King Fahd University of Petroleum and Minerals
Computer Engineering Department

SOME RESEARCH DIRECTIONS IN MASSIVELY-PARALLEL COMPUTING

Dr. Mayez Al-Mouhamed

Professor, Computer Engineering Department

Contents

Many-core Technology

- Programmable accelerators to improve performance for specific domains of applications (DSP, Games, etc)

- Discrete video processors have long been included to meet specialized needs of rendering images at video rates

- Under the pressures of the consumer gaming and professional workstation market, Graphical Processing Units (GPUs) have evolved to deliver ever-increasing amounts of performance

- The two primary vendors in GPU market, NVIDIA and ATI (part of AMD), felt the pressure to provide more programmable access to their processors.

Many-core Processors

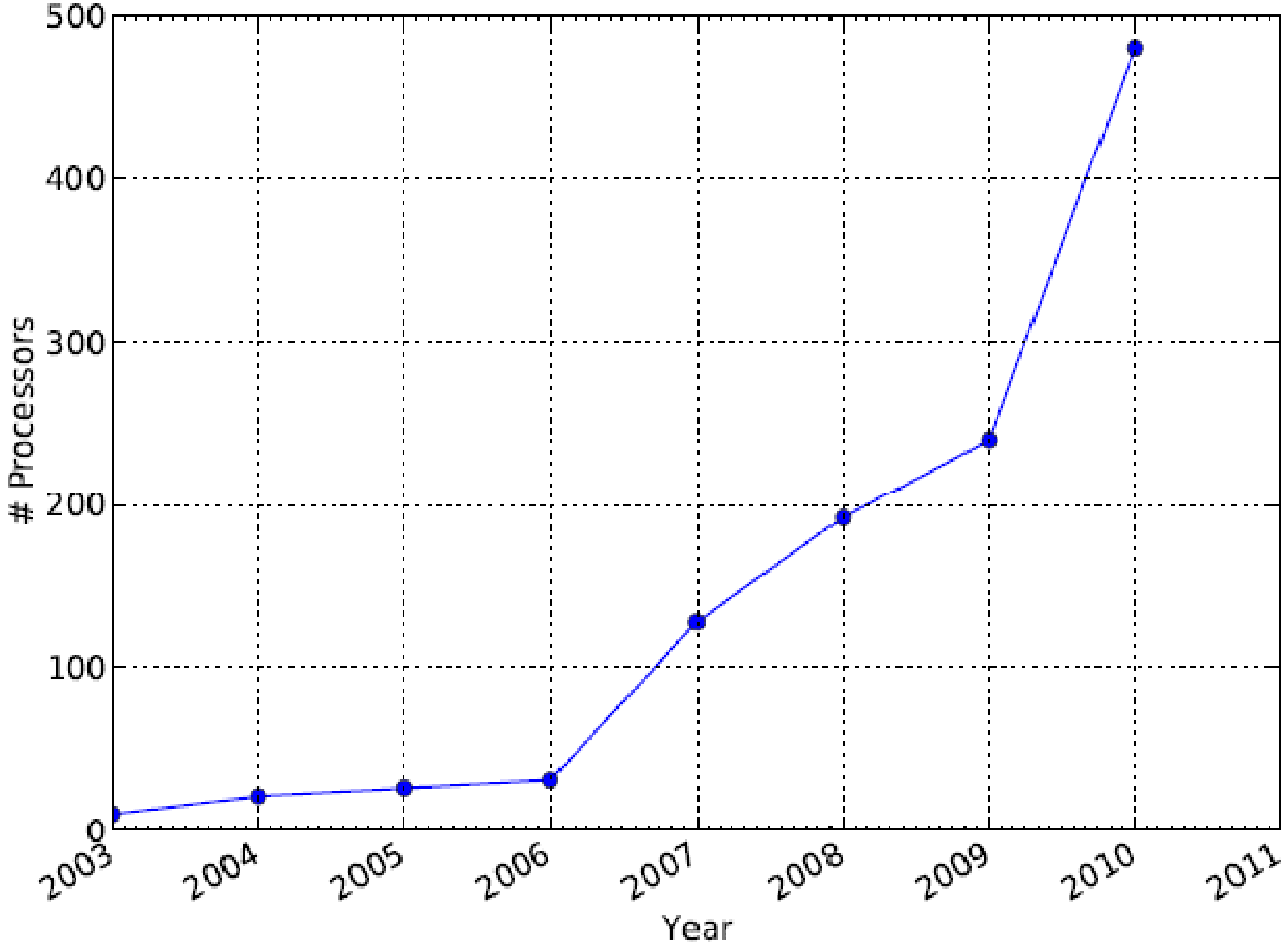
- A market was emerging for general purpose computations on non-graphics data on the GPUs leading the way with a General Purpose GPU solution (GPGPU, capable of 100s GFLOPs)

- Programming heterogeneous platform (GPUs and CPUs):
  - NVIDIA introduced CUDA (Compute Unified Device Architecture) in 2007
  - AMD adopted OpenCL (Open Computing Language) in 2009

- GPU: use massive multithreading, fast context switching, and high memory bandwidth, and overlapping long-latency loads in stalled threads with computation in other threads (multiple streaming multiprocessors with potentially hundreds of cores)

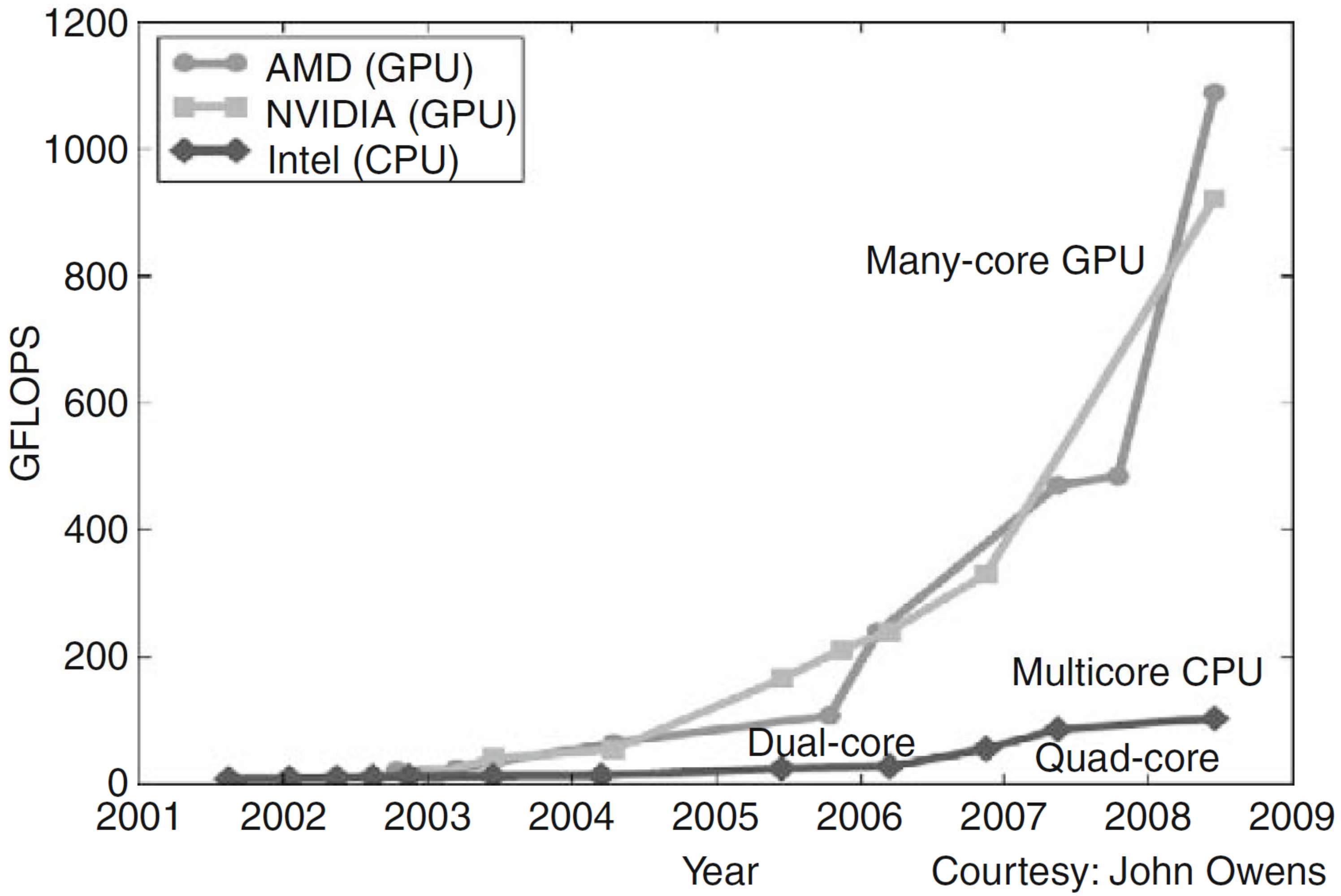
- CUDA (ext. to C) is most widely used parallel programming framework for general purpose GPU computations.

Many-core for Supercomputing



NVIDIA

Many-core for Supercomputing

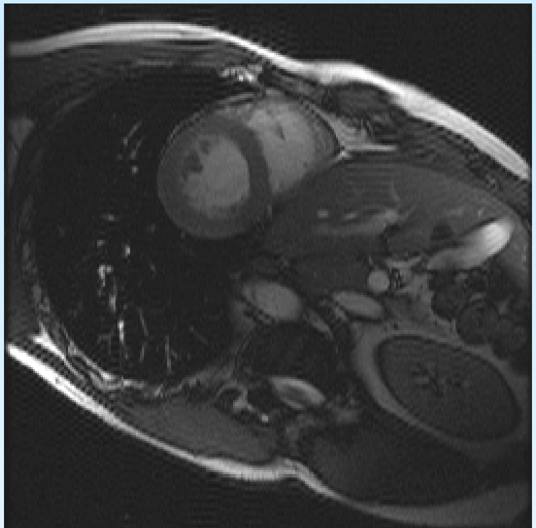


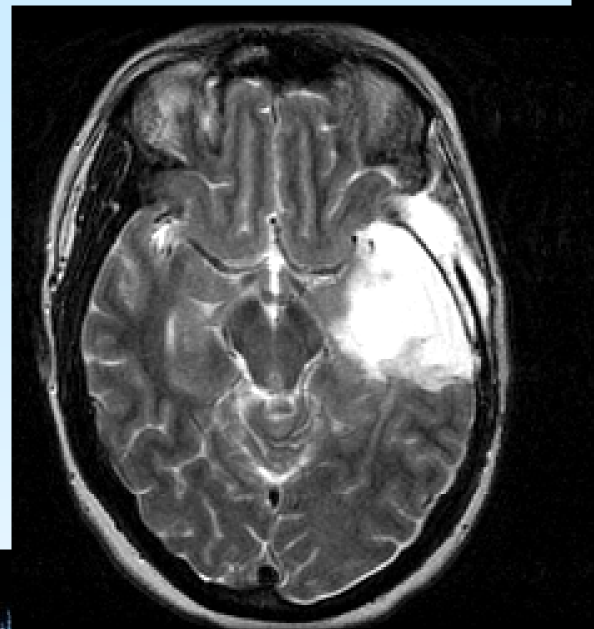
GPU Application to MRI

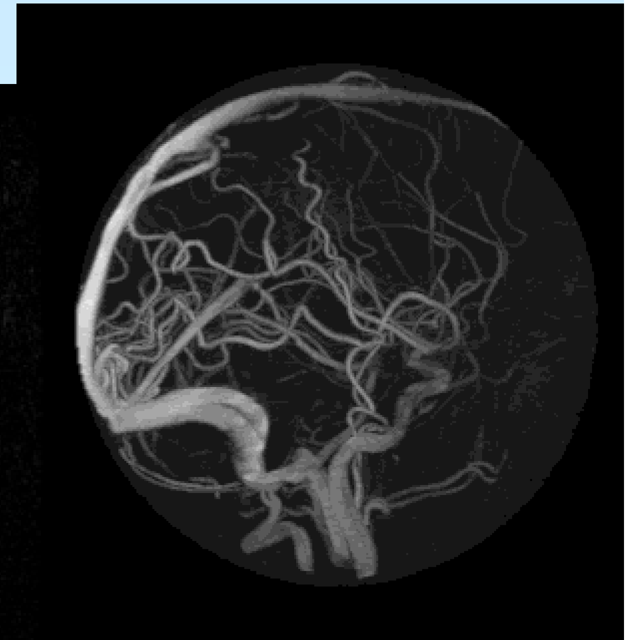
Justin Haldar
University of Illinois at Urbana Champagn

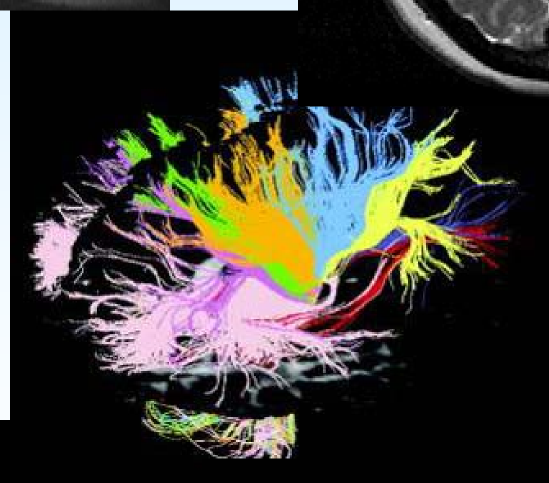
S. S. Stone, et al. "Accelerating Advanced MRI Reconstructions on GPUs." Journal of Parallel and Distributed Computing 68:1307-1318, 2008

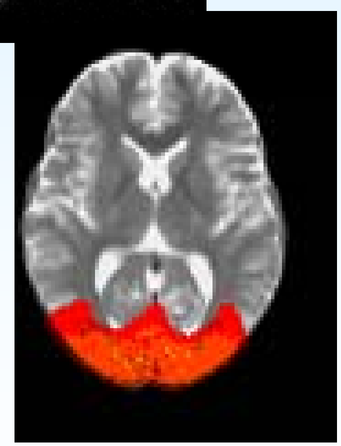
Introduction to MRI

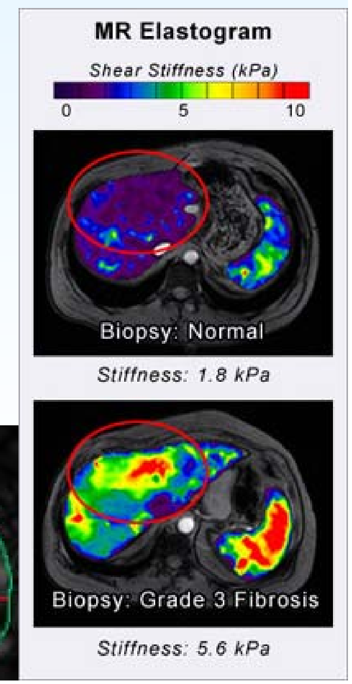


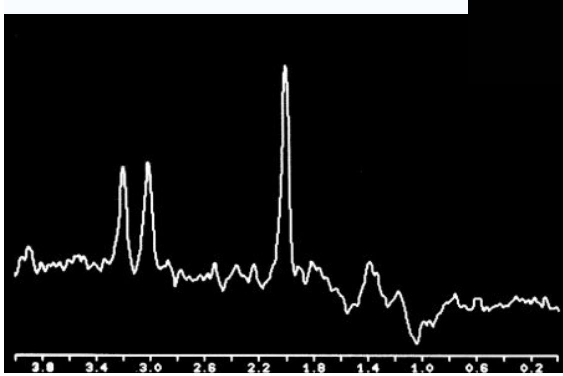


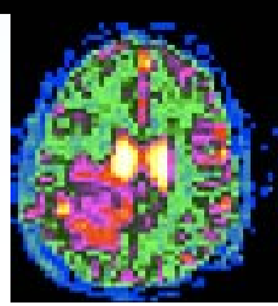


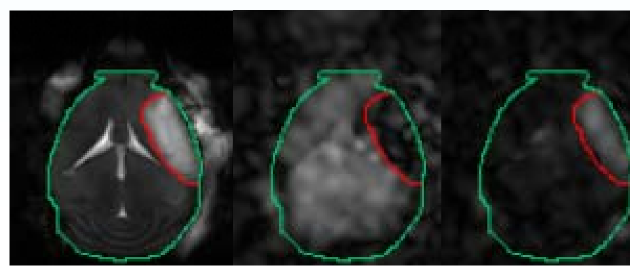






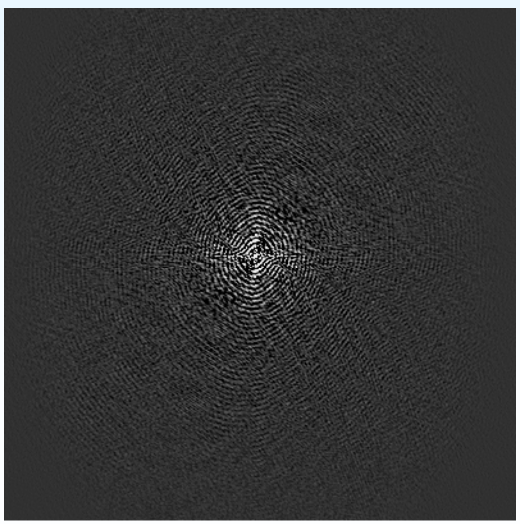






MRI Pipeline

Data Acquisition



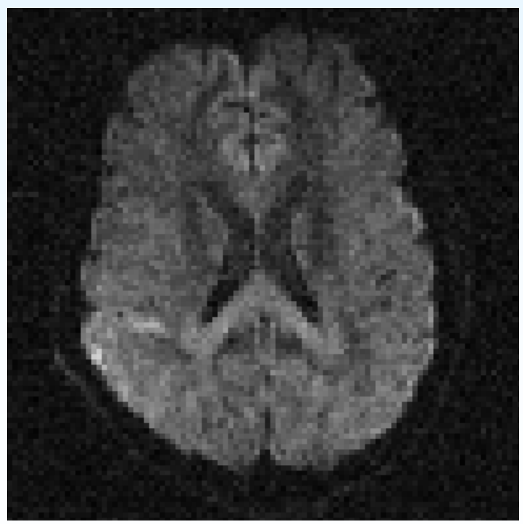
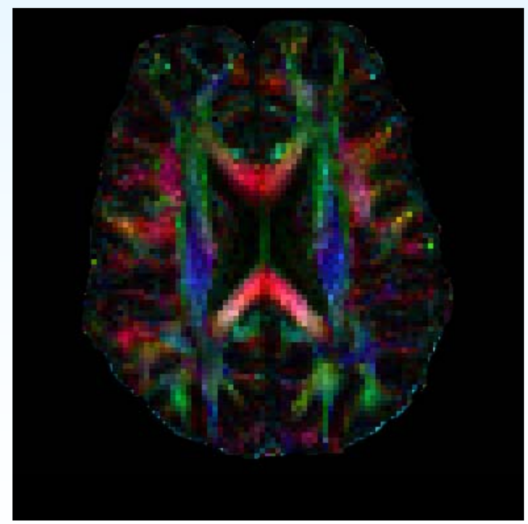


Image Reconstruction



Parameter Estimation

Interpretation

MRI Pipeline

Faster !

- **Reduce scan time/image artifacts**
- **Immediate diagnosis and refinement**

Interpretation

Data Acquisition

Parameter Estimation

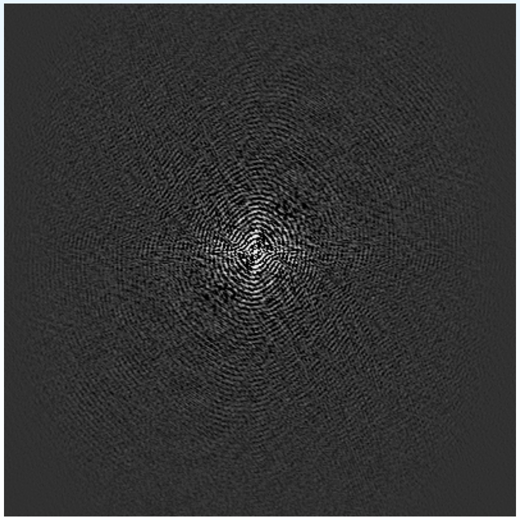
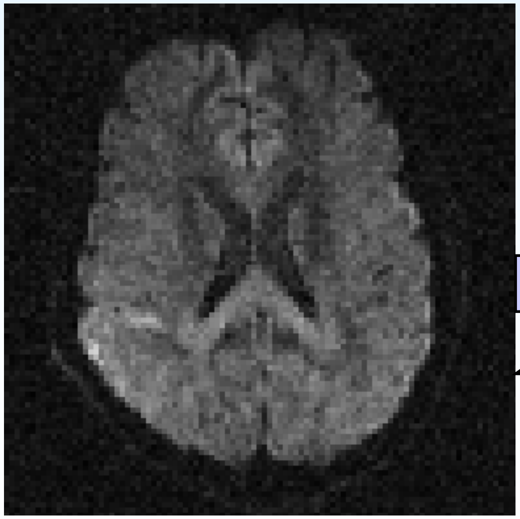
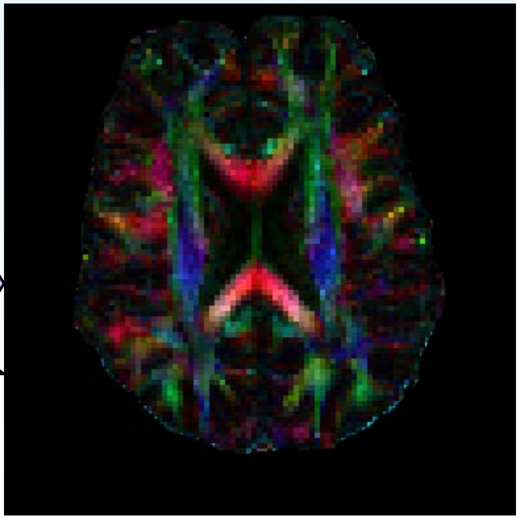


Image Reconstruction

Image Reconstruction

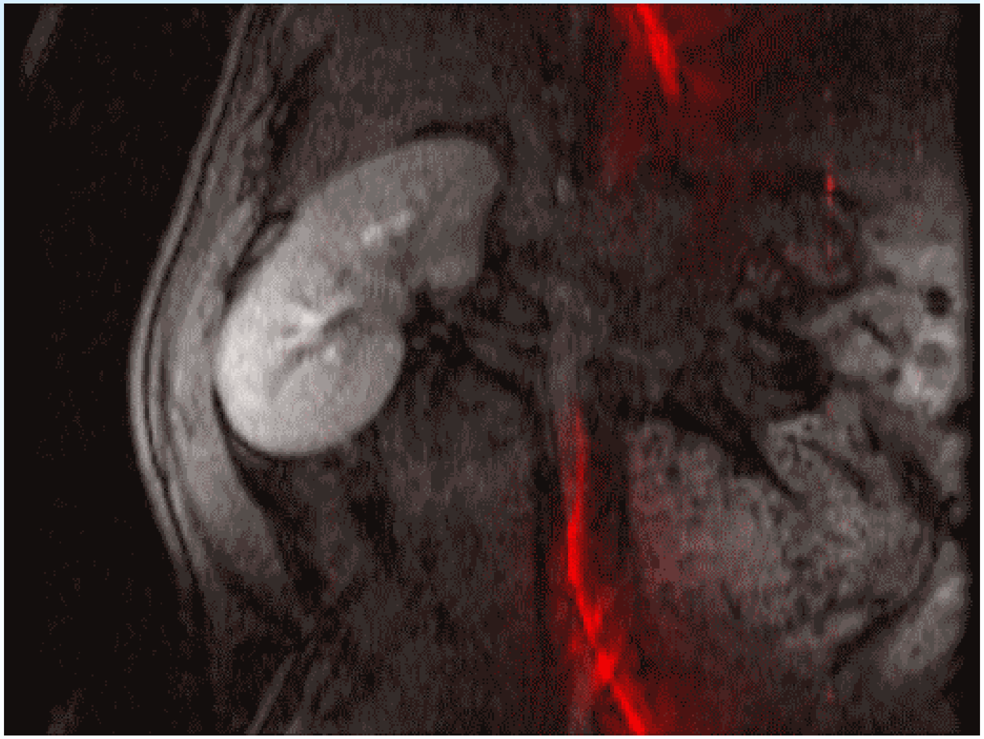




GPU

GPU

Example: Interventional MRI

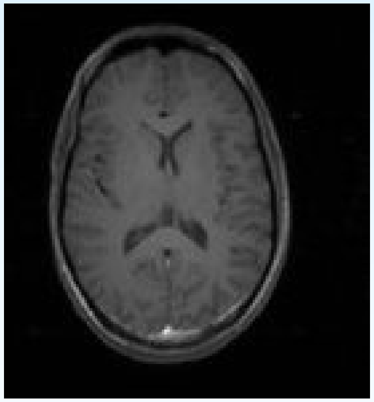


Real-time reconstruction is necessary to provide feedback to surgeon

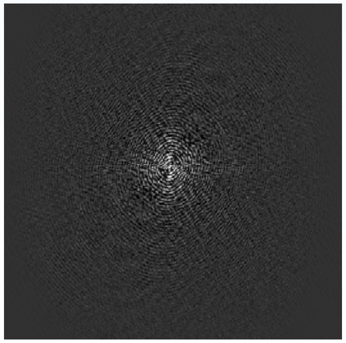
MRI data in Fourier Space

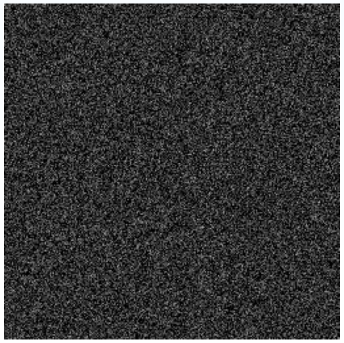
- Ignoring several effects, MRI image and signal are a Fourier transform pair and Matrix-Vector Multiplication
- Huge matrix data (2D: 34 GB, 3D: 2 PB)

true image



Fourier kernel

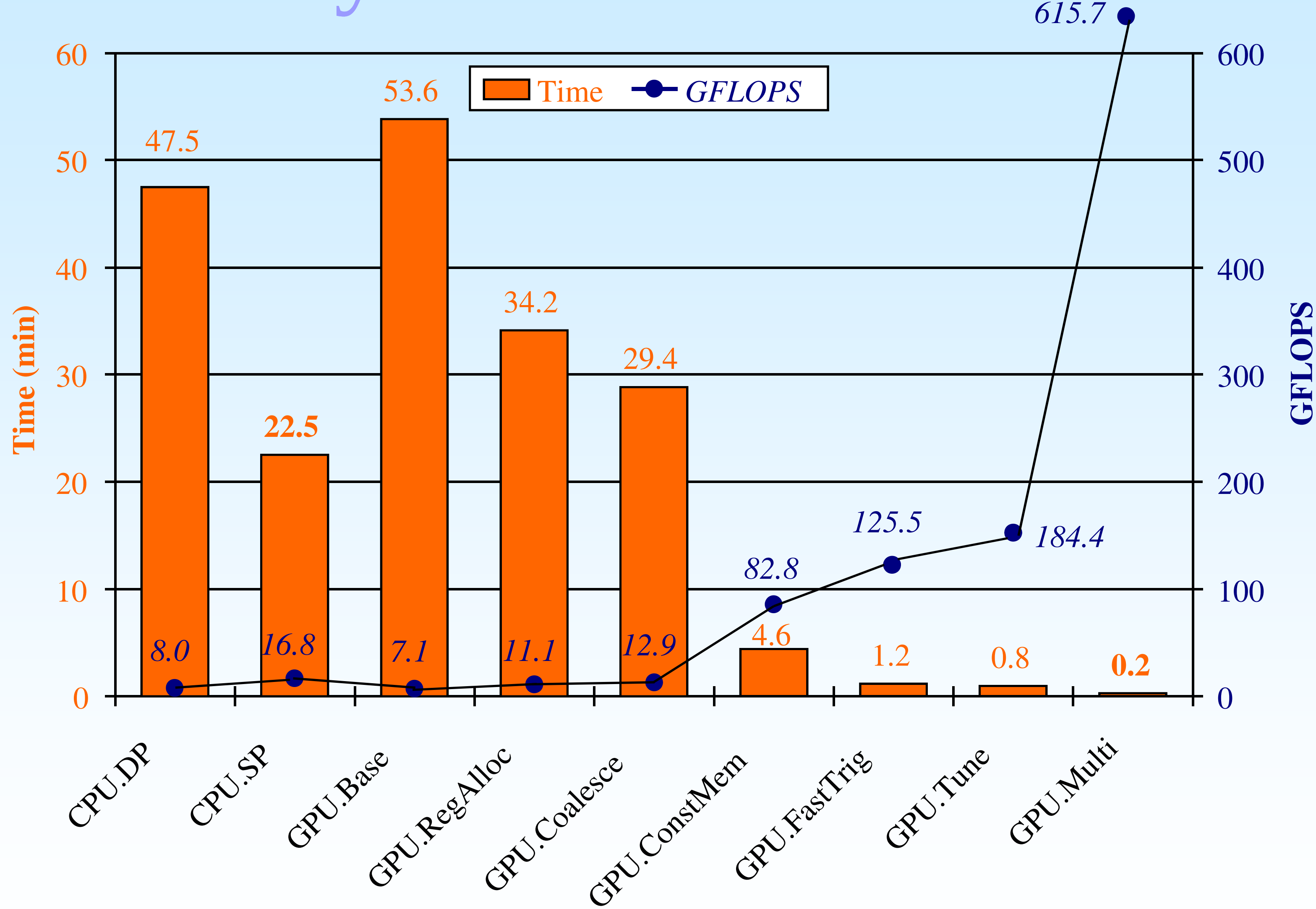




data samples

noise

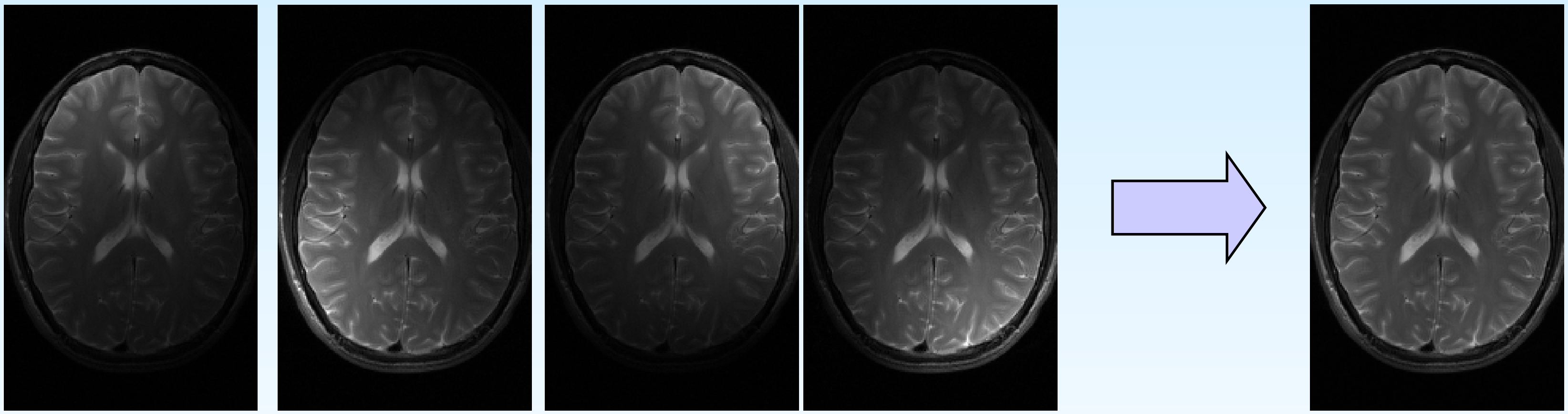
Summary of Results



S. Stone *et al., J Parallel Distrib Comput* 68:1307-1318, 2008.

MR Reconstruction

- Parallel Imaging
  - Data acquired with multiple spatially diverse sensors



30X speedup

- Field inhomogeneity correction

95X speedup

GPUs in MRI

- Common MRI computations have been accelerated by orders-of-magnitude using GPUs
  - Enables more practical use of advanced reconstruction algorithms to reduce scan time/image artifacts
  - Key primitives: 3D convolution, 3-D histogram, sparse/Toeplitz matrix-vector multiplication, sparse CG solver, and FFT
- Current challenge: To develop a common, modular framework for GPU reconstruction of MR data (and other imaging modalities)
  - Single framework for multi-core CPUs and many-core GPUs
  - Automatic tuning and selection for each primitive
- Future work:
  - Continued optimization, scaling of reconstruction algorithms
  - GPU implementation of MR parameter estimation
  - Support for integration into production MRI pipelines

CUDA – C

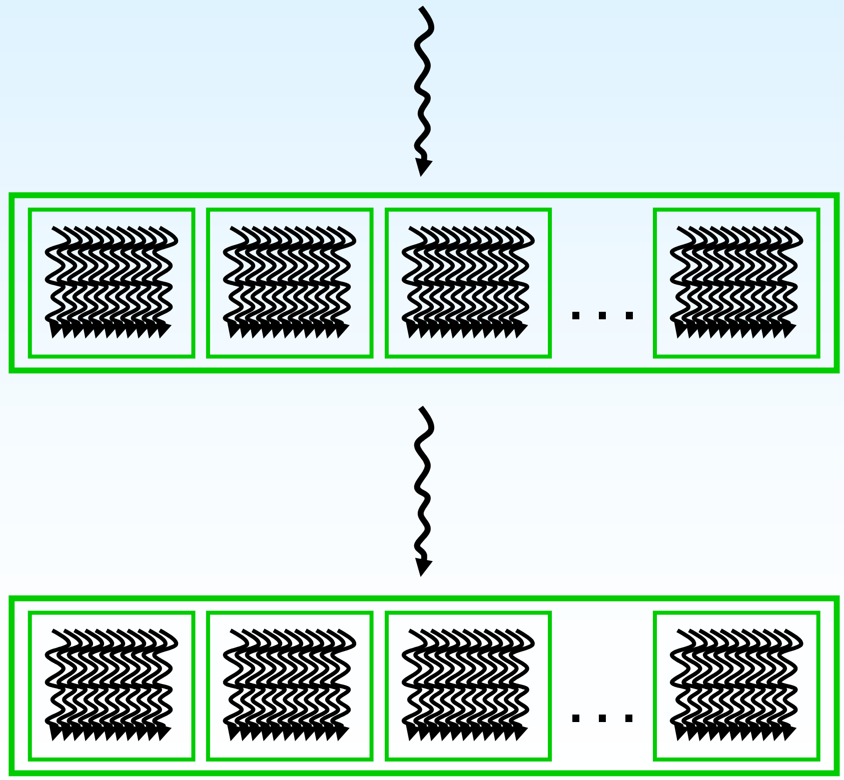
- Integrated host+device app C program
  - Serial or modestly parallel parts in **host** C code
  - Highly parallel parts in **device** SPMD kernel C code

Serial Code (host)

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

Serial Code (host)

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

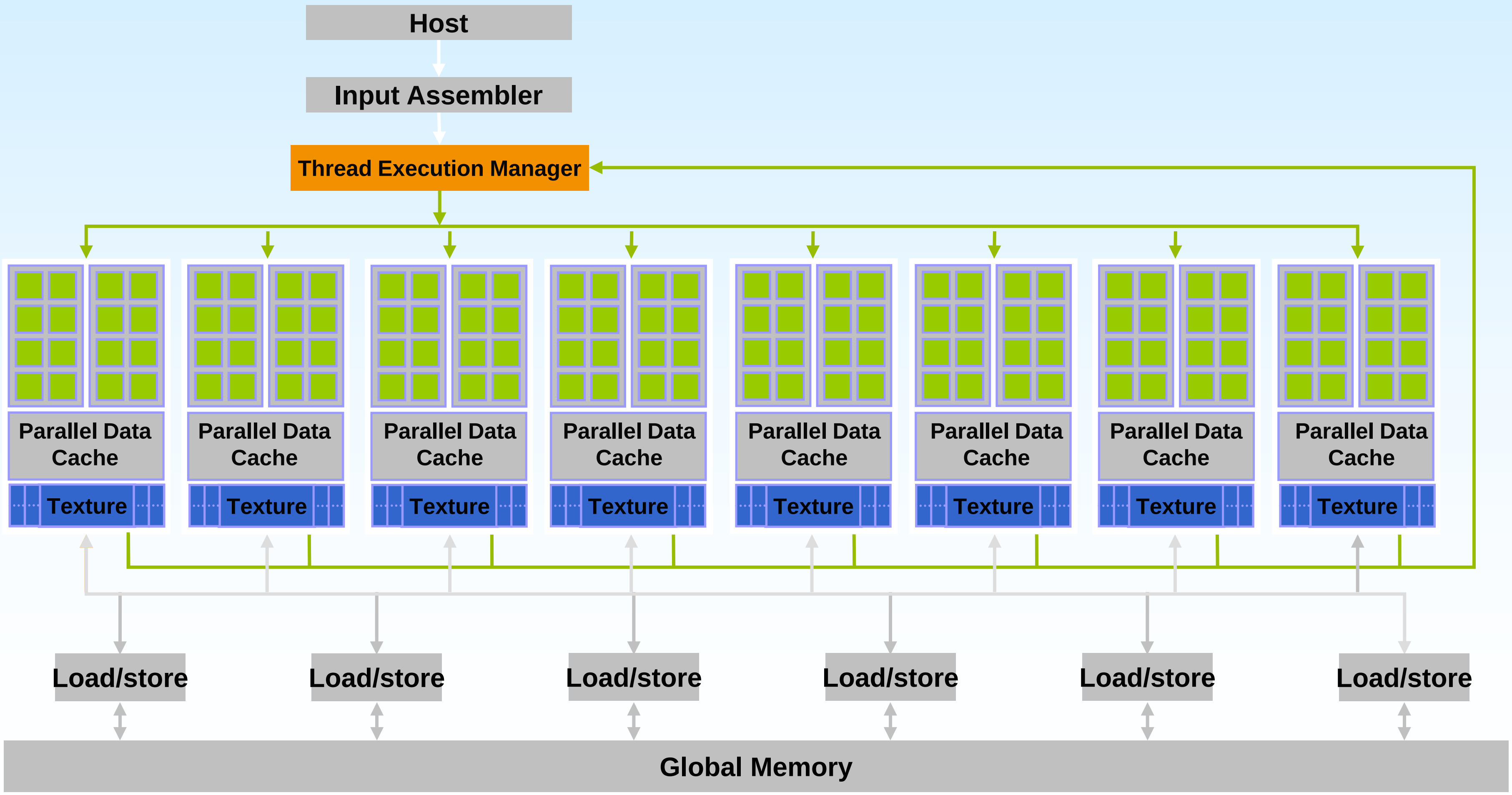


CUDA Devices and Threads

- 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 CUDA mode – A **Device** Example

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



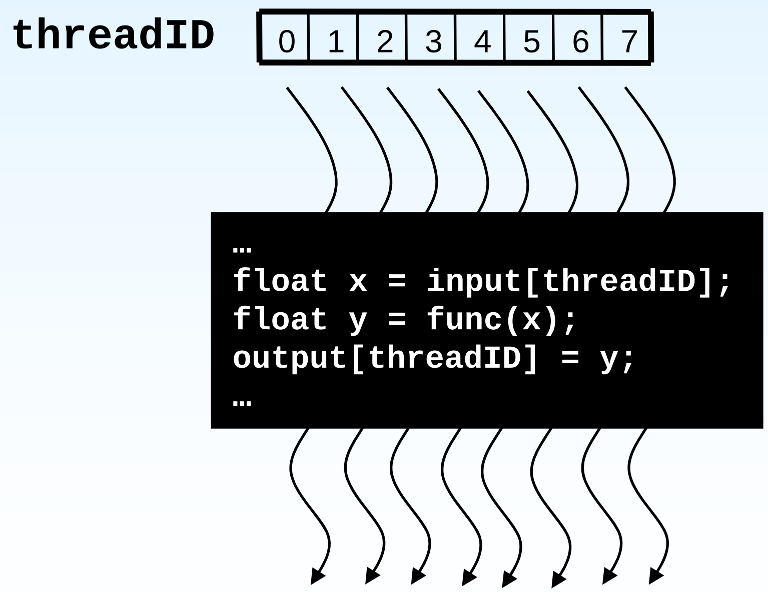
CUDA Extends C

- **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);
```

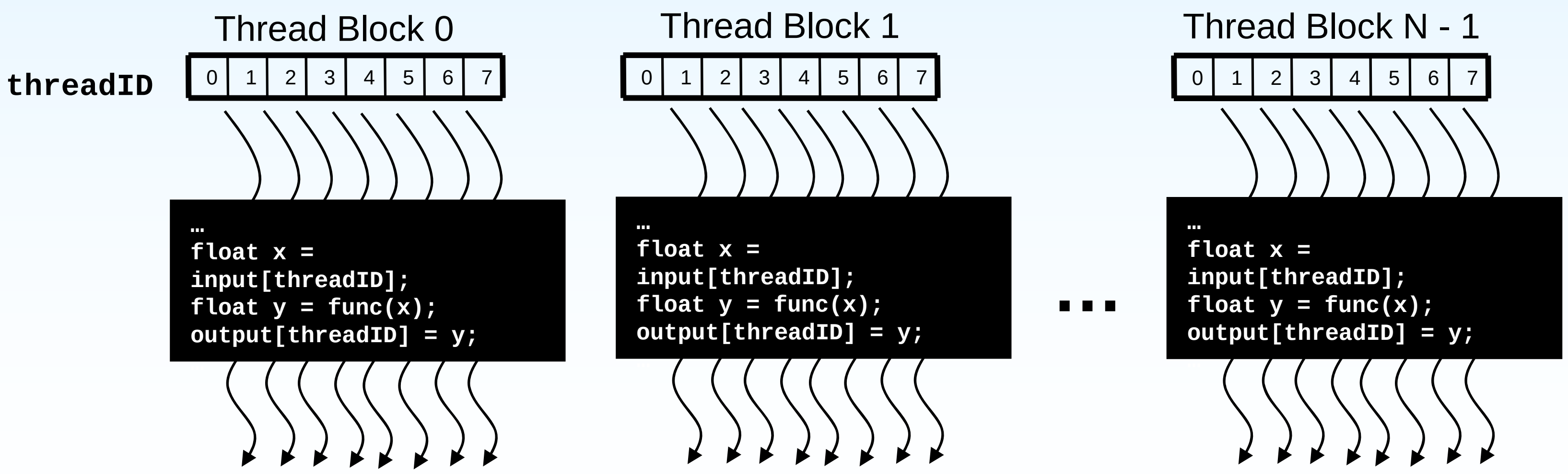
Arrays of Parallel Threads

- 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



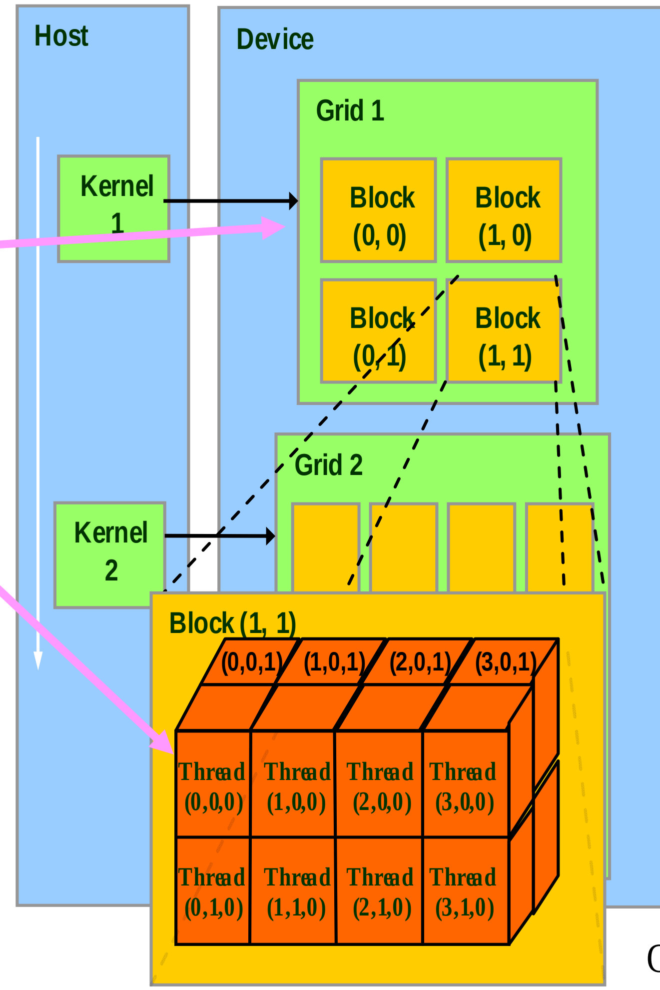
Thread Blocks: Scalable Cooperation

- 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

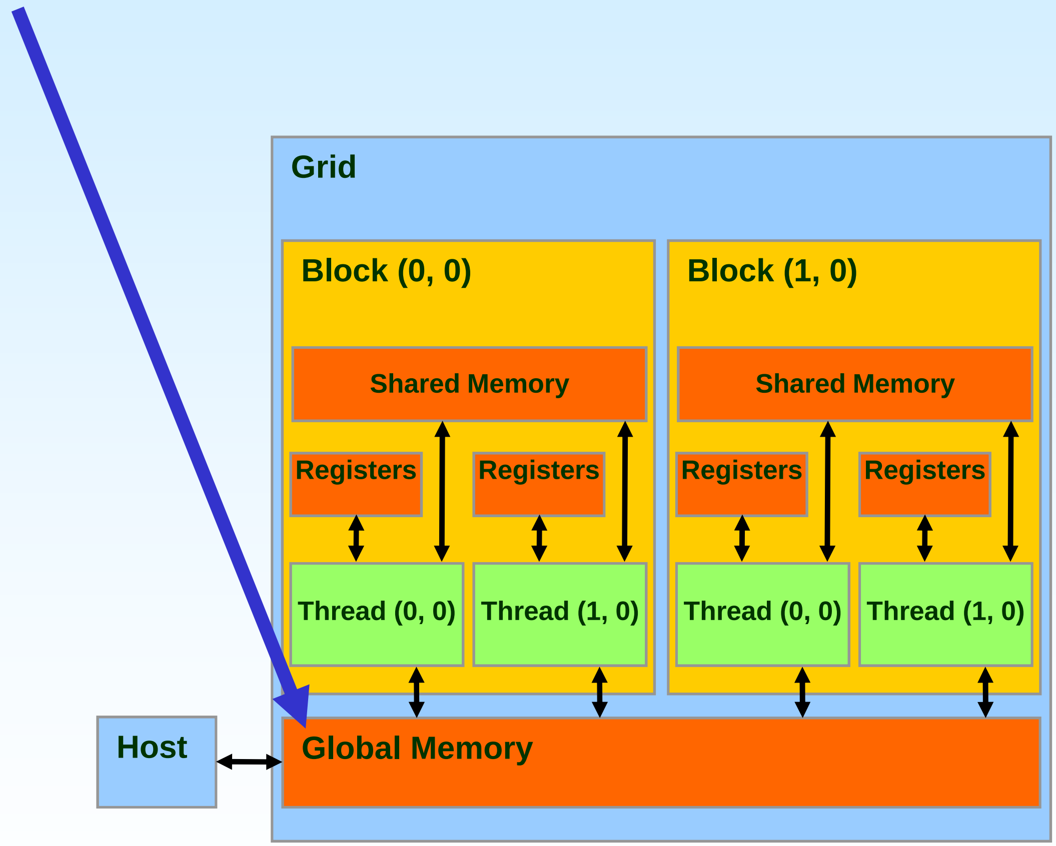
- All threads in a block execute the same kernel program (SPMD)
- Programmer declares block:
  - Block size 1 to **512** concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads
  - Blocks have Block id (X and Y)
- Threads have thread id (X,Y,Z) numbers within block
  - Thread program uses thread id to select work and address shared data
- Each block can execute in any order relative to other blocks
- **CUDA Kernel is a mapping from data parallel computations onto Block id and thread id!**



Courtesy: NDVIA

CUDA Memory Model Overview

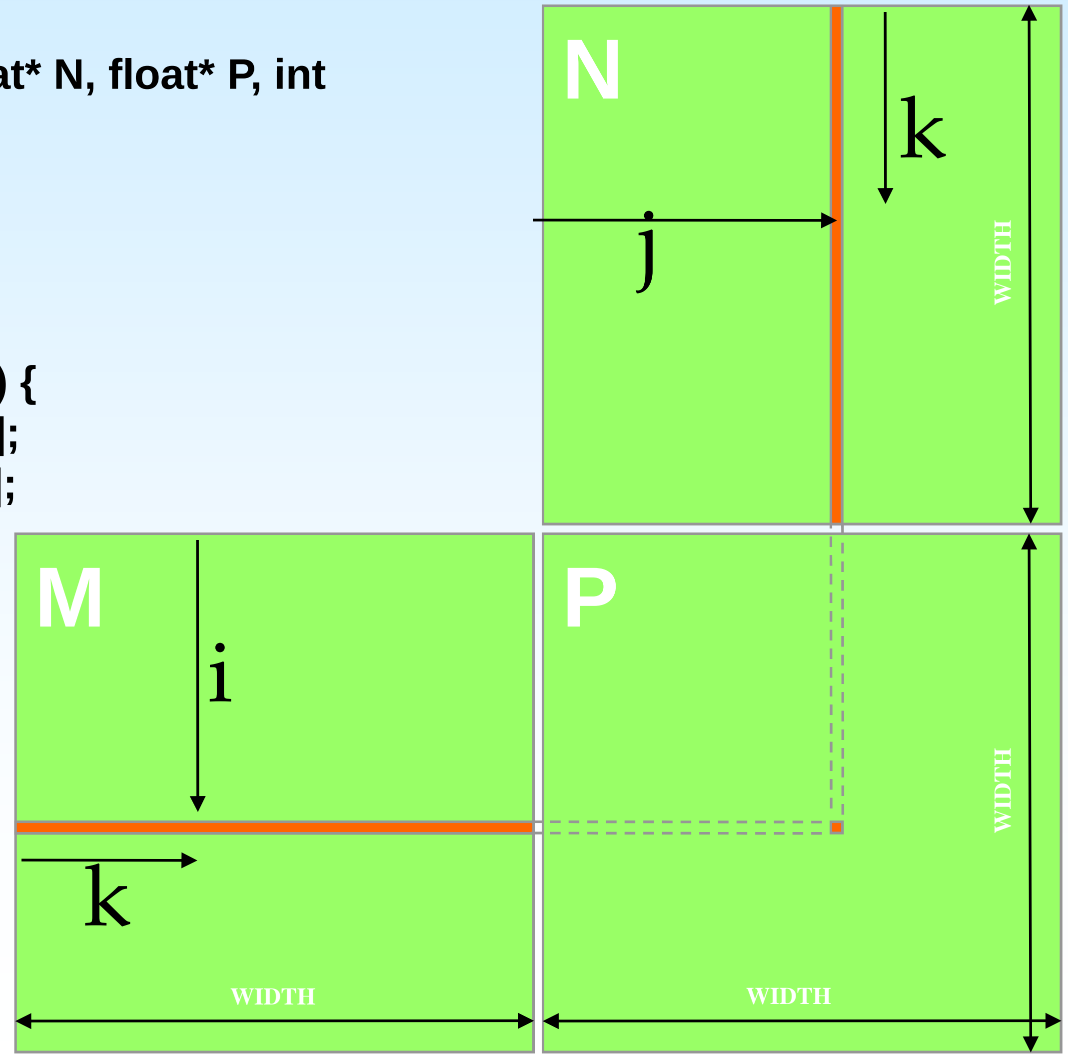
- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
  - Long latency access



Example: Matrix Multiplication

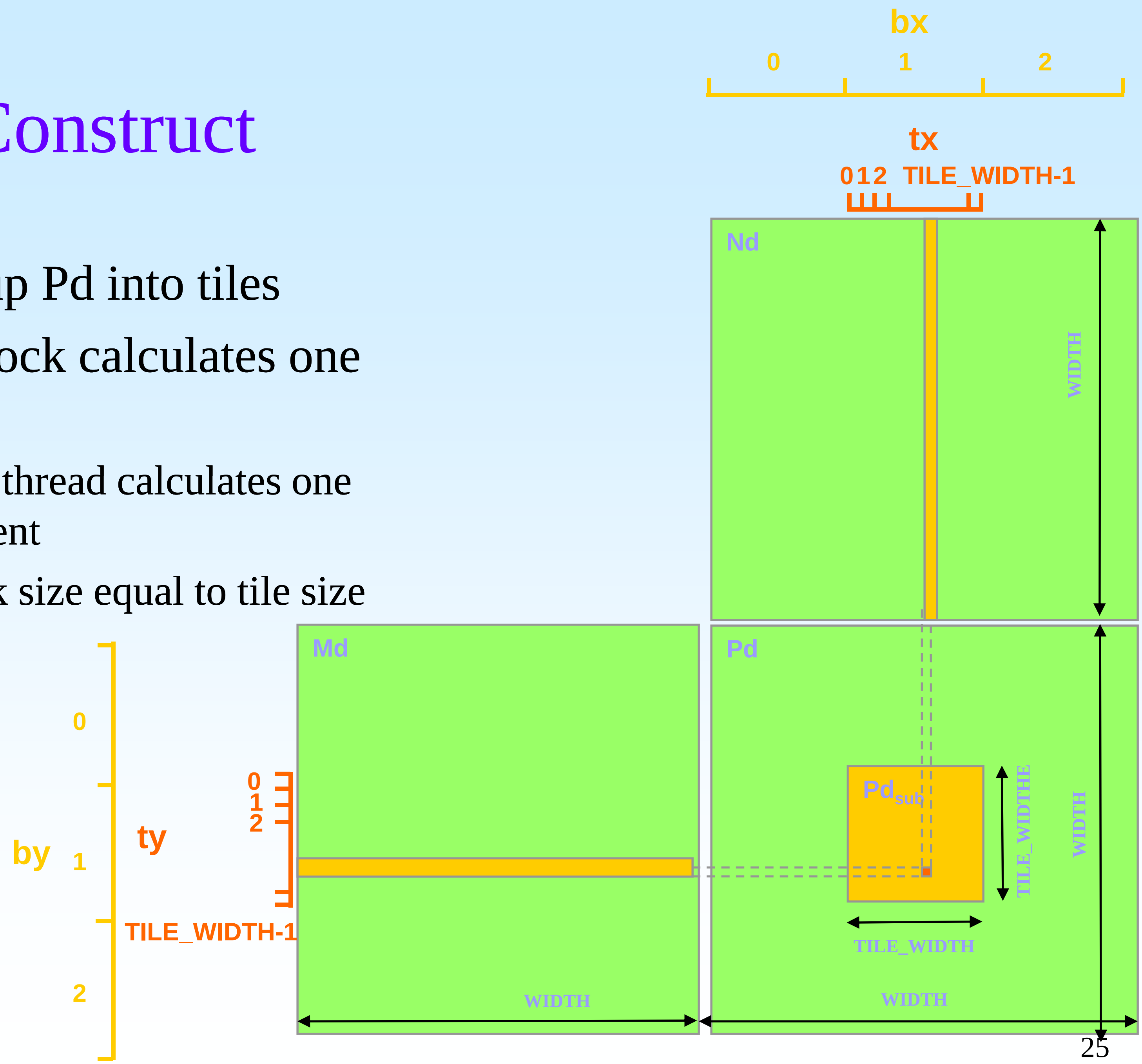
// Matrix multiplication on the (CPU) host in double precision

```
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;
        }
}
```



Tiled Construct

- Break-up Pd into tiles
- Each block calculates one tile
  - Each thread calculates one element
  - Block size equal to tile size

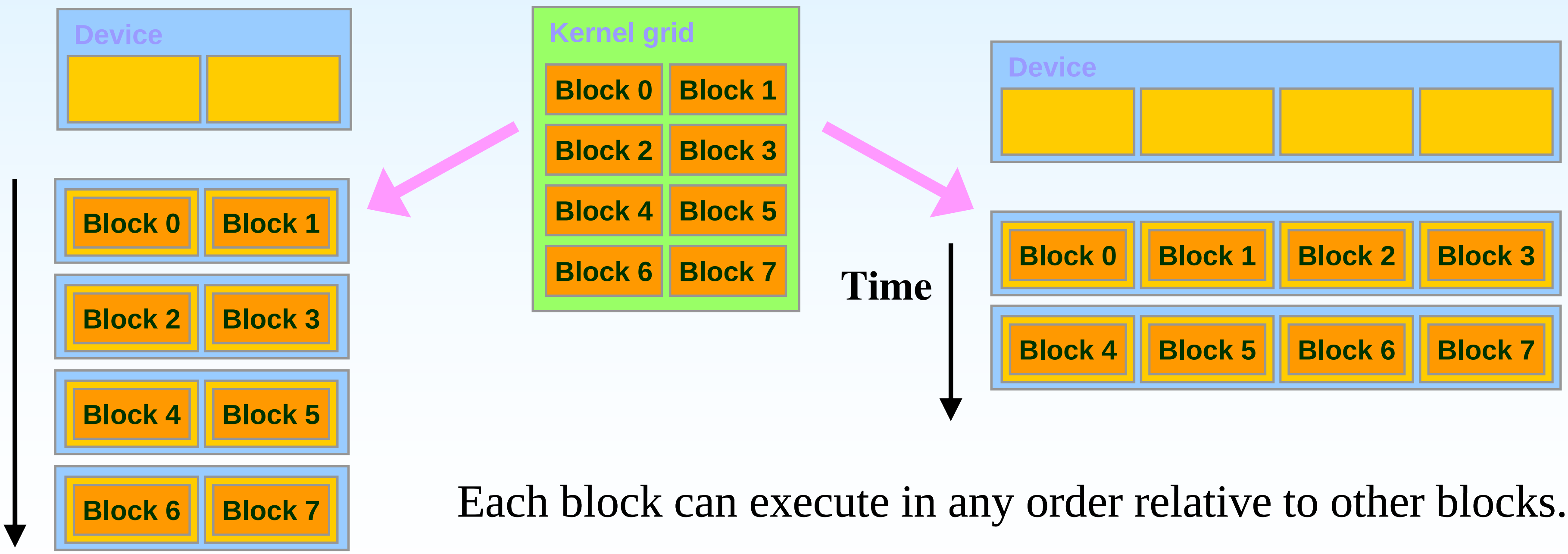


Kernel

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
// Calculate the row index of the Pd element and M
int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
// Calculate the column idenx of Pd and N
int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

float Pvalue = 0;
// each thread computes one element of the block sub-matrix
for (int k = 0; k < Width; ++k)
  Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

Pd[Row*Width+Col] = Pvalue;
}
```

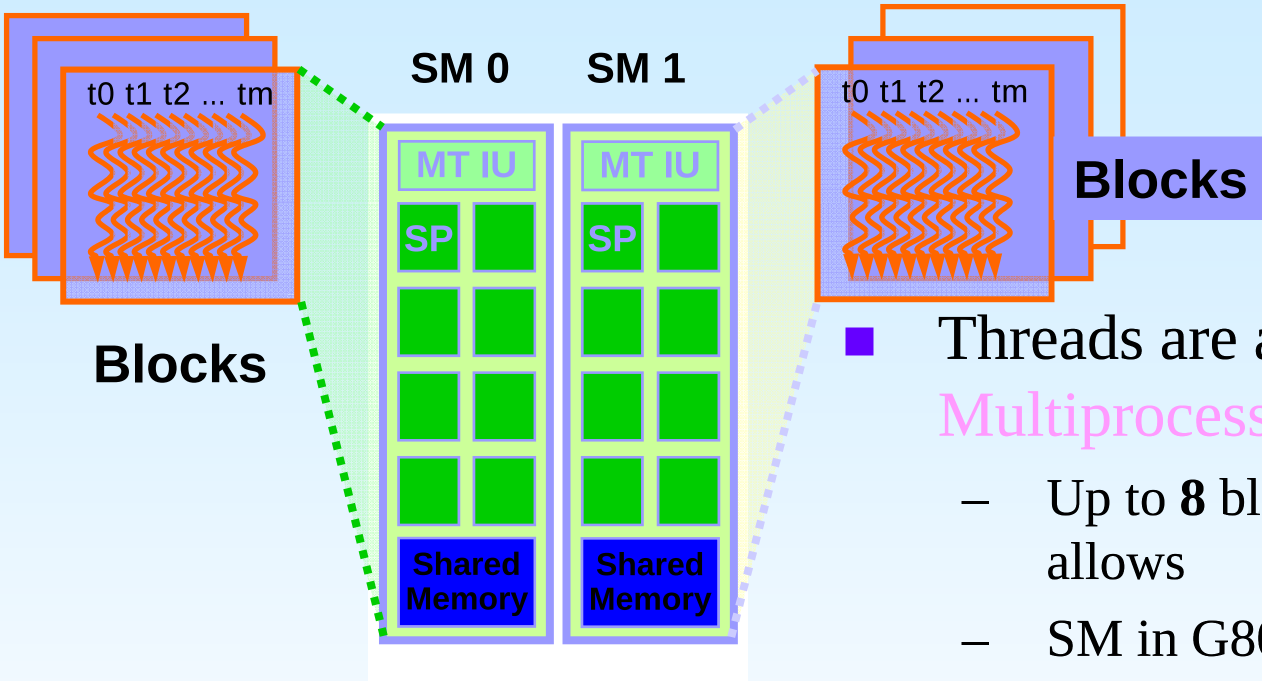
Transparent Scalability

- Hardware is free to assigns blocks to any processor at any time
  - A kernel scales across any number of parallel processors



Each block can execute in any order relative to other blocks.

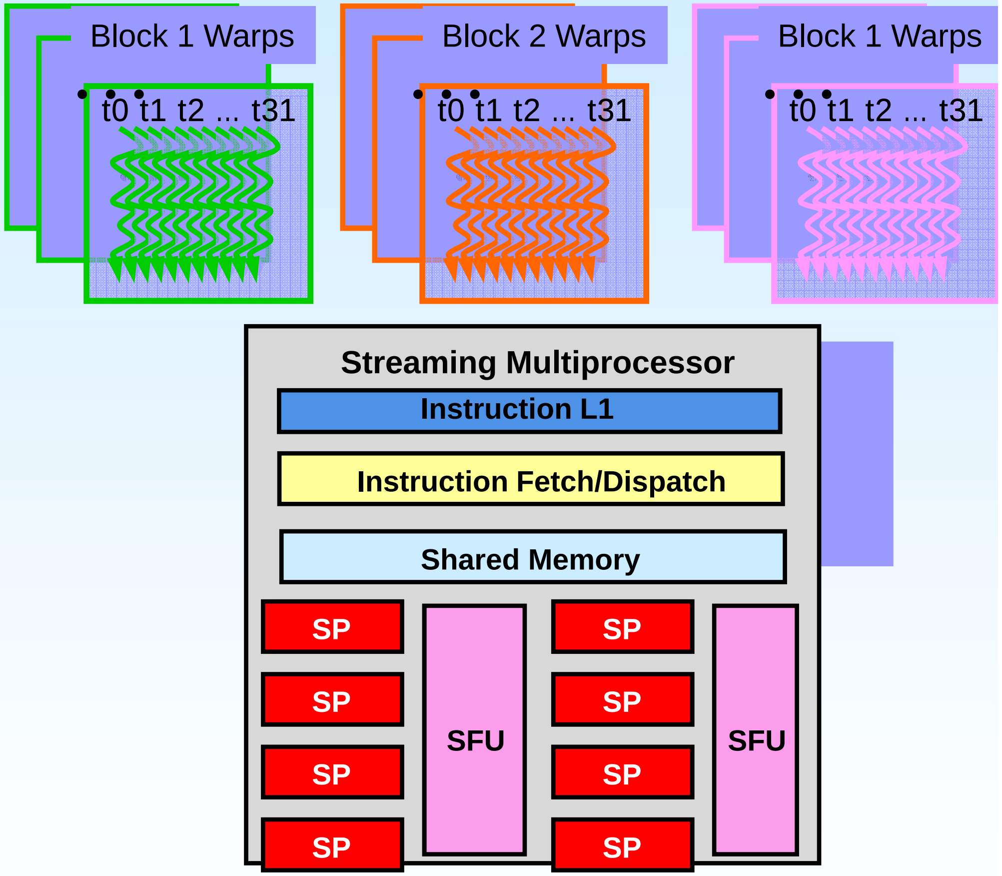
G80 Execution Model



- Threads are assigned to Streaming Multiprocessors in block granularity
  - Up to **8** blocks to each SM as resource allows
  - SM in G80 can take up to **768** threads
    - Could be 256 (threads/block) * 3 blocks
    - Or 128 (threads/block) * 6 blocks, etc.
- Threads run concurrently
  - SM maintains thread/block id #s
  - SM manages/schedules thread execution

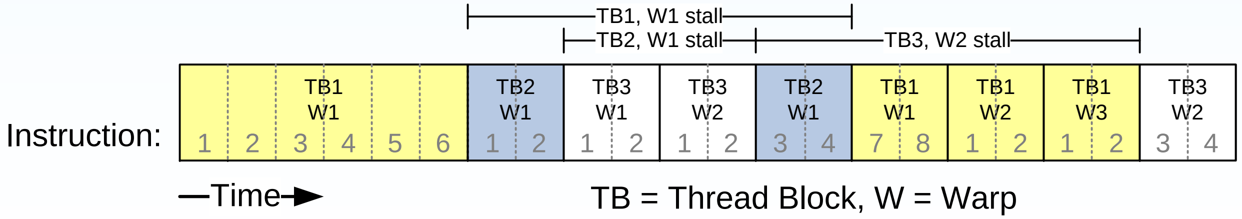
G80: Thread Scheduling

- The threads of each Block are executed as 32-thread Warps (Instr. broadcast to 8 cores in 4 cycles)
  - An implementation decision, not part of the CUDA programming model
  - Warps are scheduled in SM
- If 3 blocks are assigned to an SM and each block has 256 threads, how many Warps are there in an SM?
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps
- Steaming Multiprocessor (Thread-Level Parallelism (TLP))
  - SM performs 18 FLOPs/Cycle
  - SM has 8K registers
  - When one warp stalls (Mem. or arith.), the SM quickly switches to a ready warp from same or another block
  - On-chip memories are used to promote data locality and sharing
  - Cache is single-ported, accesses of different addresses yield stalls



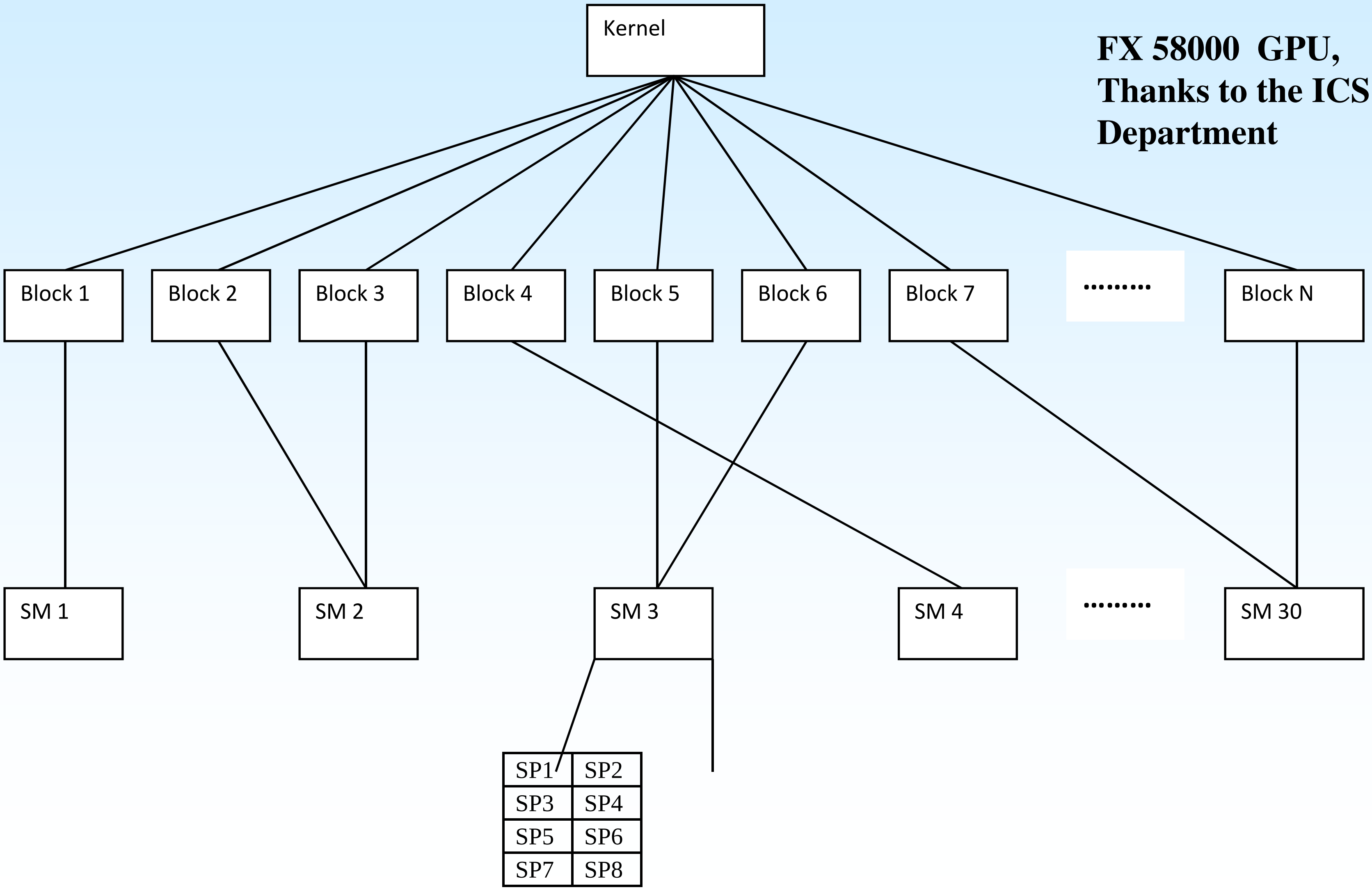
G80: Thread Scheduling (Cont.)

- Hiding latency of GM and some Arithmetics: SM implements zero-overhead warp scheduling
  - At any time, only one of the warps is executed by SM
  - Warps whose next instruction has its operands 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
  - Coalesced access to SM significantly reduce access time



Performance of MM and Jacobi

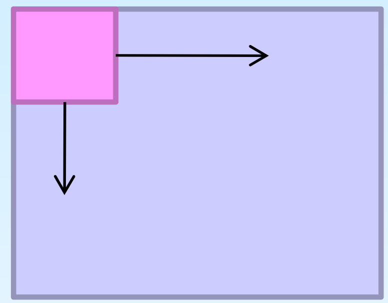
FX 58000 GPU, Thanks to the ICS Department



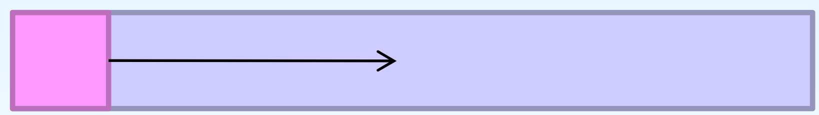
N-Blocks Over 1-Block Experiments

- 1-Block Execution:
  - Single kernel block will compute the whole matrix/array.
  - Traverses in small blocks to complete the whole matrix/array.
  - Each thread within the block compute (N*N)/256 elements (for matrix multiply) and N/16 elements (for jacobi)
- N-Block Execution:
  - Whole resultant matrix/array is divided into number of blocks 2,4,8, …, 16384
- 1 Block = 256 threads (16x16) for MM.
- 1 Block = 16 threads for Jacobi.

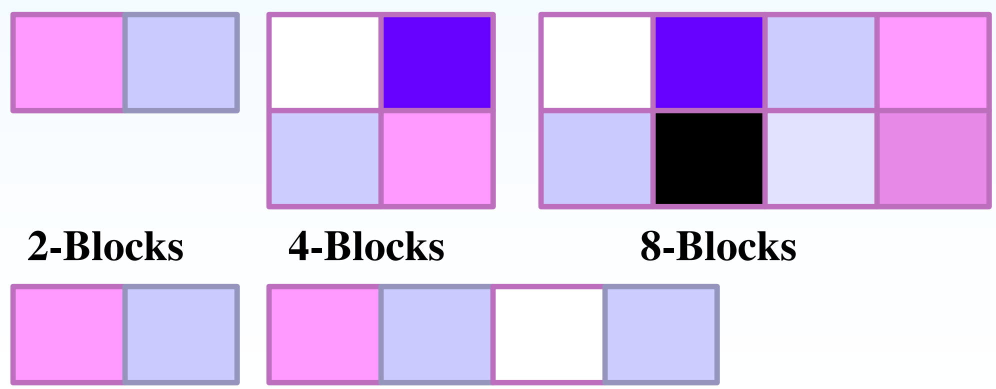
Matrix Multiply



Jacobi

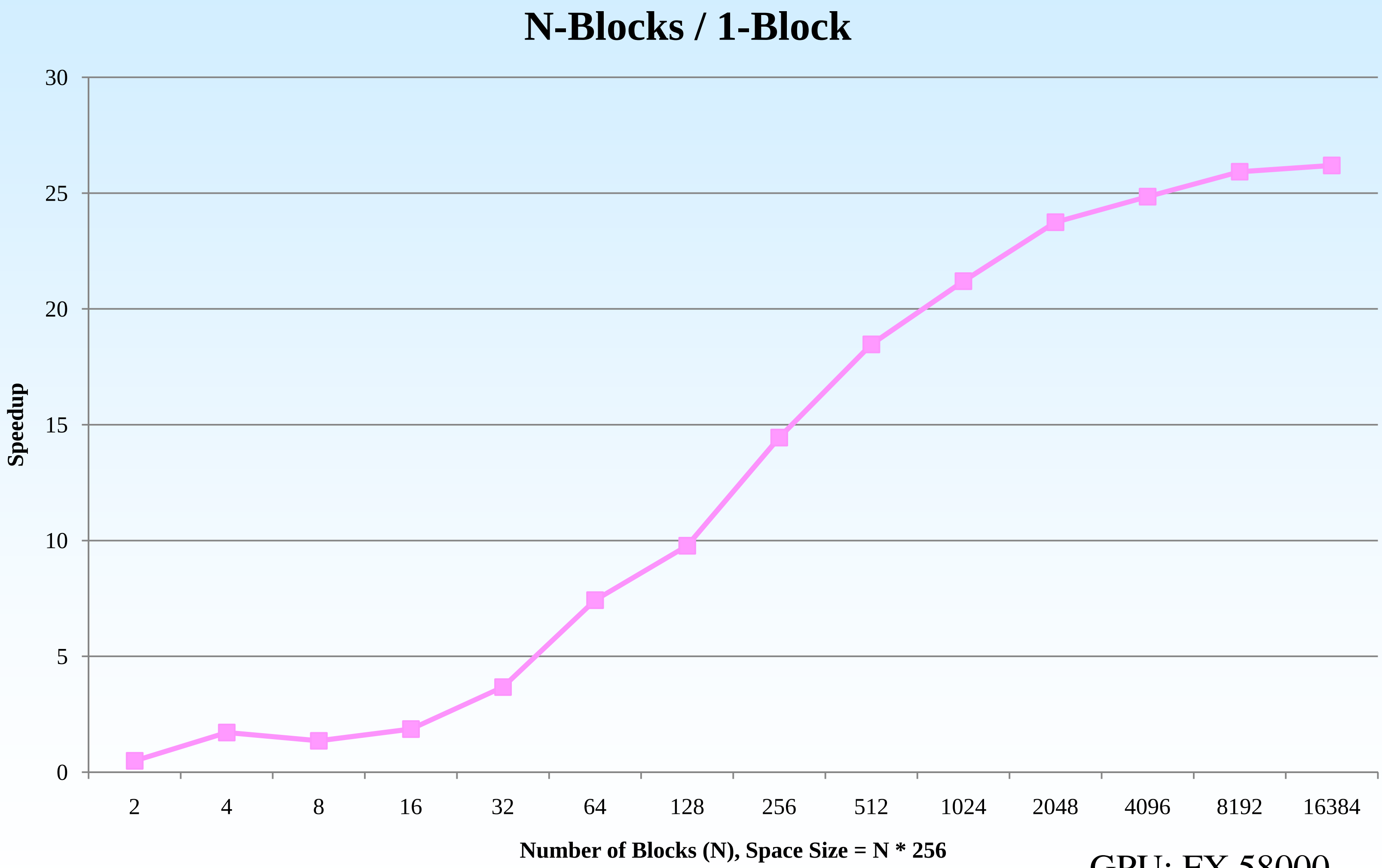


Matrix Multiply



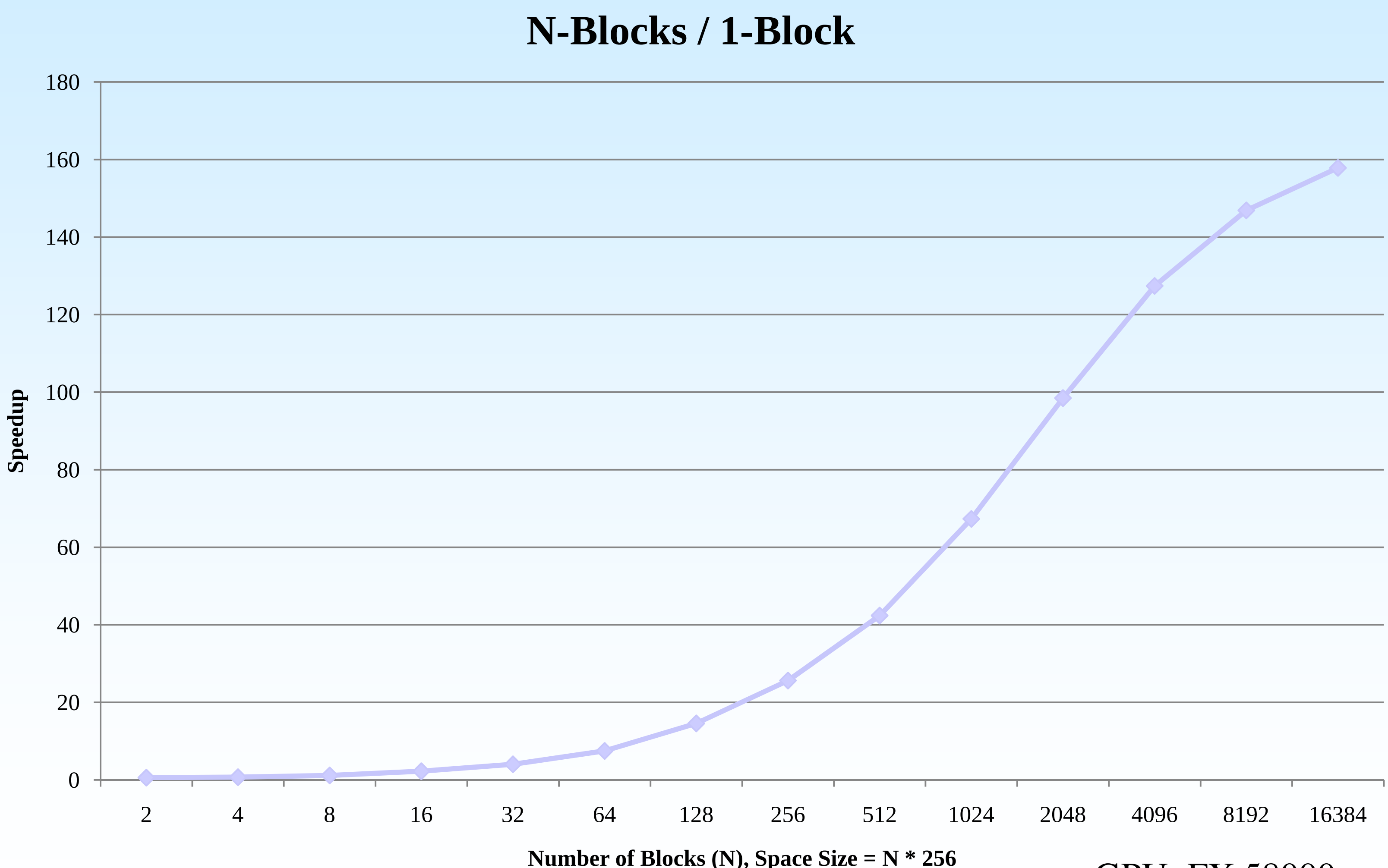
Jacobi

Matrix Multiply (only GM)



GPU: FX 58000

Matrix Multiply (With ShM)



GPU: FX 58000

Note: Significant Increase in Speedup is also due to Coalesced Memory Access

Coalesced Memory Access

- The size of the memory element accessed by each thread is either 4, 8, or 16 bytes
- The elements form a contiguous block of memory
- The Nth element is accessed by the Nth thread in the half-warp, does not affect if any thread in between not accessing the global memory that is divergent warp.
- The address of the first element is aligned to 16 times the element's size

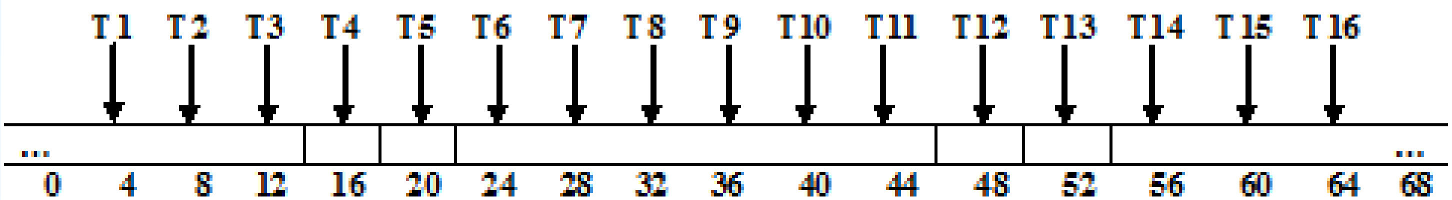


Figure 1: Coalesced float Memory Access

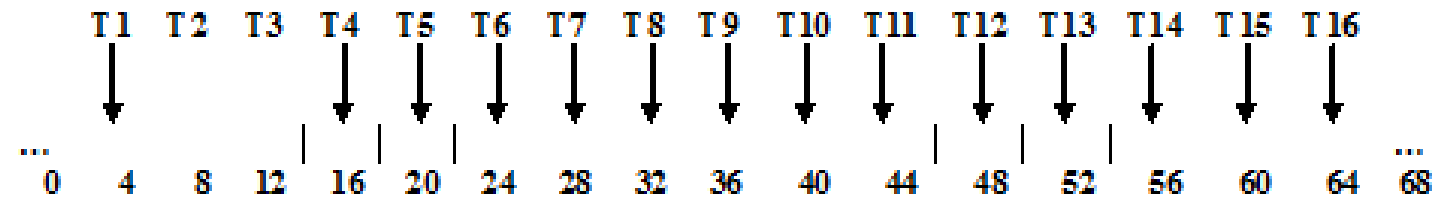
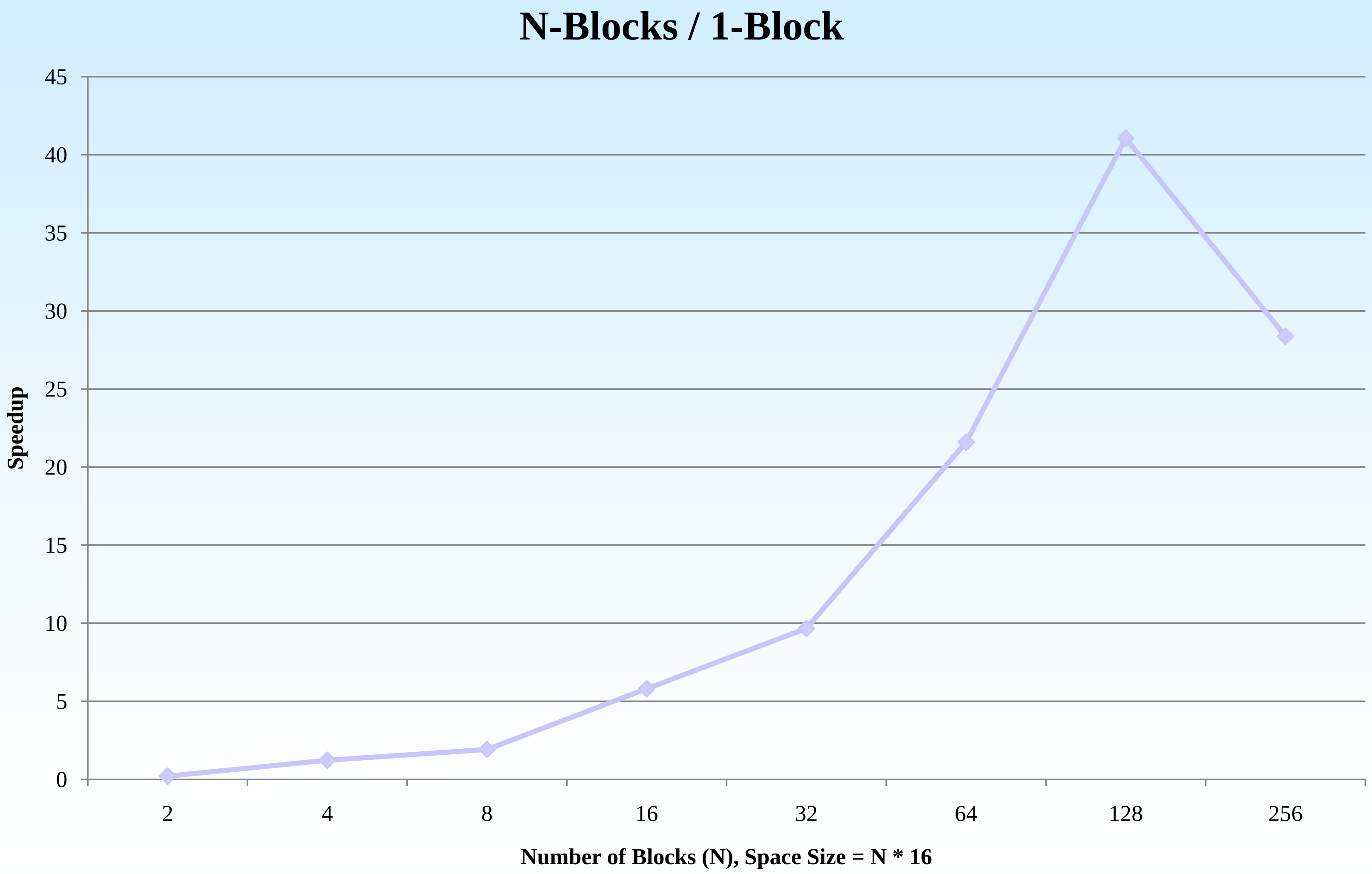
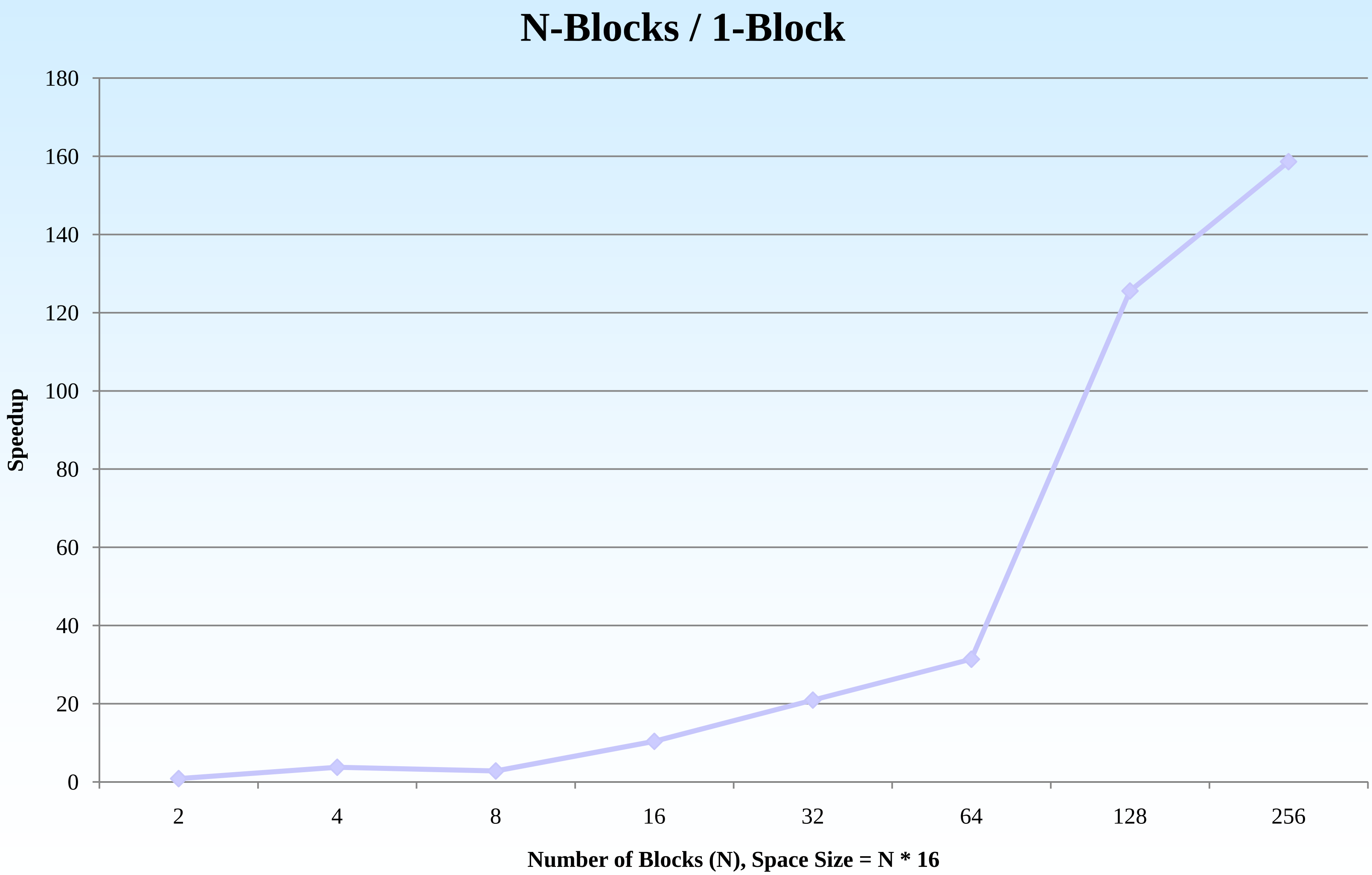


Figure 2: Coalesced float Memory Access (divergent Warp)

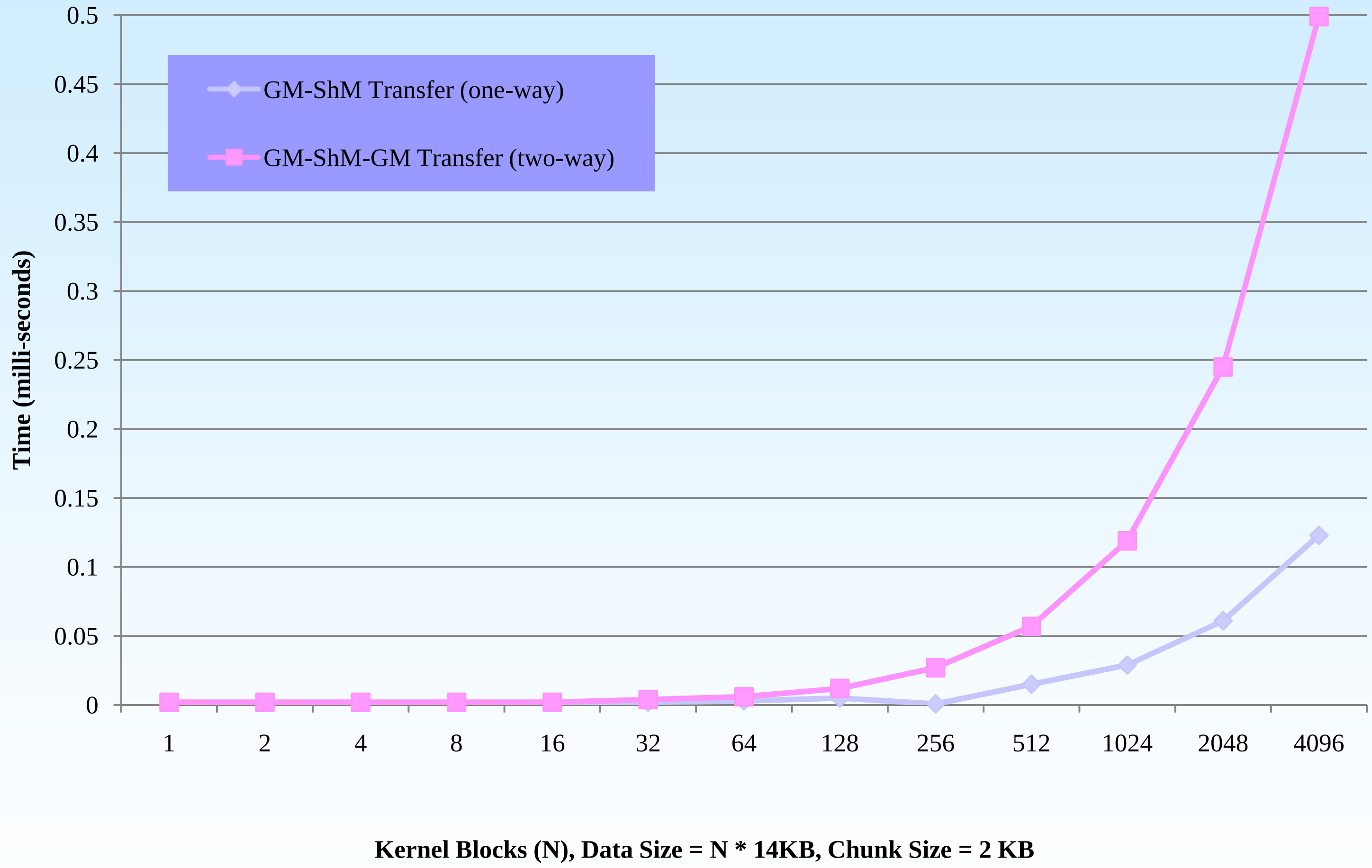
Jacobi (only GM)



Jacobi (With ShM)



Benchmarking data copy



Research Directions

- Programming GPGPU require an expert level understanding of the memory hierarchy and execution model to reach peak performance.
- Even for experts, rewriting a program to exploit the architecture in achieving high speedup can be tedious and error prone.
- Compilers and their ability to make code transformations can assist in parallel programming of large scale GPGPU applications as well as handling many of the target specific details.
- A source to source compiler transformation and code generation framework is needed for the parallelization and optimization of computations expressed in sequential loop nests for running on many-core GPUs.
- May use a complete scripting language to describe composable compiler transformations that can be written, shared and reused by non expert application and library developers.
- A research framework must exhibit high-performance on standard benchmarks that show it capable of matching or outperforming hand-tuned GPU kernels.

Research Directions

Some research directions:

- Automating the GPU kernel generation including computation partitioning, allocating memory for GPU I/O, GPU copying data, and performing block and thread decomposition.

- Compiler transformations can be applied in the decomposition and mapping process, and in subsequently optimizing the kernel code to manage the memory hierarchy and parallelism tradeoffs.

- Since there is significant performance variation on GPUs for very subtle differences in code, there is need to explore a space of different implementations, and different values of parameters associated with the mapping.

- A programming tool may support both automated compiler optimization as well as programmer-guided optimization.

Thank you