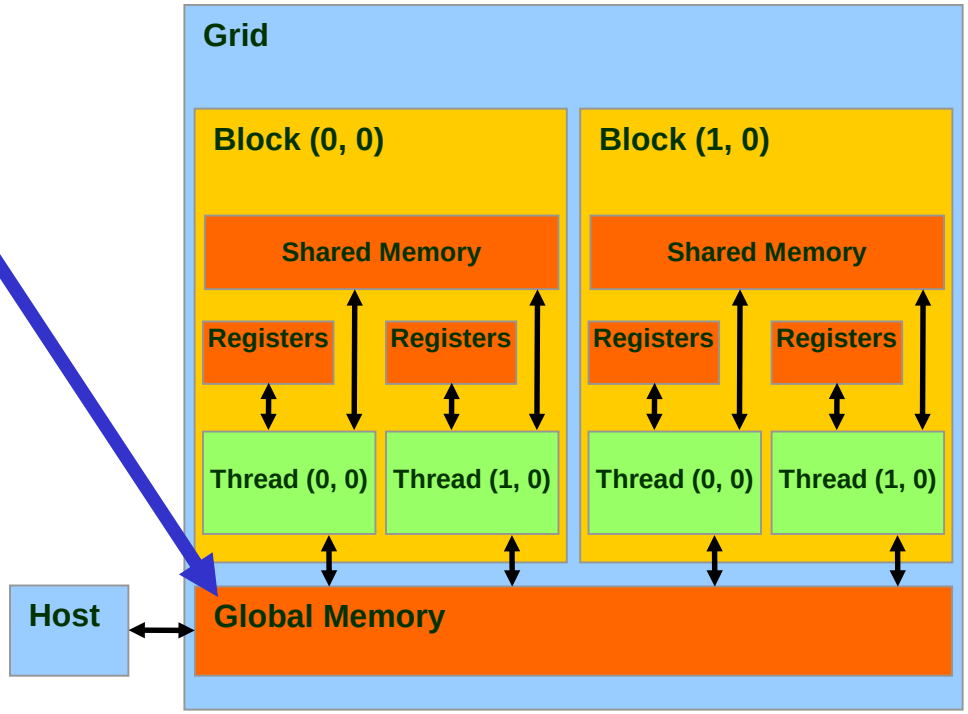
- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



Courtesy: NDVIA

CUDA Memory Model Overview

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
  - Long latency access
- We will focus on global memory for now
  - Constant and texture memory will come later



CUDA API Highlights: Easy and Lightweight

- The API is an extension to the ANSI C programming language

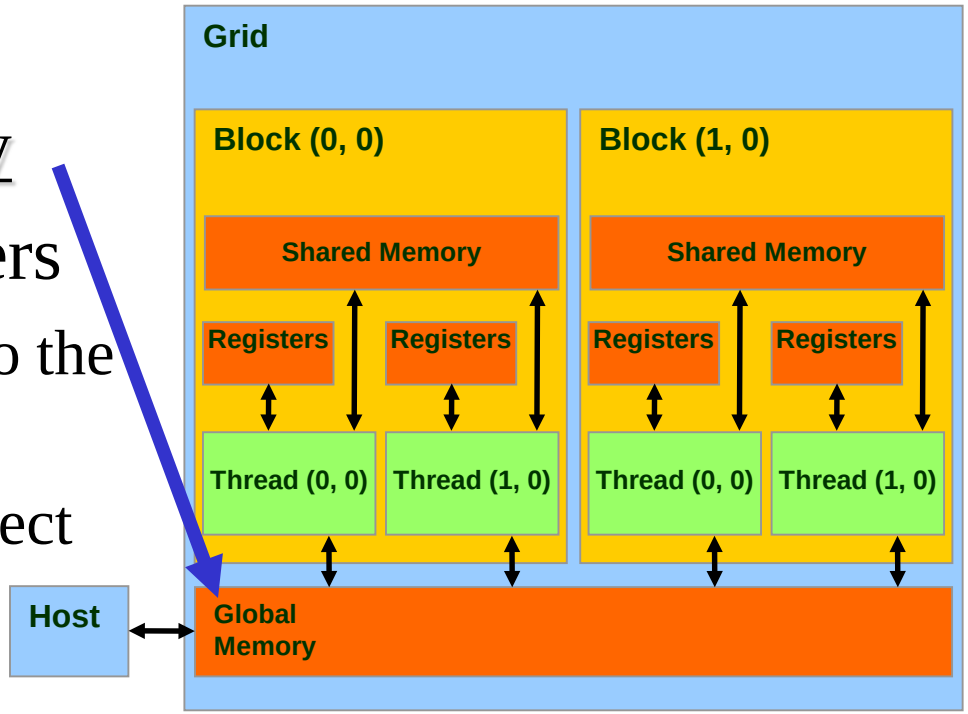
  → Low learning curve

- The hardware is designed to enable lightweight runtime and driver

  → High performance

CUDA Device Memory Allocation

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



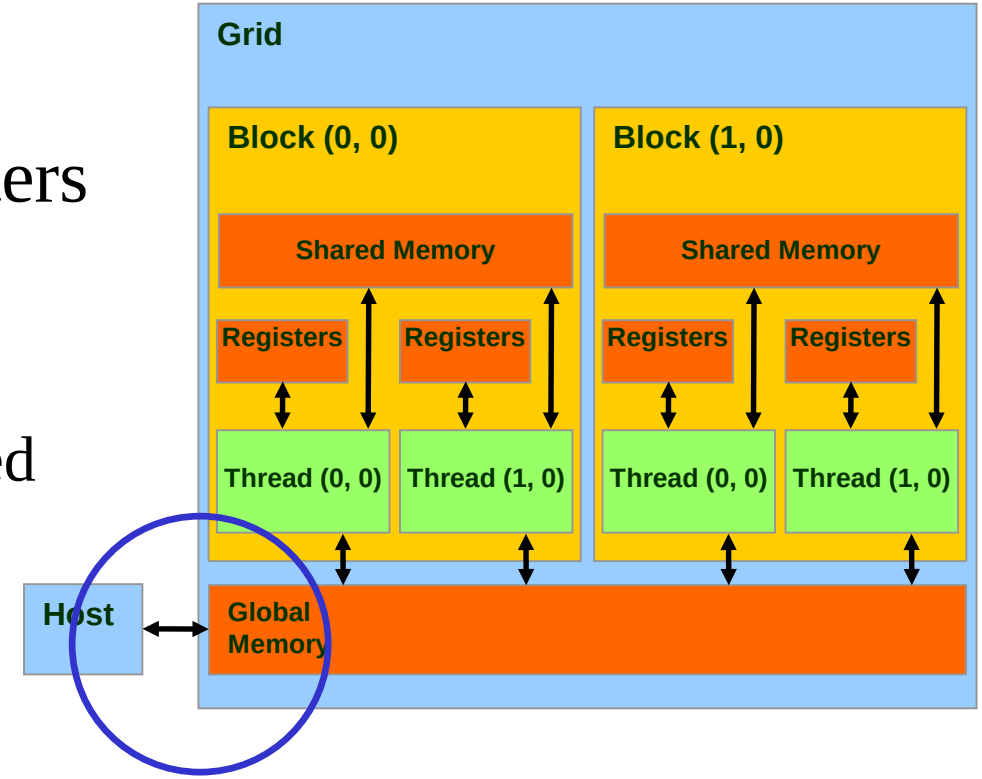
CUDA Device Memory Allocation (cont.)

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

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

cudaMalloc((void**)&Md, size);
cudaFree(Md);
```

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.)

- Code example:
  - Transfer a 64 * 64 single precision float array
  - M is in host memory and Md is in device memory
  - cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost are symbolic constants

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

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

CUDA Keywords

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
- For functions executed on the device:
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments

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
   >>>(...);
```

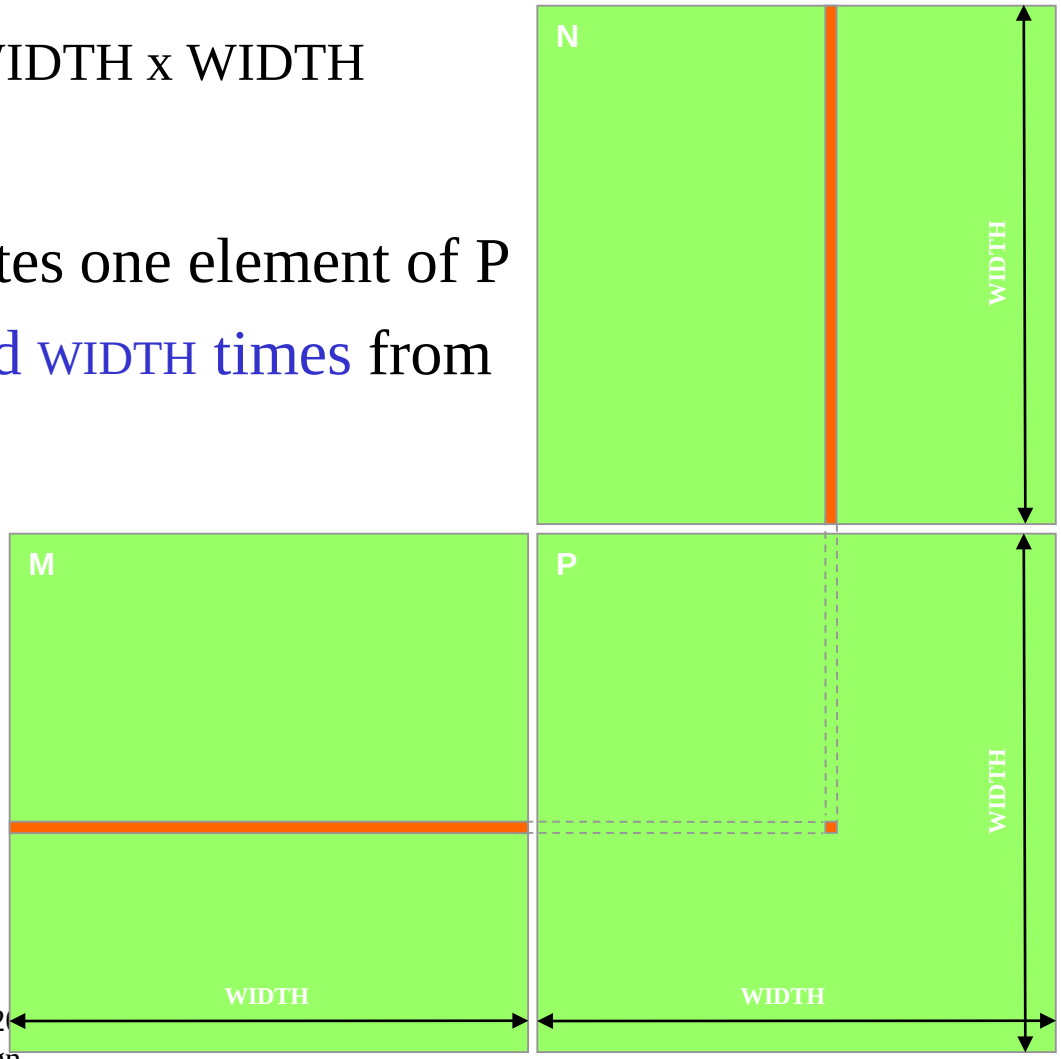
- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking

A Simple Running Example Matrix Multiplication

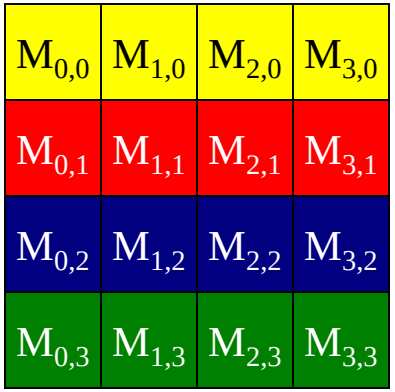
- A simple matrix multiplication example that illustrates the basic features of memory and thread management in CUDA programs
  - Leave shared memory usage until later
  - Local, register usage
  - Thread ID usage
  - Memory data transfer API between host and device
  - Assume square matrix for simplicity

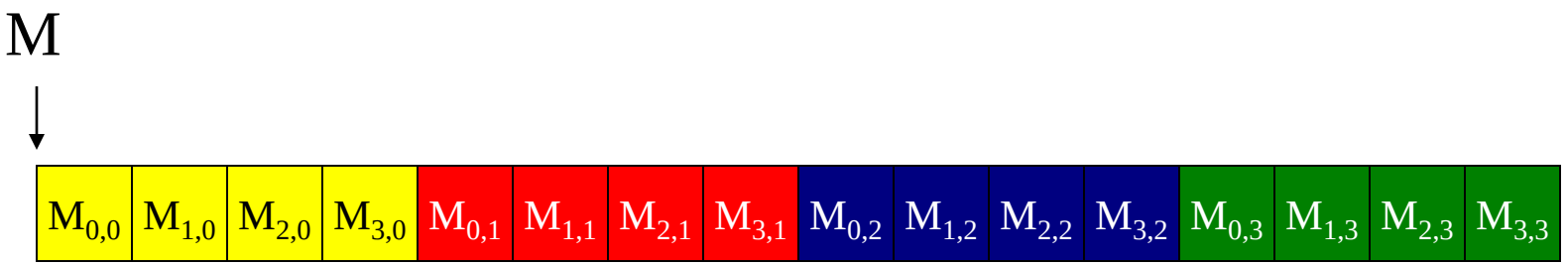
Programming Model: Square Matrix Multiplication Example

- 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



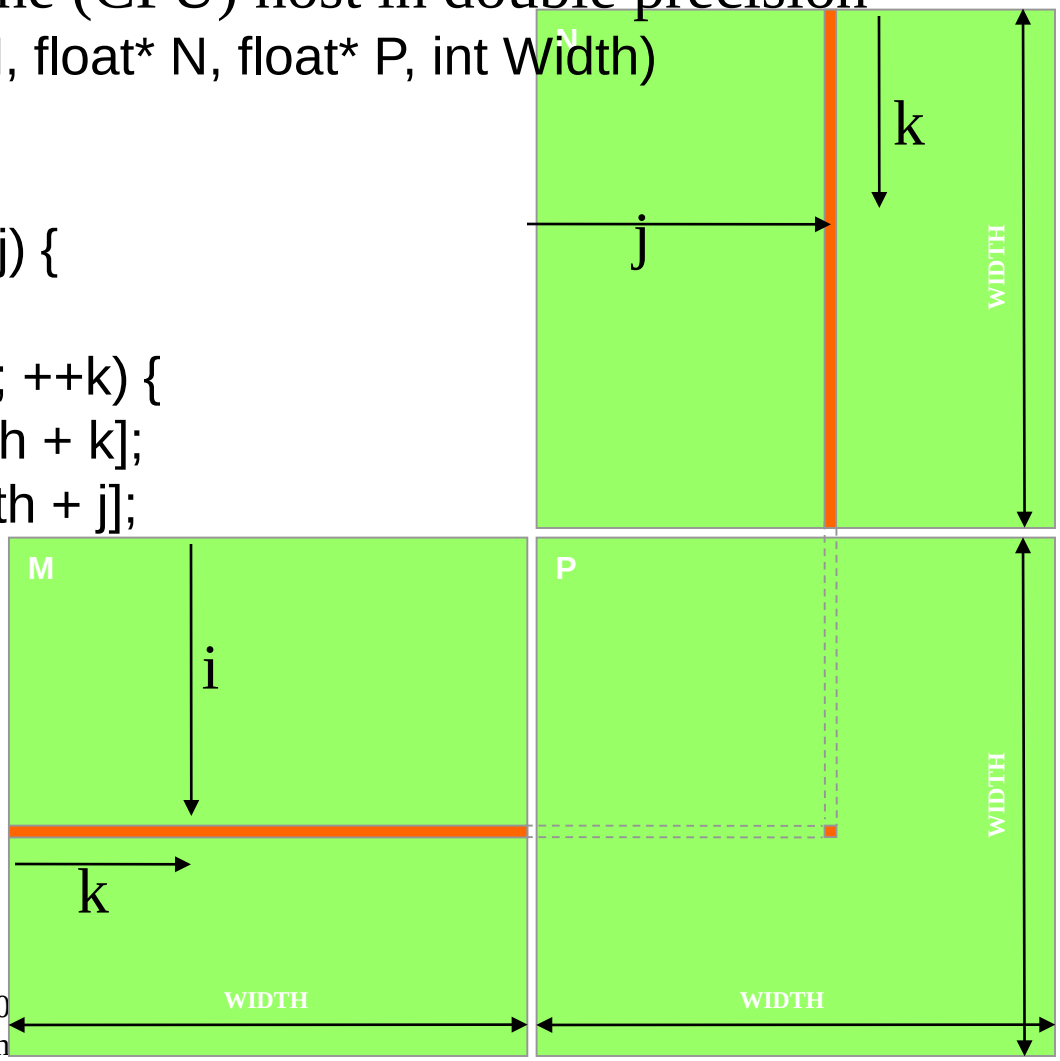
Memory Layout of a Matrix in C





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);

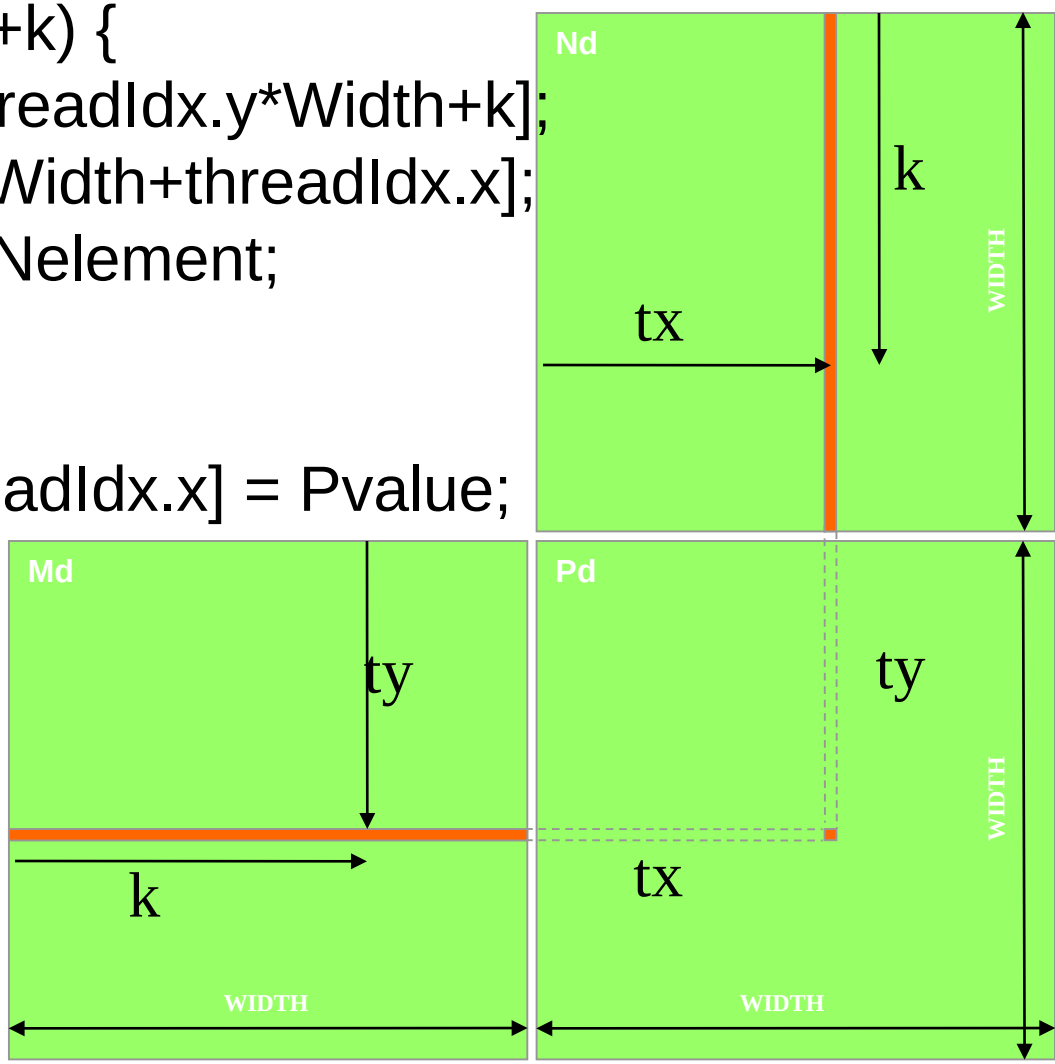
}

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;
}
```

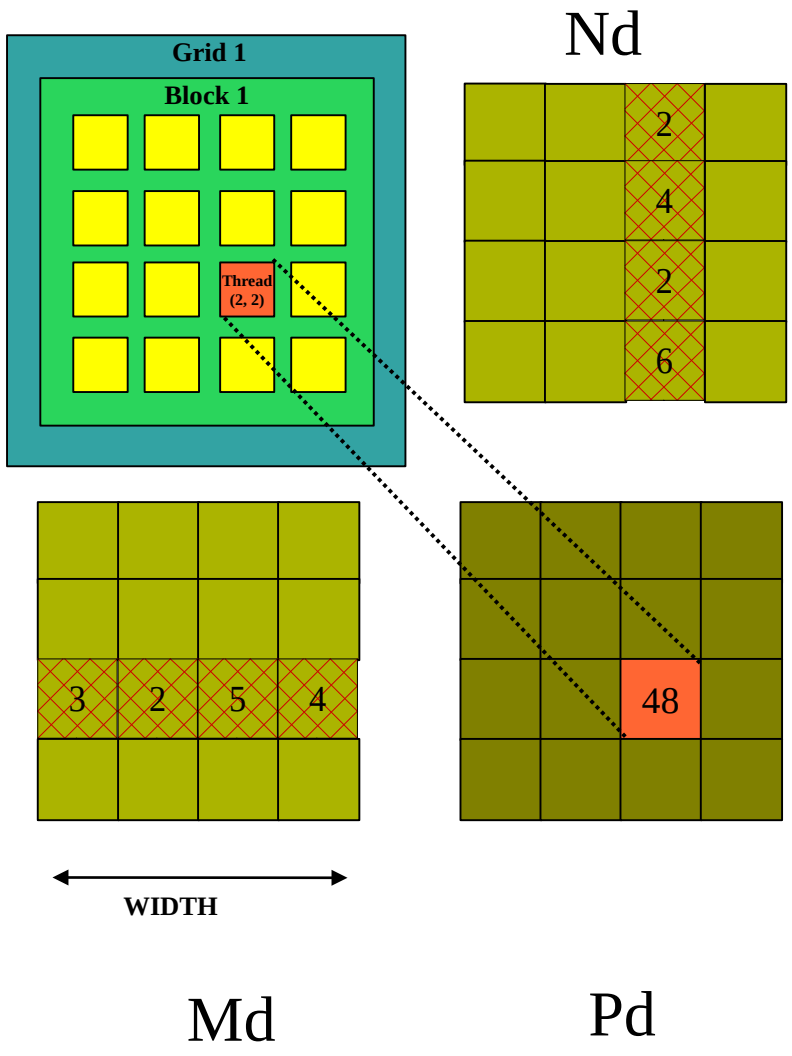


Step 5: Kernel Invocation (Host-side Code)

```
// 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

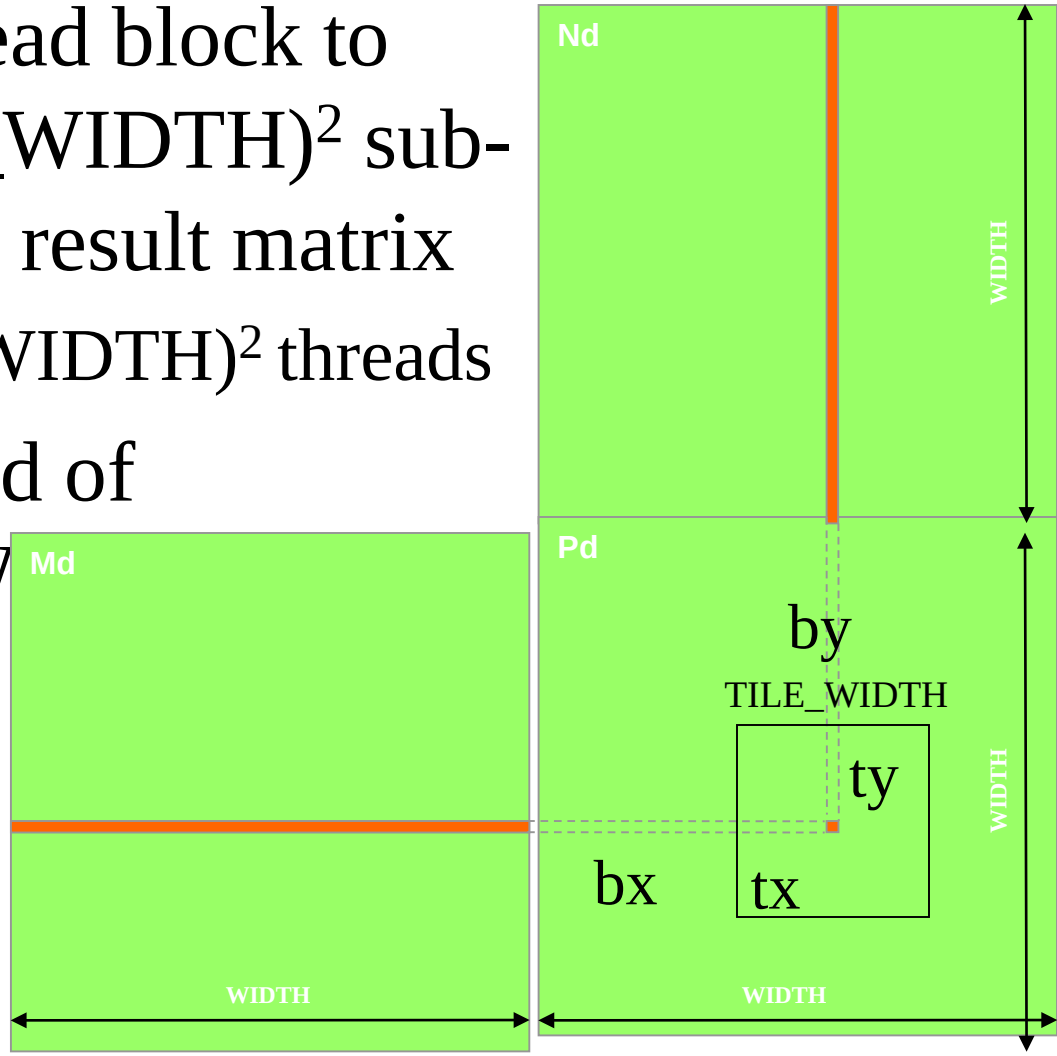
- One Block of threads compute matrix Pd
  - Each thread computes one element of Pd
- Each thread
  - Loads a row of matrix Md
  - Loads a column of matrix Nd
  - Perform one multiply and addition for each pair of Md and Nd elements
  - Compute to off-chip memory access ratio close to 1:1 (not very high)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



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

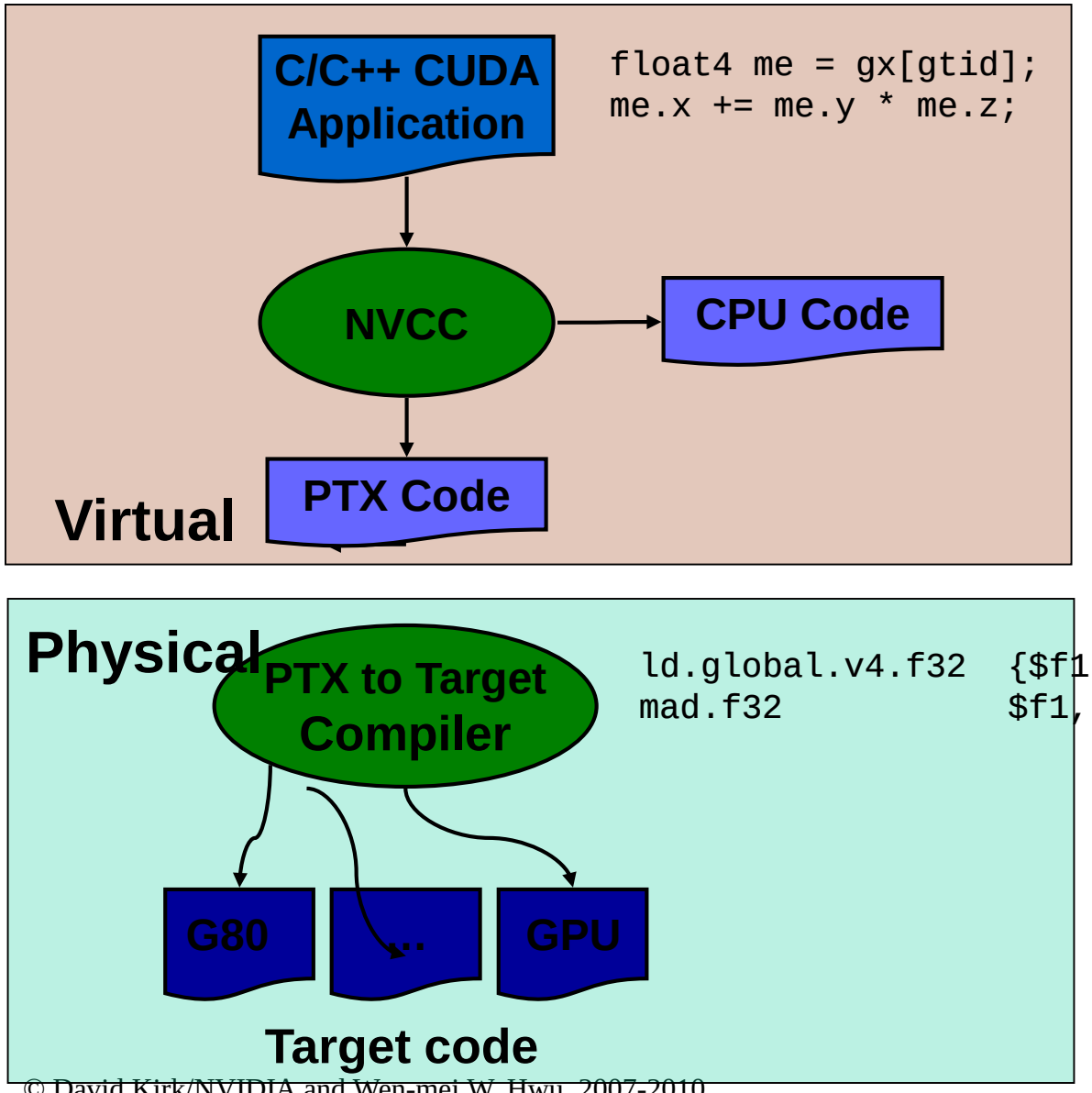
- Have each 2D thread block to compute a $(TILE\_WIDTH)^2$ sub-matrix (tile) of the result matrix
  - Each has $(TILE\_WIDTH)^2$ threads
- Generate a 2D Grid of (WIDTH/TILE_W

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



Some Useful Information on Tools

Compiling a CUDA Program



- Parallel Thread eXecution (PTX)
  - Virtual Machine and ISA
  - Programming model
  - Execution resources and state

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++, cl, ...
- 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

Linking

- Any executable with CUDA code requires two dynamic libraries:
  - The CUDA runtime library (**`cudart`**)
  - The CUDA core library (**`cuda`**)

Debugging Using the Device Emulation Mode

- An executable compiled in device emulation mode (**`nvcc -deviceemu`**) runs completely on the host using the CUDA runtime
  - No need of any device and CUDA driver
  - Each device thread is emulated with a host thread

- Running in device emulation mode, one can:
  - Use host native debug support (breakpoints, inspection, etc.)
  - Access any device-specific data from host code and vice-versa
  - Call any host function from device code (e.g. **`printf`**) and vice-versa
  - Detect deadlock situations caused by improper usage of **`__syncthreads`**

Device Emulation Mode Pitfalls

- Emulated device threads execute sequentially, so simultaneous accesses of the same memory location by multiple threads could produce different results.
- Dereferencing device pointers on the host or host pointers on the device can produce correct results in device emulation mode, but will generate an error in device execution mode

Floating Point

- Results of floating-point computations will slightly differ 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

COMPUTATIONAL THINKING

Objective

- To provide you with a framework based on the techniques and best practices used by experienced parallel programmers for
  - Thinking about the problem of parallel programming
  - Discussing your work with others
  - Addressing performance and functionality issues in your parallel program
  - Using or building useful tools and environments
  - understanding case studies and projects

Fundamentals of Parallel Computing

- Parallel computing requires that
  - The problem can be decomposed into sub-problems that can be safely solved at the same time
  - The programmer structures the code and data to solve these sub-problems concurrently
- The goals of parallel computing are
  - To solve problems in less time, and/or
  - To achieve better solutions

The problems must be large enough to justify parallel computing and to exhibit exploitable concurrency.

A Recommended Reading

Mattson, Sanders, Massingill, *Patterns for Parallel Programming*, Addison Wesley, 2005, ISBN 0-321-22811-1.

- We draw quite a bit from the book
- A good overview of challenges, best practices, and common techniques in all aspects of parallel programming

Key Parallel Programming Steps

1) **To find the concurrency in the problem**
2) To structure the algorithm so that concurrency can be exploited
3) To implement the algorithm in a suitable programming environment
4) To execute and tune the performance of the code on a parallel system

Unfortunately, these have not been separated into levels of abstractions that can be dealt with independently.

Challenges of Parallel Programming

- Finding and exploiting concurrency often requires looking at the problem from a non-obvious angle
  - Computational thinking (J. Wing)
- Dependences need to be identified and managed
  - The order of task execution may change the answers
    - Obvious: One step feeds result to the next steps
    - Subtle: numeric accuracy may be affected by ordering steps that are logically parallel with each other
- Performance can be drastically reduced by many factors
  - Overhead of parallel processing
  - Load imbalance among processor elements
  - Inefficient data sharing patterns
  - Saturation of critical resources such as memory bandwidth

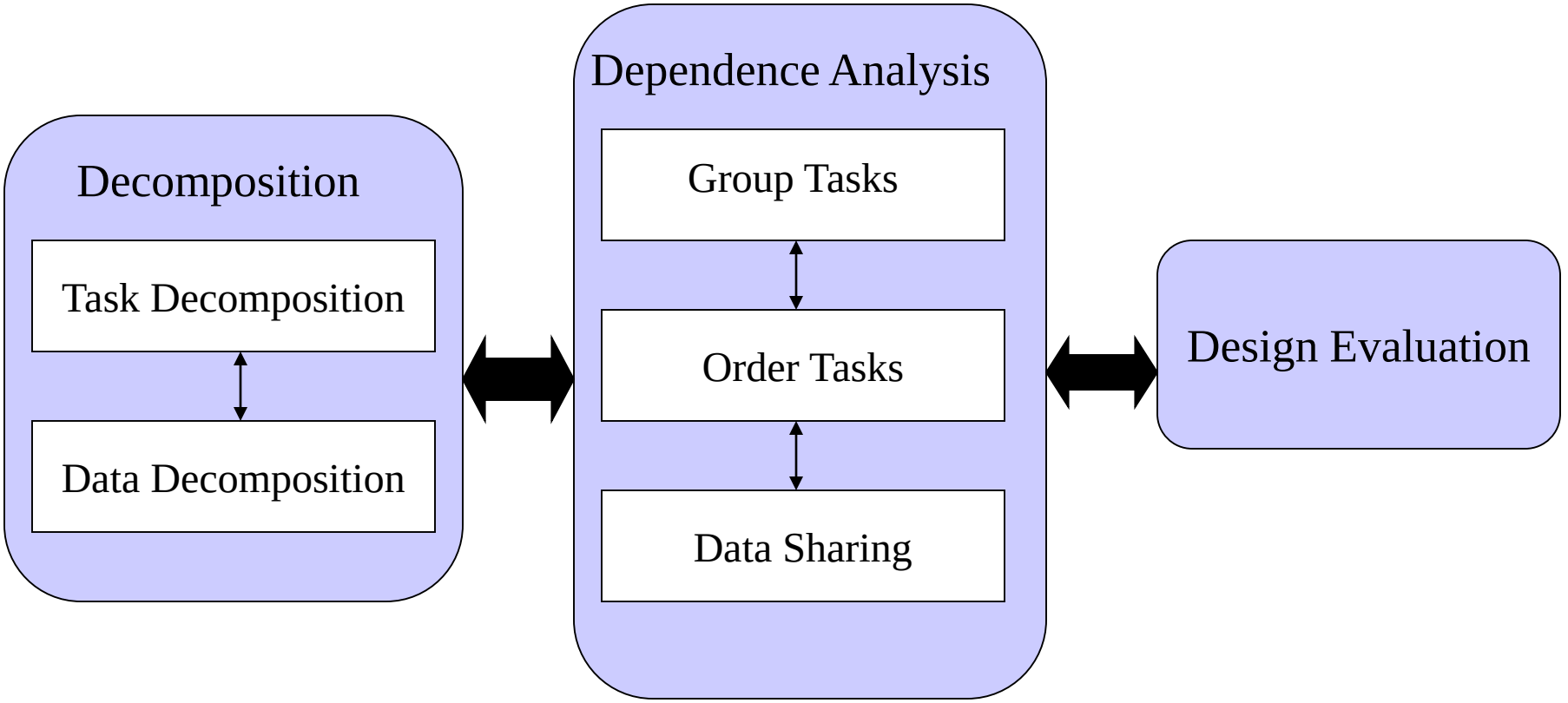
Shared Memory vs. Message Passing

- We will focus on shared memory parallel programming
  - This is what CUDA is based on
  - Future massively parallel microprocessors are expected to support shared memory at the chip level
- The programming considerations of message passing model is quite different!
  - Look at MPI (Message Passing Interface) and its relatives such as Charm++

Finding Concurrency in Problems

- Identify a decomposition of the problem into sub-problems that can be solved simultaneously
  - A task decomposition that identifies tasks for potential concurrent execution
  - A data decomposition that identifies data local to each task
  - A way of grouping tasks and ordering the groups to satisfy temporal constraints
  - An analysis on the data sharing patterns among the concurrent tasks
  - A design evaluation that assesses of the quality the choices made in all the steps

Finding Concurrency – The Process



This is typically a iterative process.
Opportunities exist for dependence analysis to play earlier role in decomposition.

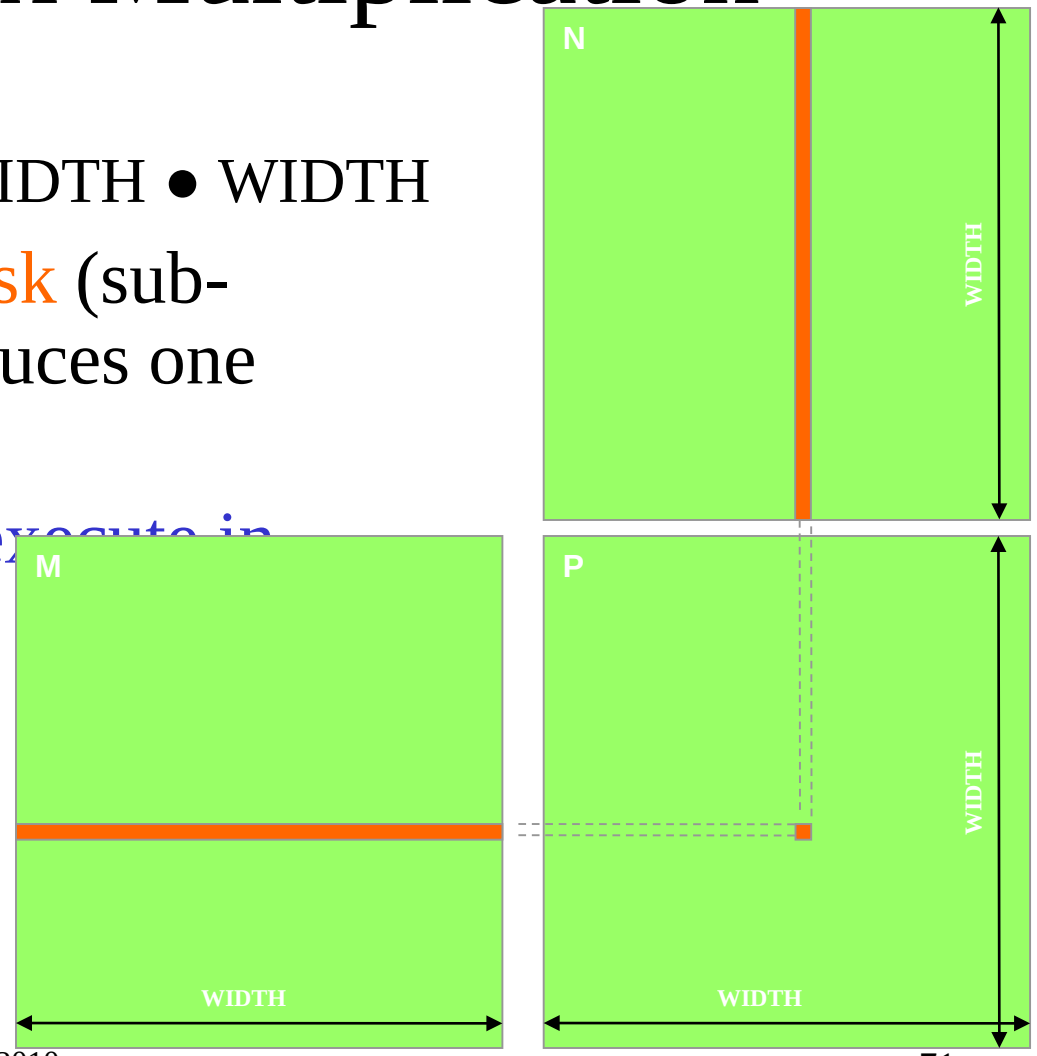
Task Decomposition

- Many large problems can be naturally decomposed into tasks – CUDA kernels are largely tasks
  - The number of tasks used should be adjustable to the execution resources available.
  - Each task must include sufficient work in order to compensate for the overhead of managing their parallel execution.

"In an ideal world, the compiler would find tasks for the programmer. Unfortunately, this almost never happens."
- Mattson, Sanders, Massingill

Task Decomposition Example - Square Matrix Multiplication

- P = M * N of WIDTH • WIDTH
  - One natural task (sub-problem) produces one element of P
  - All tasks can execute in parallel in this

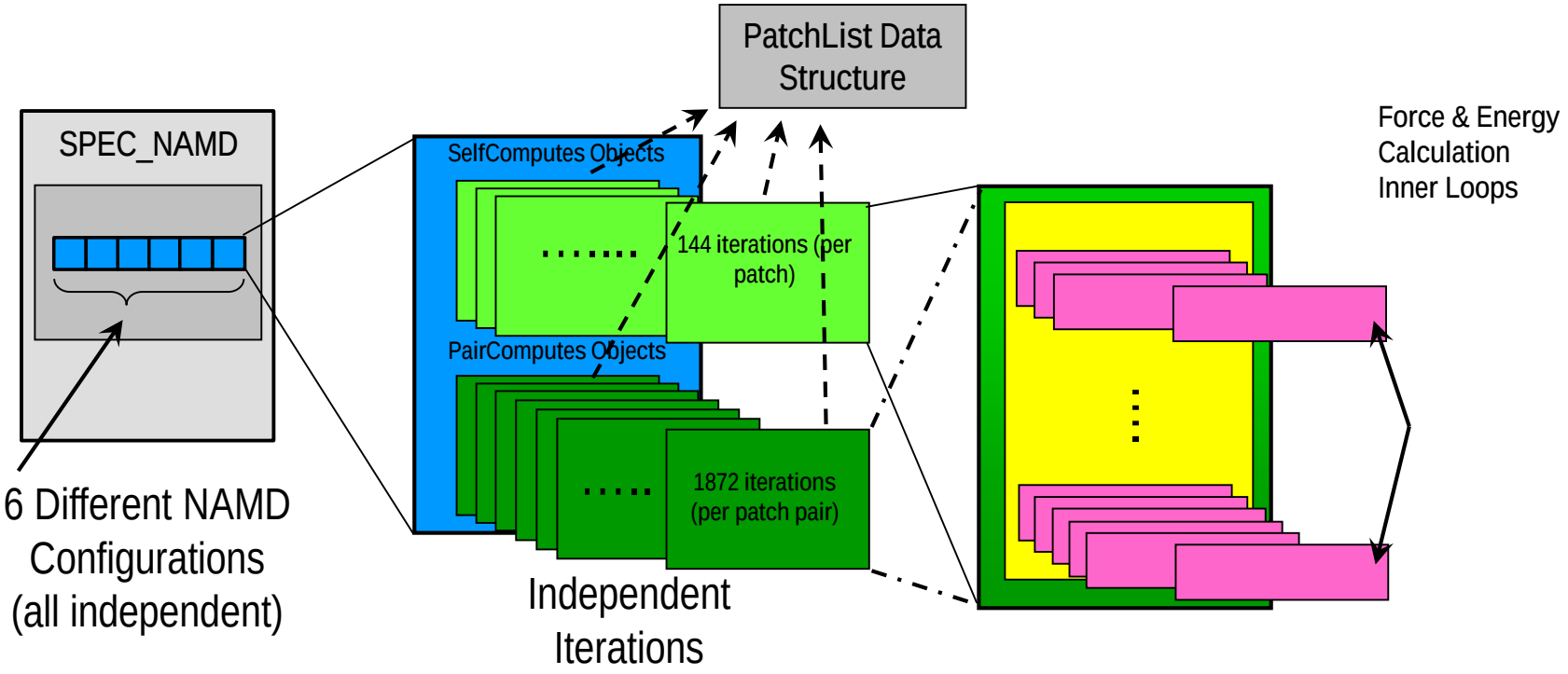


Task Decomposition Example – Molecular Dynamics

- Simulation of motions of a large molecular system
- For each atom, there are natural tasks to calculate
  - Vibrational forces
  - Rotational forces
  - Neighbors that must be considered in non-bonded forces
  - Non-bonded forces
  - Update position and velocity
  - Misc physical properties based on motions
- Some of these can go in parallel for an atom

It is common that there are multiple ways to decompose any given problem.

NAMD

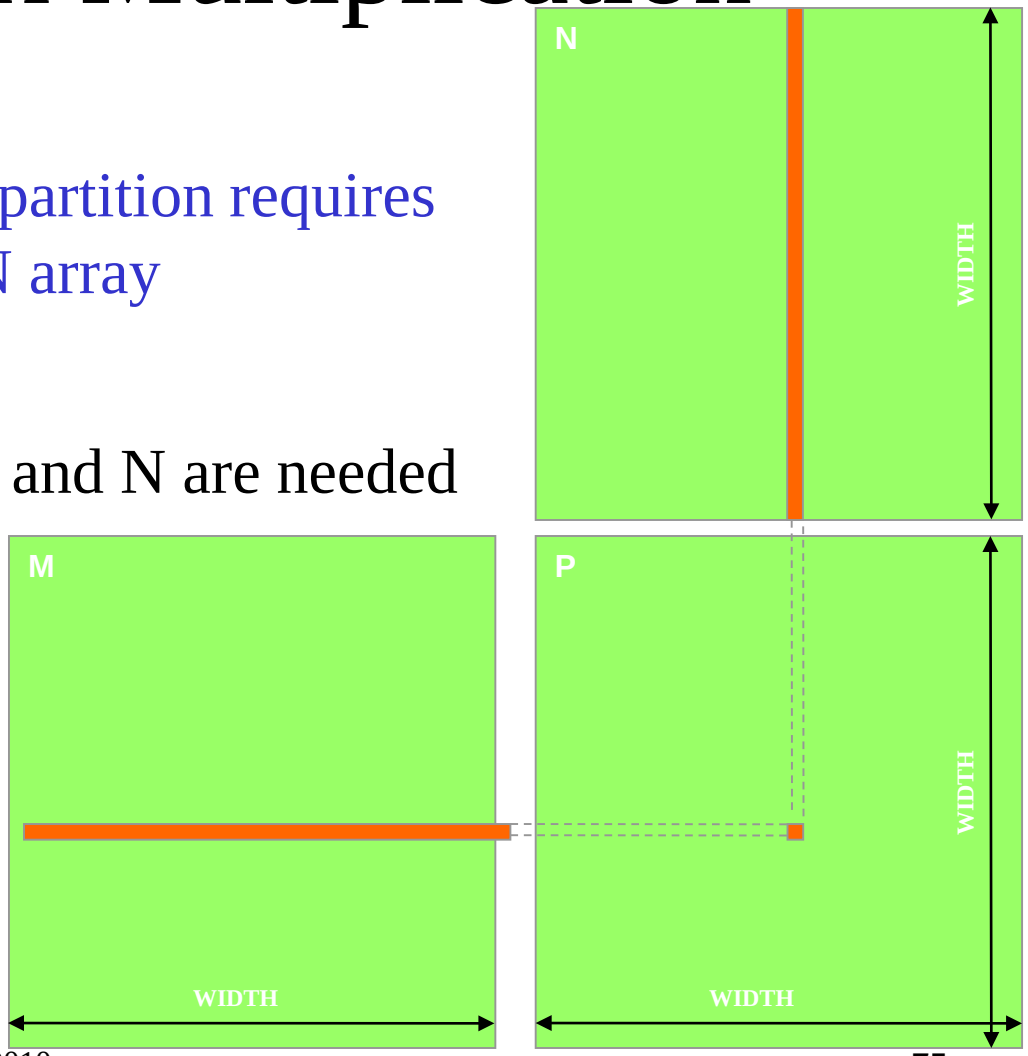


Data Decomposition

- The most compute intensive parts of many large problem manipulate a large data structure
  - Similar operations are being applied to different parts of the data structure, in a mostly independent manner.
  - This is what CUDA is optimized for.
- The data decomposition should lead to
  - Efficient data usage by tasks within the partition
  - Few dependencies across the tasks that work on different partitions
  - Adjustable partitions that can be varied according to the hardware characteristics

Data Decomposition Example - Square Matrix Multiplication

- Row blocks
  - Computing each partition requires access to entire N array
- Square sub-blocks
  - Only bands of M and N are needed

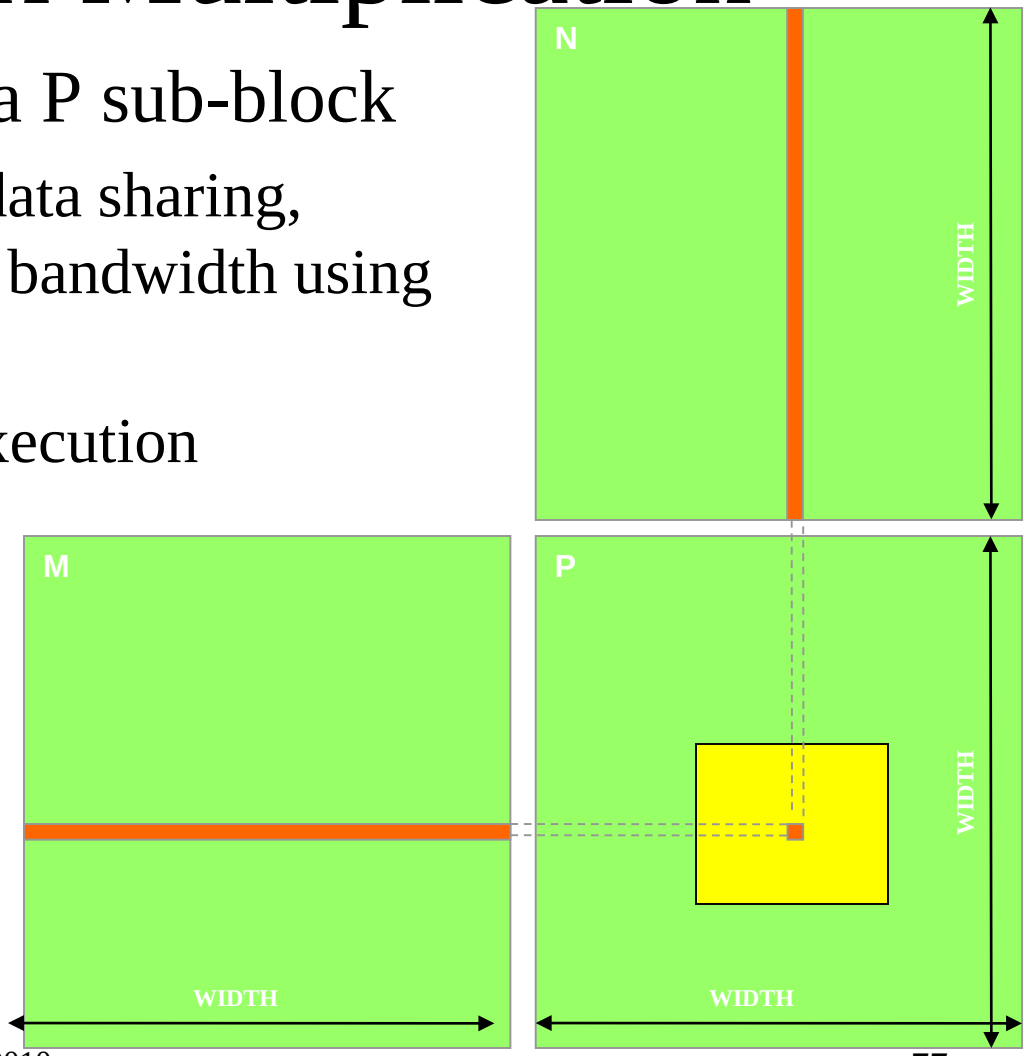


Tasks Grouping

- Sometimes natural tasks of a problem can be grouped together to improve efficiency
  - Reduced synchronization overhead – all tasks in the group can use a barrier to wait for a common dependence
  - All tasks in the group efficiently share data loaded into a common on-chip, shared storage (Shard Memory)
  - Grouping and merging dependent tasks into one task reduces need for synchronization
  - CUDA thread blocks are task grouping examples.

Task Grouping Example - Square Matrix Multiplication

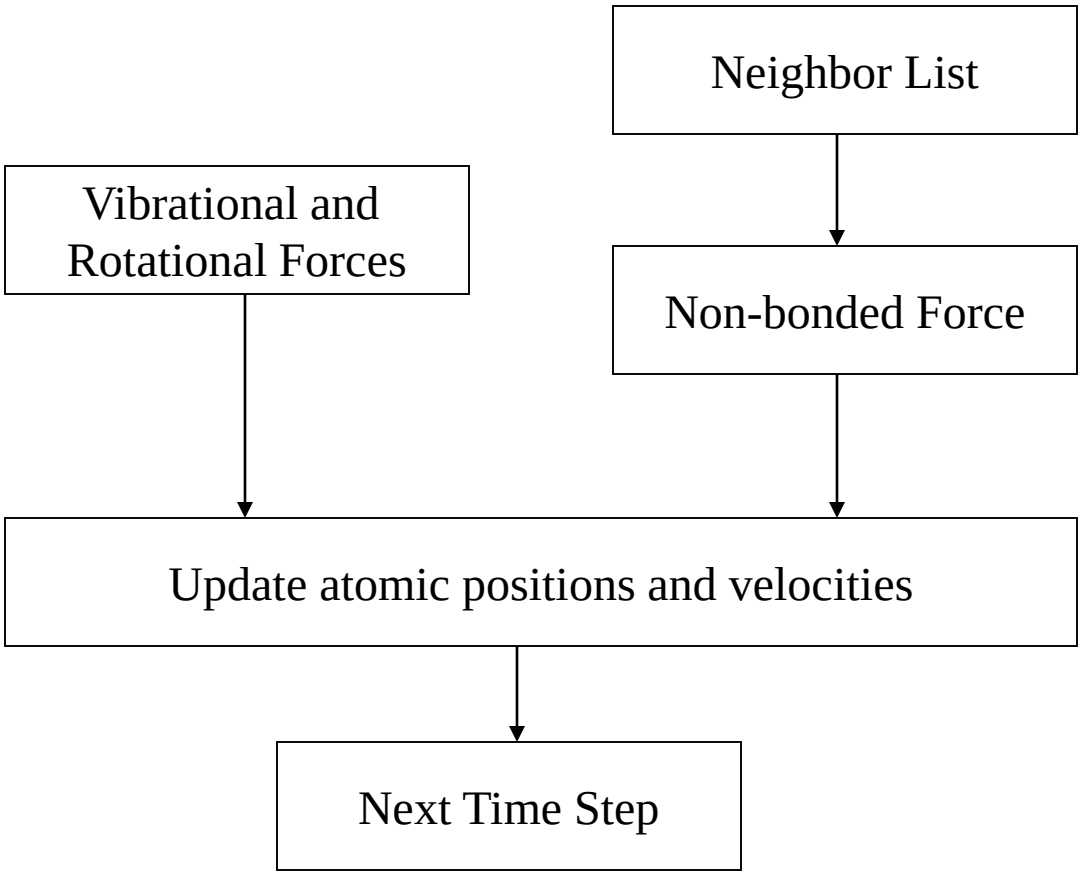
- Tasks calculating a P sub-block
  - Extensive input data sharing, reduced memory bandwidth using Shared Memory
  - All synched in execution



Task Ordering

- Identify the data and resource required by a group of tasks before they can execute them
  - Find the task group that creates it
  - Determine a temporal order that satisfy all data constraints

Task Ordering Example: Molecular Dynamics



Data Sharing

- Data sharing can be a double-edged sword
  - Excessive data sharing can drastically reduce advantage of parallel execution
  - Localized sharing can improve memory bandwidth efficiency
- Efficient memory bandwidth usage can be achieved by synchronizing the execution of task groups and coordinating their usage of memory data
  - Efficient use of on-chip, shared storage
- Read-only sharing can usually be done at much higher efficiency than read-write sharing, which often requires synchronization

Data Sharing Example – Matrix Multiplication

- Each task group will finish usage of each sub-block of N and M before moving on
  - N and M sub-blocks loaded into Shared Memory for use by all threads of a P sub-block
  - Amount of on-chip Shared Memory strictly limits the number of threads working on a P sub-block
- Read-only shared data can be more efficiently accessed as Constant or Texture data

Data Sharing Example – Molecular Dynamics

- The atomic coordinates
  - Read-only access by the neighbor list, bonded force, and non-bonded force task groups
  - Read-write access for the position update task group
- The force array
  - Read-only access by position update group
  - Accumulate access by bonded and non-bonded task groups
- The neighbor list
  - Read-only access by non-bonded force task groups
  - Generated by the neighbor list task group

Key Parallel Programming Steps

1) To find the concurrency in the problem
2) **To structure the algorithm to translate concurrency into performance**
3) To implement the algorithm in a suitable programming environment
4) To execute and tune the performance of the code on a parallel system

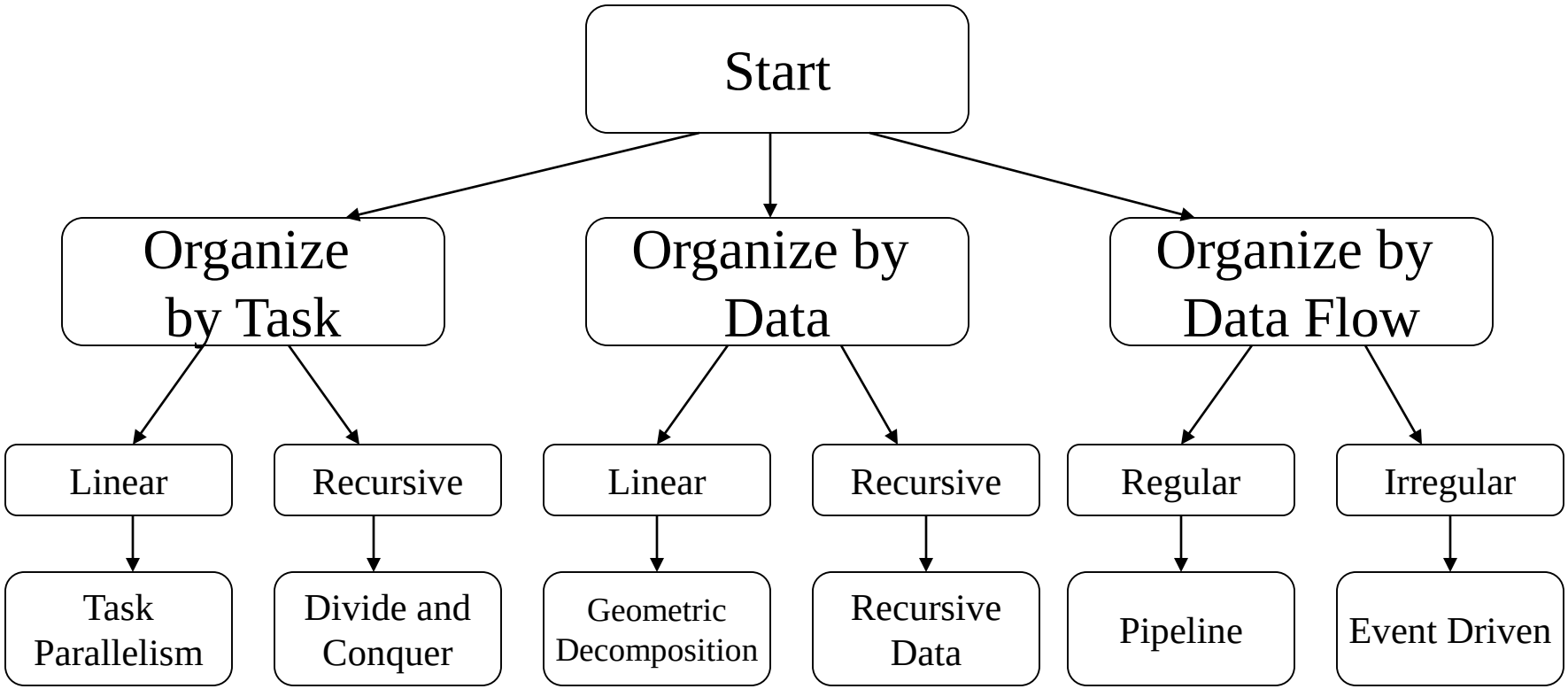
Unfortunately, these have not been separated into levels of abstractions that can be dealt with independently.

Algorithm

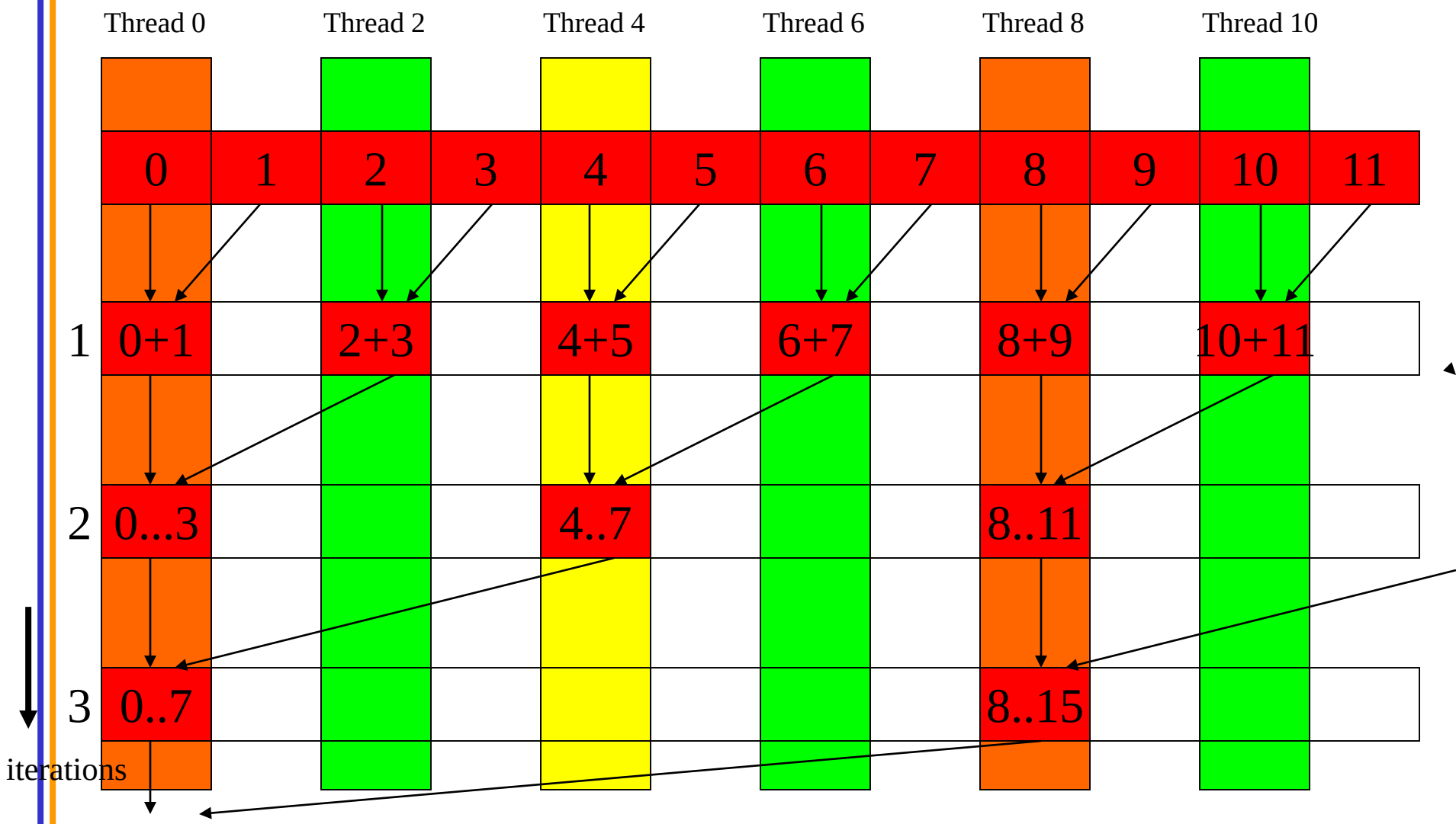
- A step by step procedure that is guaranteed to terminate, such that each step is precisely stated and can be carried out by a computer
  - Definiteness – the notion that each step is precisely stated
  - Effective computability – each step can be carried out by a computer
  - Finiteness – the procedure terminates

- Multiple algorithms can be used to solve the same problem
  - Some require fewer steps
  - Some exhibit more parallelism
  - Some have larger memory footprint than others

Choosing Algorithm Structure

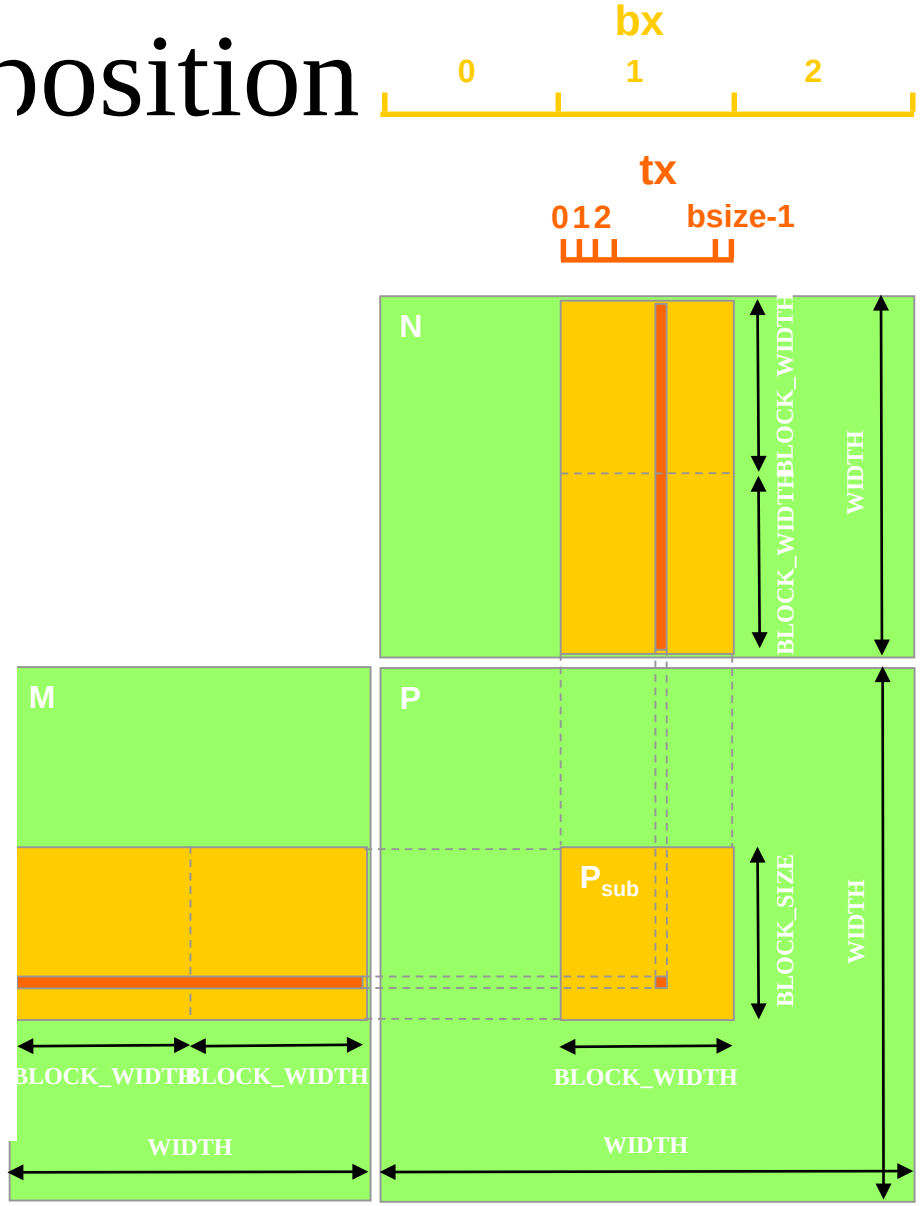


Mapping a Divide and Conquer Algorithm



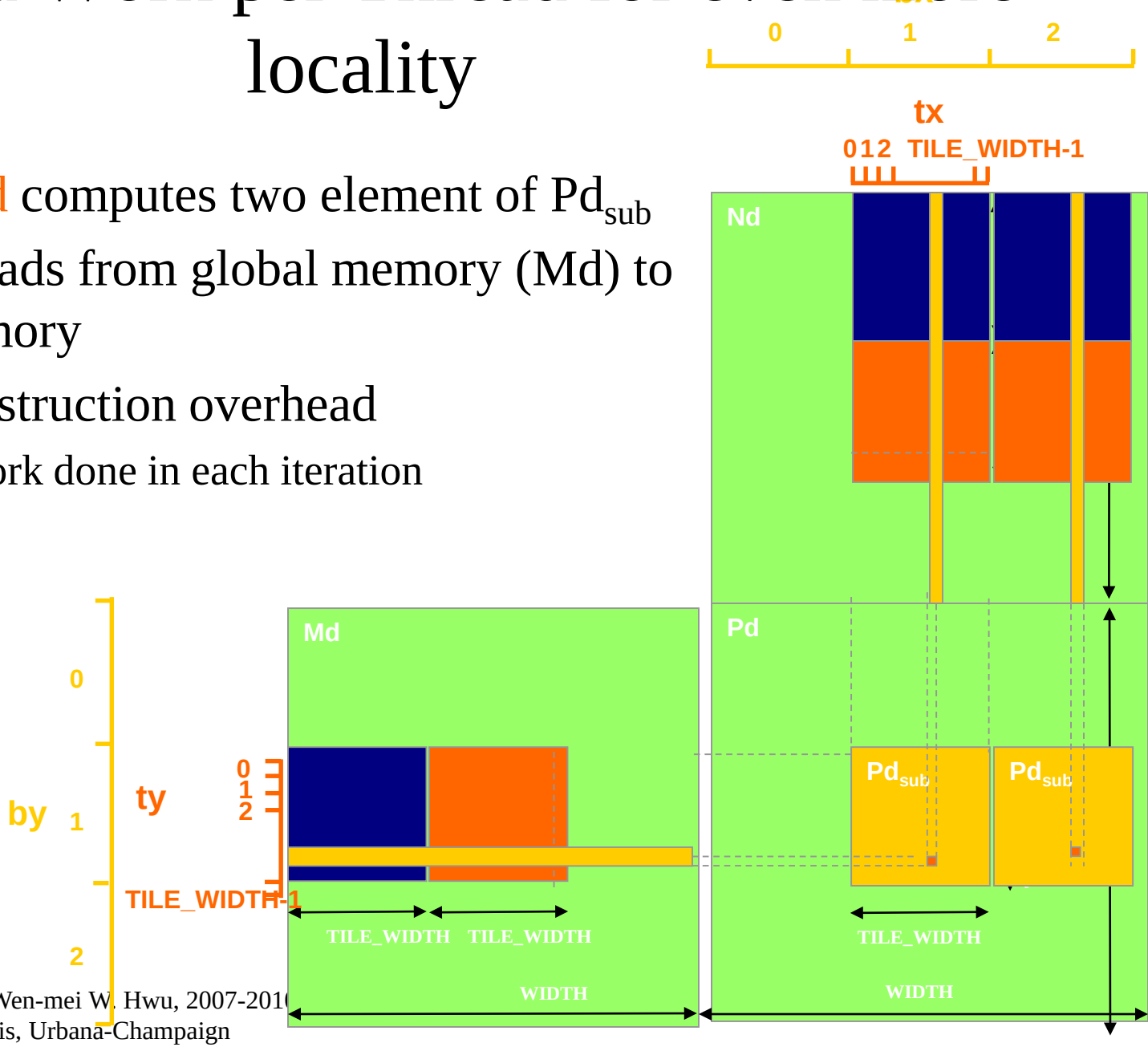
Tiled (Stenciled) Algorithms are Important for Geometric Decomposition

- A framework for memory data sharing and reuse by increasing data access locality.
  - Tiled access patterns allow small cache/scartchpad memories to hold on to data for re-use.
  - For matrix multiplication, a 16X16 thread block perform 2*256 = 512 float loads from device memory for 256 * (2*16) = 8,192 mul/add operations.
- A convenient framework for organizing threads (tasks)



Increased Work per Thread for even more locality

- Each thread computes two element of Pd_{sub}
- Reduced loads from global memory (Md) to shared memory
- Reduced instruction overhead
  - More work done in each iteration



Double Buffering - a frequently used algorithm pattern

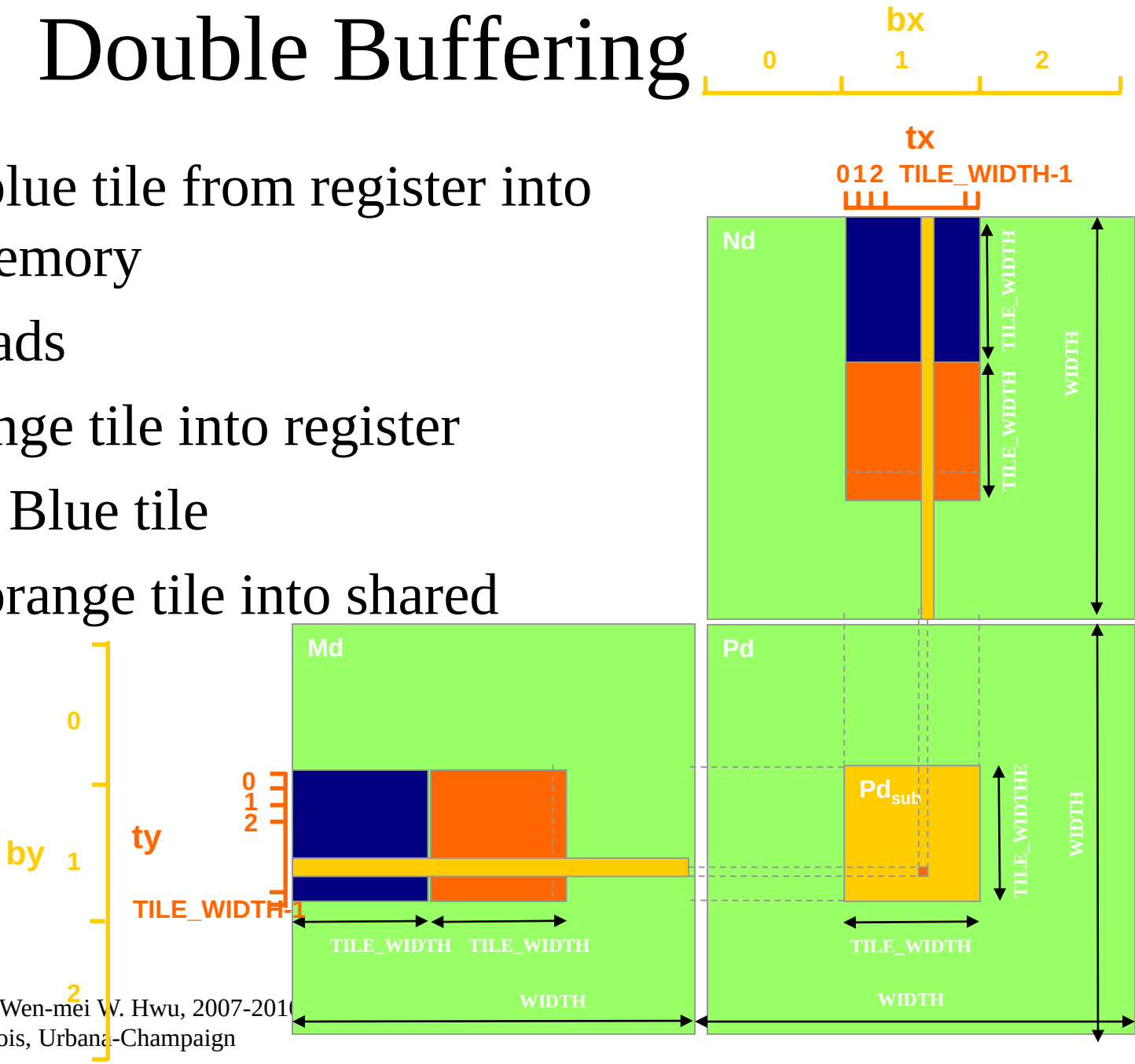
- One could double buffer the computation, getting better instruction mix within each thread
  - This is classic software pipelining in ILP compilers

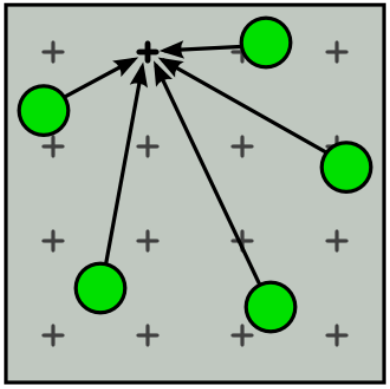
```
Loop {

  Load current tile to shared memory

  syncthreads()

  Compute current tile

  syncthreads()
}
```

```
Load next tile from global memory

Loop {
  Deposit current tile to shared memory

  syncthreads()

  Load next tile from global memory

  Compute current tile

  syncthreads()
}
```

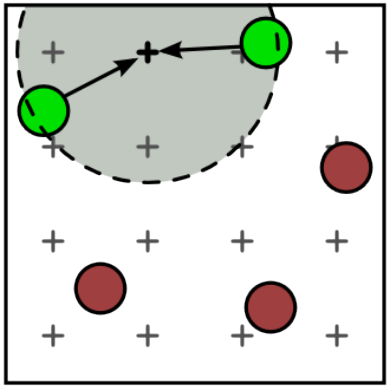
Double Buffering

- Deposit blue tile from register into shared memory
- Syncthreads
- Load orange tile into register
- Compute Blue tile
- Deposit orange tile into shared memory
- ….

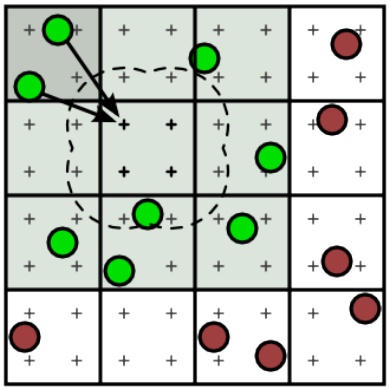




(a) Direct summation
At each grid point, sum the electrostatic potential from all charges



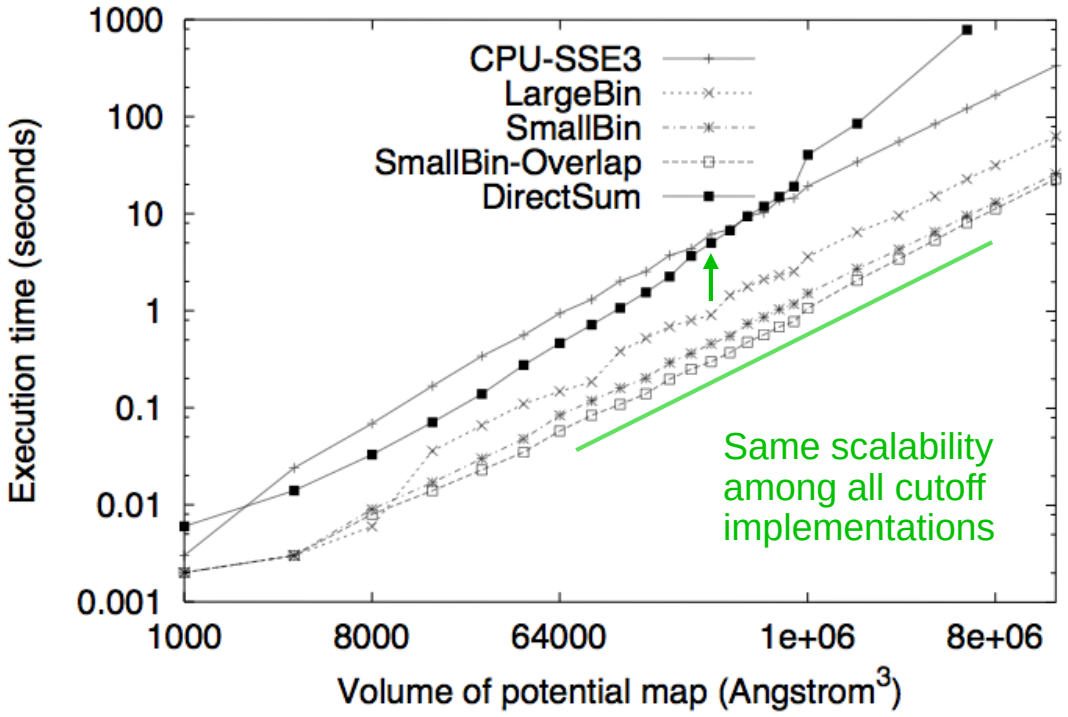
(b) Cutoff summation
Electrostatic potential from nearby charges summed; spatially sort charges first



(c) Cutoff summation using direct summation kernel
Spatially sort charges into bins; adapt direct summation to process a bin

Figure 10.2 Cutoff Summation algorithm

Cut-Off Summation Restores Data Scalability



Scalability and Performance of different algorithms for calculating electrostatic potential map.