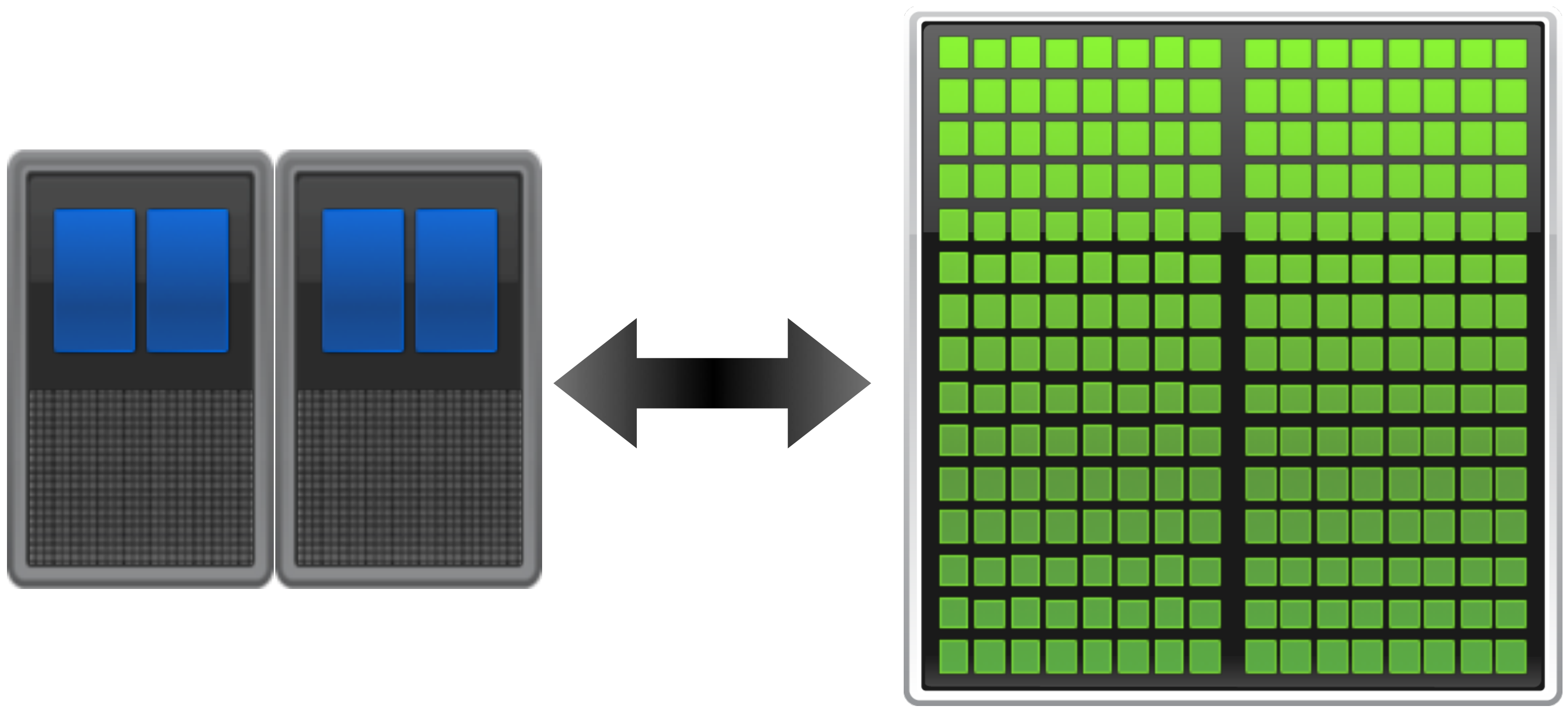
An Introduction to CUDA/OpenCL and Manycore Graphics Processors

Bryan Catanzaro, NVIDIA Research

Overview

- Terminology: Multicore, Manycore, SIMD
- The CUDA and OpenCL programming models
- Mapping CUDA to Nvidia GPUs
- OpenCL

Heterogeneous Parallel Computing



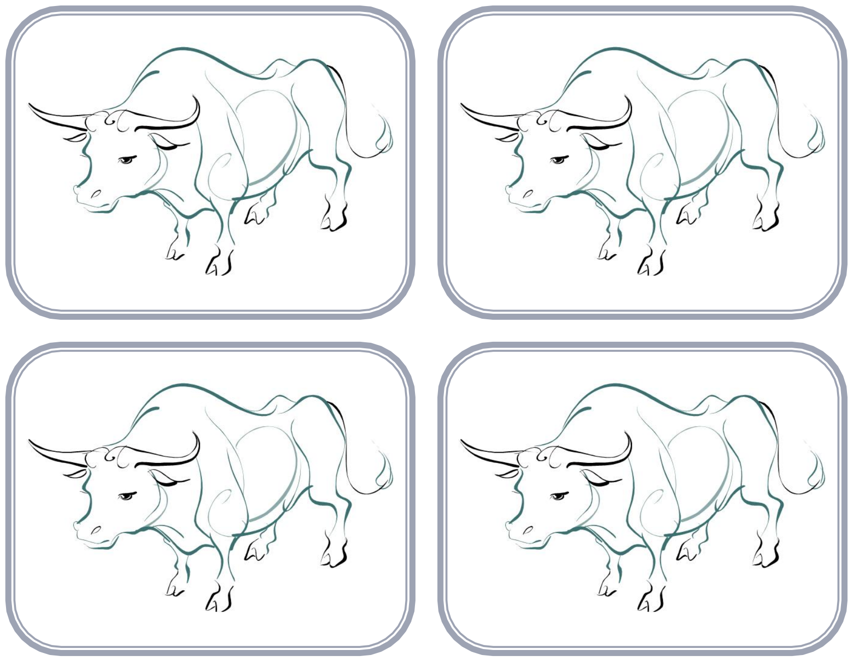
Multicore CPU

Fast Serial Processing

Manycore GPU

Scalable Parallel Processing

Multicore and Manycore



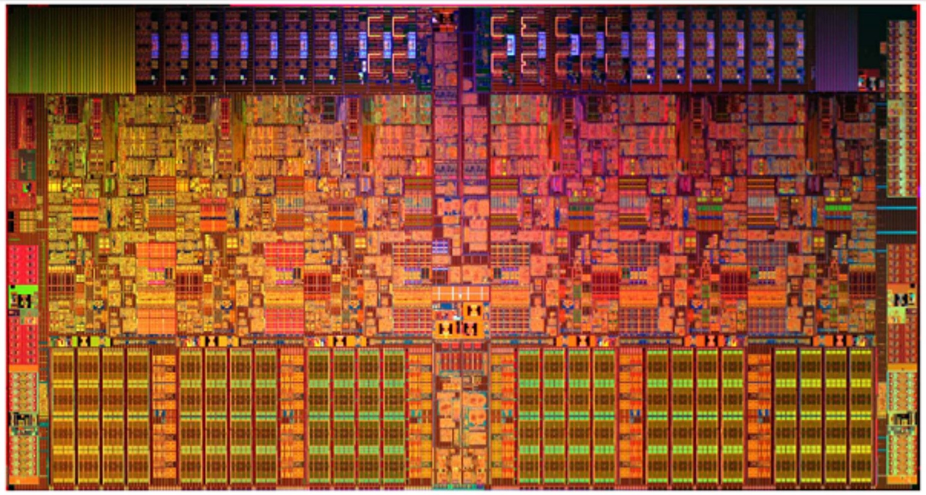
Multicore

Manycore

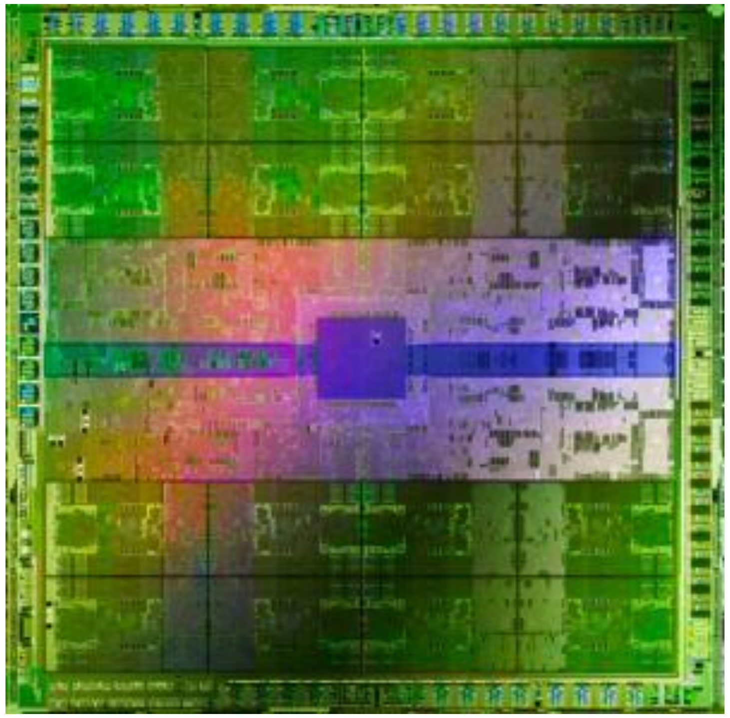
- Multicore: yoke of oxen
  - Each core optimized for executing a single thread
- Manycore: flock of chickens
  - Cores optimized for aggregate throughput, deemphasizing individual performance

Multicore & Manycore, *cont.*

Specifications	Westmere-EP	Fermi (Tesla C2050)
Processing Elements	6 cores, 2 issue, 4 way SIMD @3.46 GHz	14 SMs, 2 issue, 16 way SIMD @1.15 GHz
Resident Strands/ Threads (max)	6 cores, 2 threads, 4 way SIMD: 48 strands	14 SMs, 48 SIMD vectors, 32 way SIMD: 21504 threads
SP GFLOP/s	166	1030
Memory Bandwidth	32 GB/s	144 GB/s
Register File	6 kB (?)	1.75 MB
Local Store/L1 Cache	192 kB	896 kB
L2 Cache	1536 kB	0.75 MB
L3 Cache	12 MB	-



Westmere-EP (32nm)

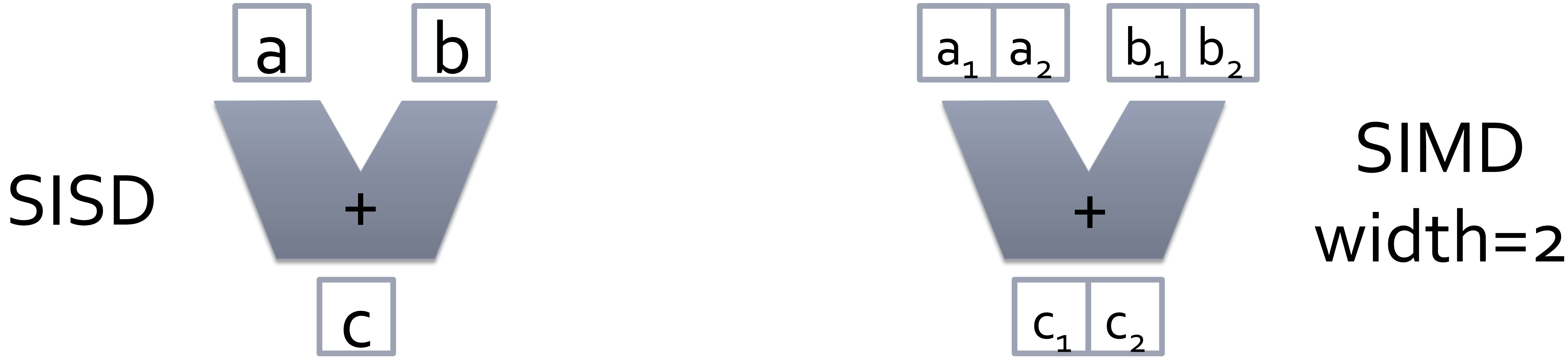


Fermi (40nm)

Why Heterogeneity?

- Different goals produce different designs
  - Manycore assumes work load is highly parallel
  - Multicore must be good at everything, parallel or not
- Multicore: minimize latency experienced by 1 thread
  - lots of big on-chip caches
  - extremely sophisticated control
- Manycore: maximize throughput of all threads
  - lots of big ALUs
  - multithreading can hide latency … so skip the big caches
  - simpler control, cost amortized over ALUs via SIMD

SIMD

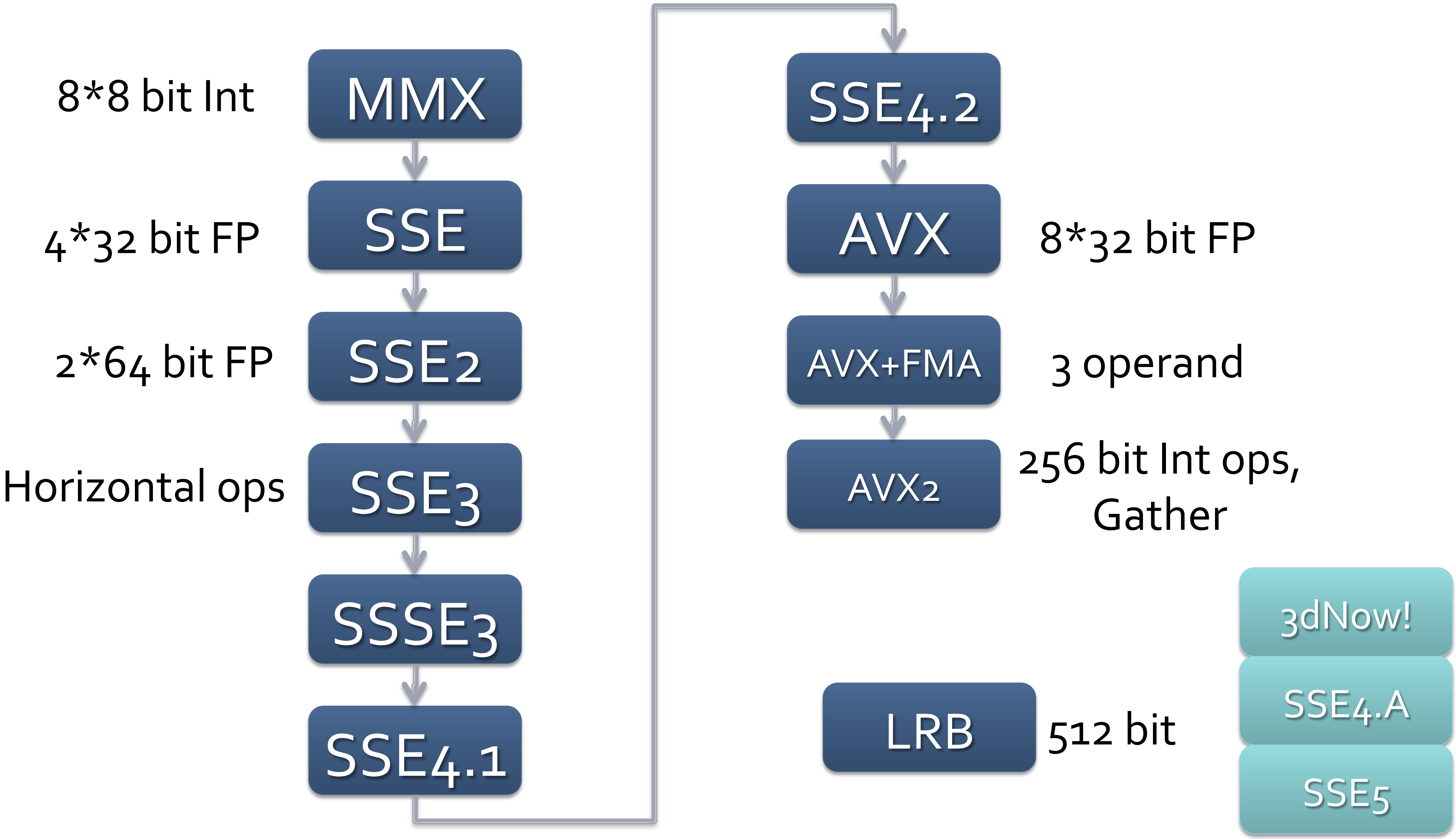


- Single Instruction Multiple Data architectures make use of data parallelism
- We care about SIMD because of area and power efficiency concerns
  - Amortize control overhead over SIMD width
- Parallelism exposed to programmer & compiler

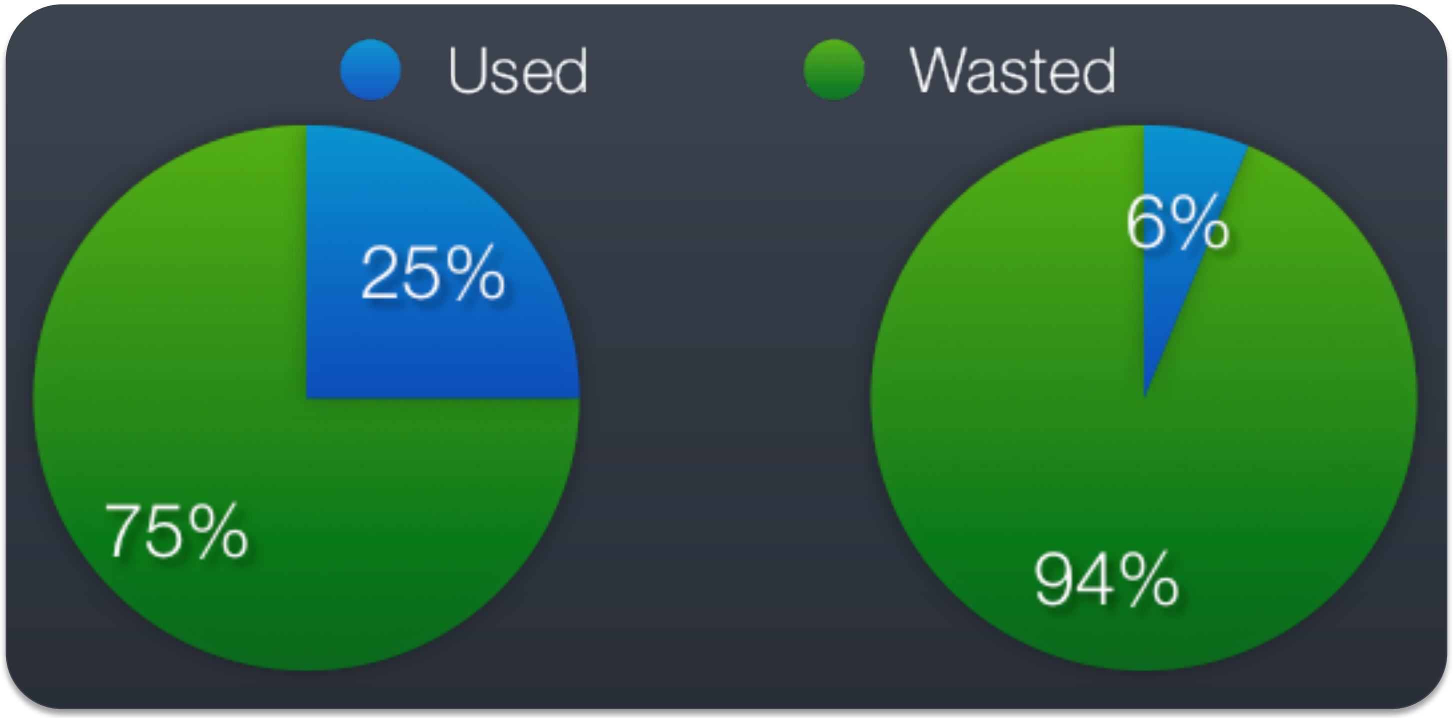
SIMD: Neglected Parallelism

- It is difficult for a compiler to exploit SIMD
- How do you deal with sparse data & branches?
  - Many languages (like C) are difficult to vectorize
  - Fortran is somewhat better
- Most common solution:
  - Either forget about SIMD
    - Pray the autovectorizer likes you
  - Or instantiate intrinsics (assembly language)
  - Requires a new code version for every SIMD extension

A Brief History of x86 SIMD Extensions



What to do with SIMD?



4 way SIMD (SSE) 16 way SIMD (LRB)

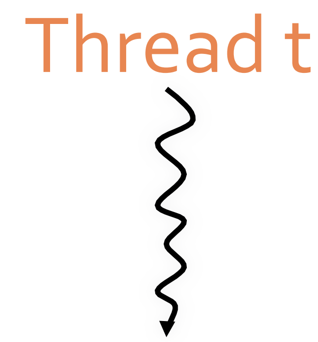
- Neglecting SIMD is becoming more expensive
  - AVX: 8 way SIMD, Larrabee: 16 way SIMD, Nvidia: 32 way SIMD, ATI: 64 way SIMD
- This problem composes with thread level parallelism
- We need a programming model which addresses both problems

The CUDA Programming Model

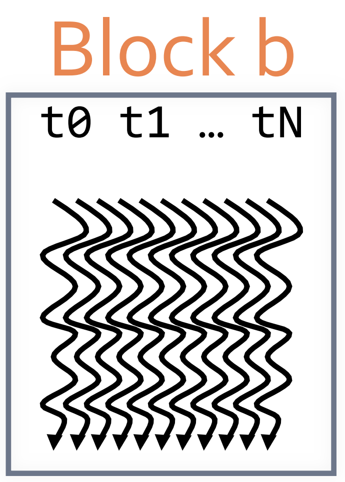
- CUDA is a recent programming model, designed for
  - Manycore architectures
  - Wide SIMD parallelism
  - Scalability
- CUDA provides:
  - A thread abstraction to deal with SIMD
  - Synchronization & data sharing between small groups of threads
- CUDA programs are written in C++ with minimal extensions
- OpenCL is inspired by CUDA, but HW & SW vendor neutral
  - Similar programming model, C only for device code

Hierarchy of Concurrent Threads

- Parallel kernels composed of many threads
  - all threads execute the same sequential program



- Threads are grouped into thread blocks
  - threads in the same block can cooperate



- Threads/blocks have unique IDs

What is a CUDA Thread?

- Independent thread of execution
  - has its own PC, variables (registers), processor state, etc.
  - no implication about how threads are scheduled

- CUDA threads might be physical threads
  - as mapped onto NVIDIA GPUs

- CUDA threads might be virtual threads
  - might pick 1 block = 1 physical thread on multicore CPU

What is a CUDA Thread Block?

- Thread block = a (data) parallel task
  - all blocks in kernel have the same entry point
  - but may execute any code they want
- Thread blocks of kernel must be independent tasks
  - program valid for ***any interleaving*** of block executions

CUDA Supports:

- Thread parallelism
  - each thread is an independent thread of execution
- Data parallelism
  - across threads in a block
  - across blocks in a kernel
- Task parallelism
  - different blocks are independent
  - independent kernels executing in separate streams

Synchronization

- Threads within a block may synchronize with barriers

```
… Step 1 …
__syncthreads();
… Step 2 …
```

- Blocks coordinate via atomic memory operations
  - e.g., increment shared queue pointer with `atomicInc()`
- Implicit barrier between dependent kernels

```
vec_minus<<<nblocks, blksize>>>(a, b, c);
```

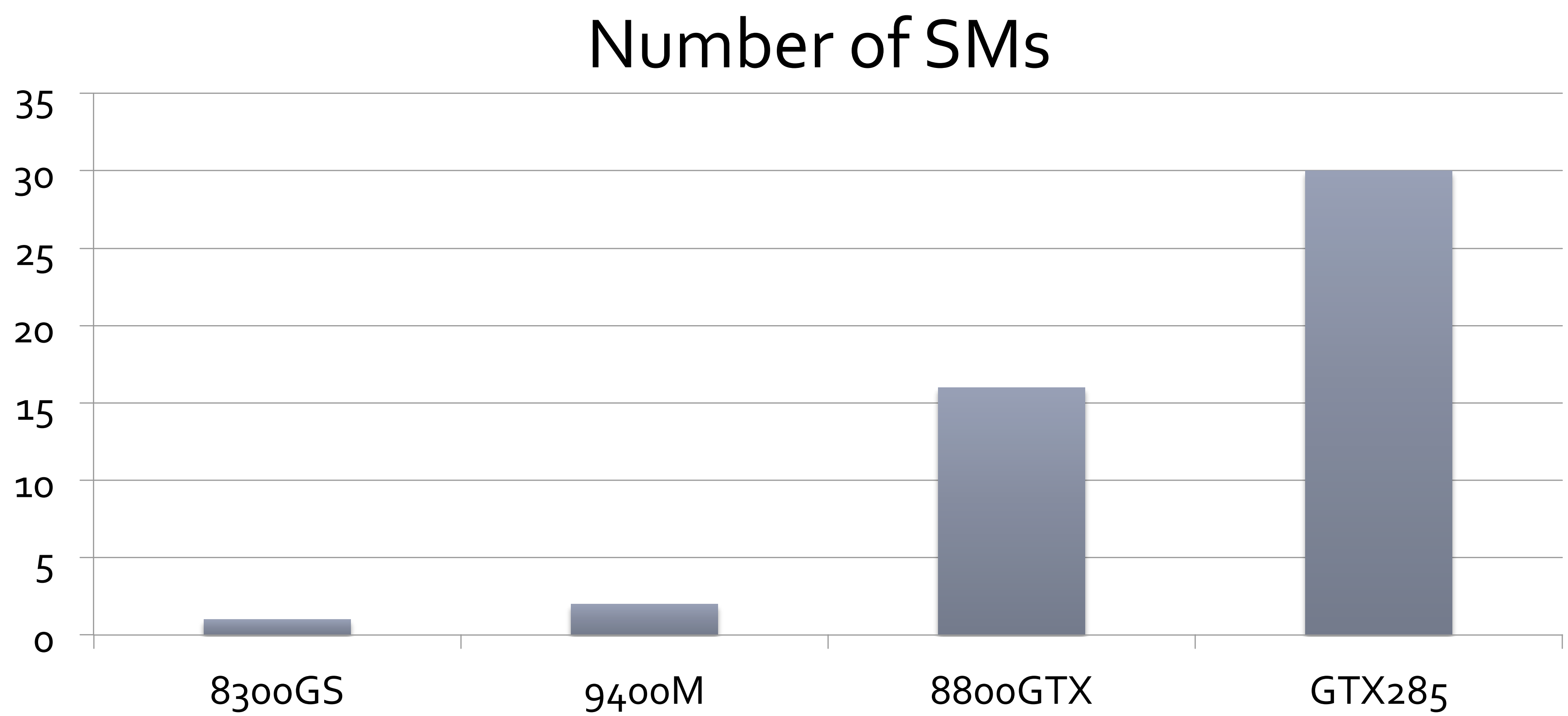
```
vec_dot<<<nblocks, blksize>>>(c, c);
```

Blocks must be independent

- Any possible interleaving of blocks should be valid
  - presumed to run to completion without pre-emption
  - can run in any order
  - can run concurrently OR sequentially
- Blocks may coordinate but not synchronize
  - shared queue pointer: OK
  - shared lock: BAD … can easily deadlock
- Independence requirement gives scalability

Scalability

- Manycore chips exist in a diverse set of configurations

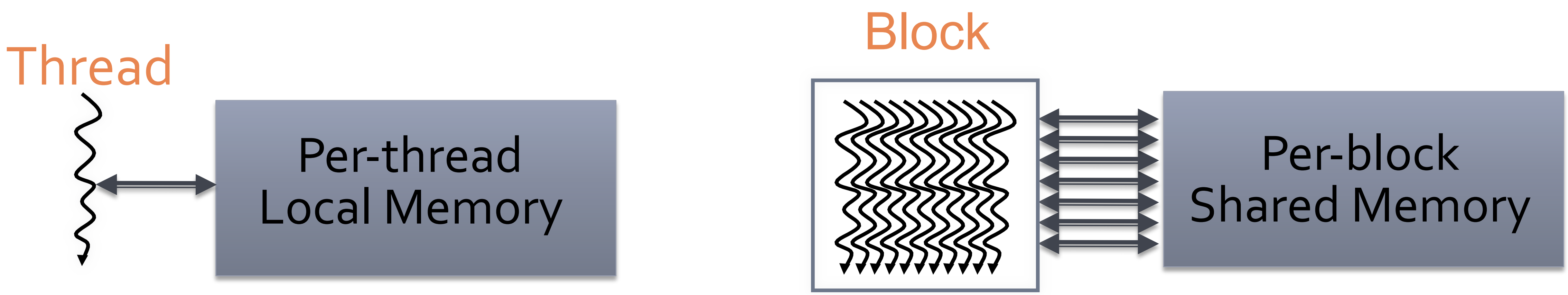


- CUDA allows one binary to target all these chips
- Thread blocks bring scalability!

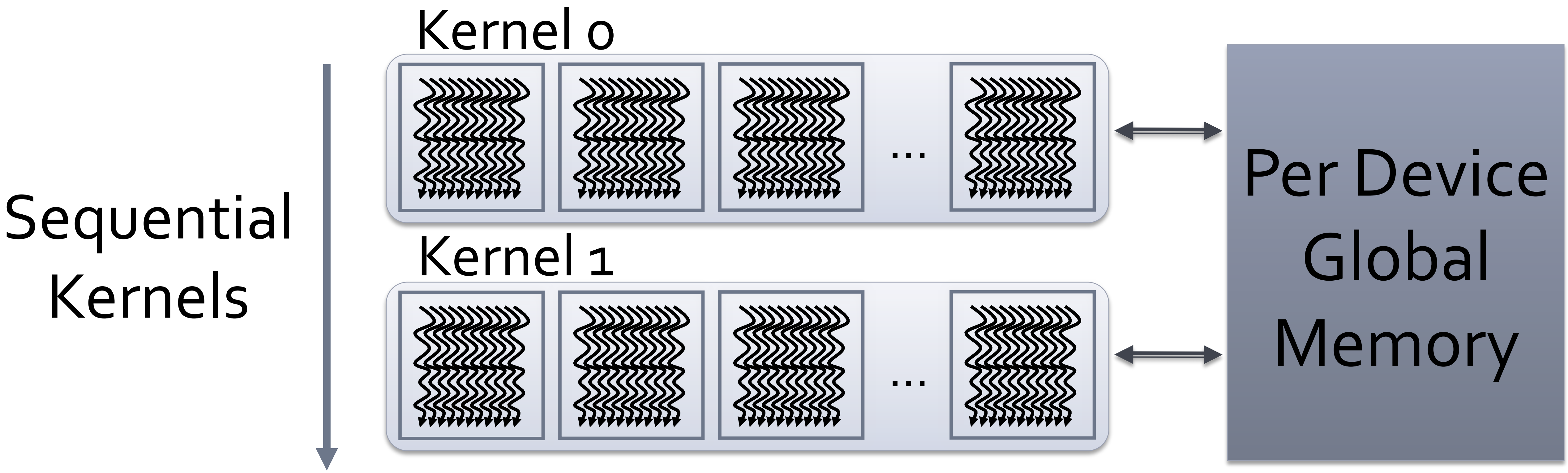
Hello World: Vector Addition

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

int main() {
   //Run N/256 blocks of 256 threads each
   vecAdd<<<N/256, 256>>>(d_a, d_b, d_c);
}
```

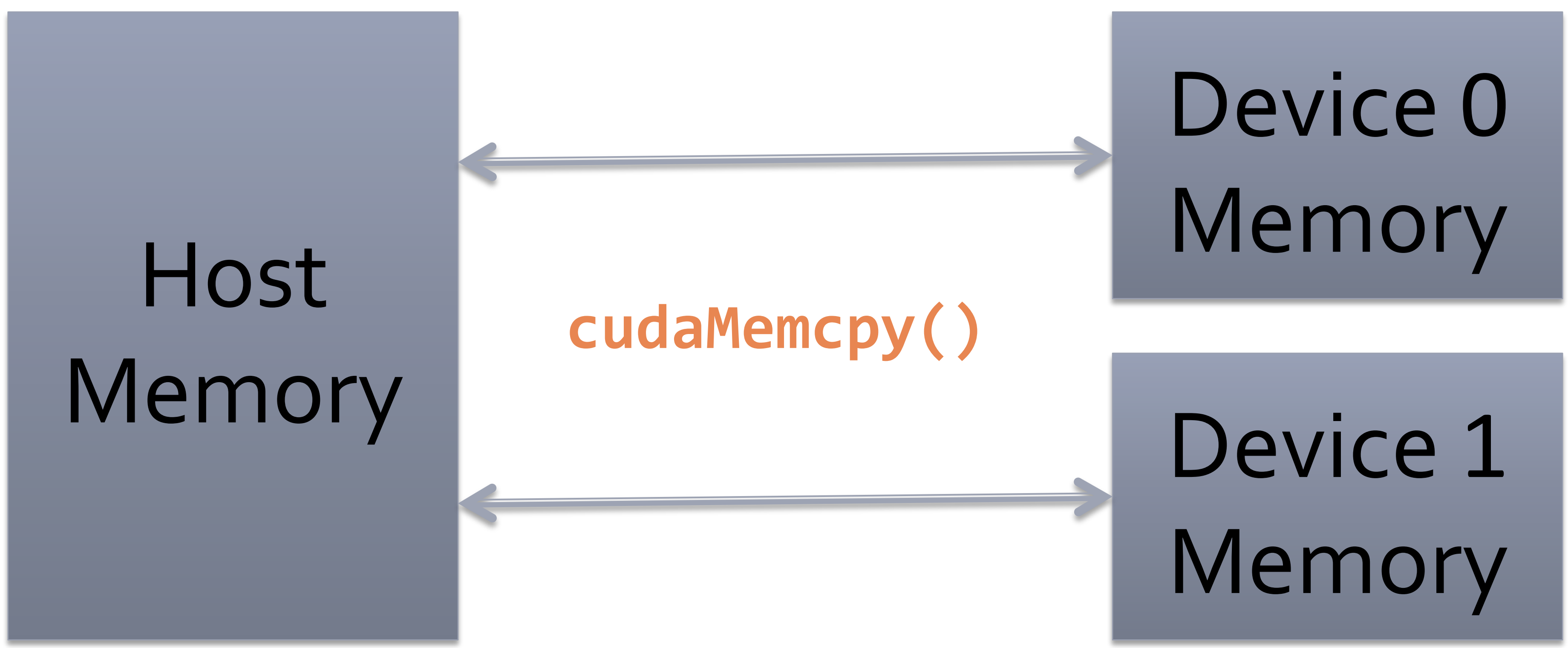
Memory model



Memory model



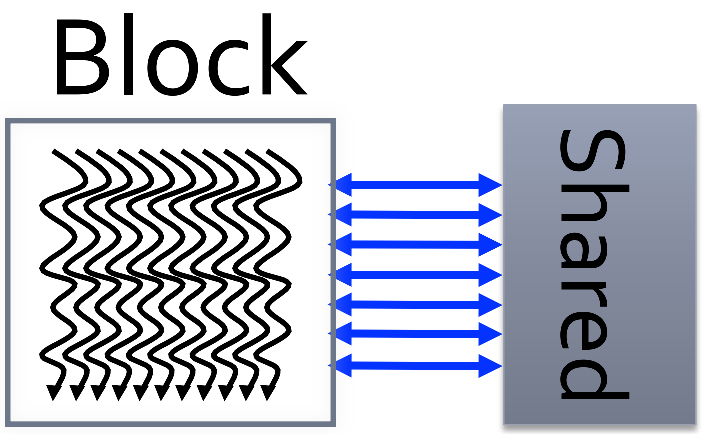
Memory model



Hello World: Managing Data

```
int main() {
   int N = 256 * 1024;
   float* h_a = malloc(sizeof(float) * N);
   //Similarly for h_b, h_c. Initialize h_a, h_b

   float *d_a, *d_b, *d_c;
   cudaMalloc(&d_a, sizeof(float) * N);
   //Similarly for d_b, d_c

   cudaMemcpy(d_a, h_a, sizeof(float) * N, cudaMemcpyHostToDevice);
   //Similarly for d_b

   //Run N/256 blocks of 256 threads each
   vecAdd<<<N/256, 256>>>(d_a, d_b, d_c);

   cudaMemcpy(h_c, d_c, sizeof(float) * N, cudaMemcpyDeviceToHost);
}
```

Using per-block shared memory



- Variables shared across block

```
__shared__ int *begin, *end;
```

- Scratchpad memory

```
__shared__ int scratch[BLOCKSIZE];
scratch[threadIdx.x] = begin[threadIdx.x];
// … compute on scratch values …
begin[threadIdx.x] = scratch[threadIdx.x];
```

- Communicating values between threads

```
scratch[threadIdx.x] = begin[threadIdx.x];
__syncthreads();
int left = scratch[threadIdx.x - 1];
```

- Per-block shared memory is faster than L1 cache, slower than register file
- It is relatively small: register file is 2-4x larger

CUDA: Minimal extensions to C/C++

- Declaration specifiers to indicate where things live

```
__global__ void KernelFunc(...);   // kernel callable from host
__device__ void DeviceFunc(...);   // function callable on device
__device__ int  GlobalVar;         // variable in device memory
__shared__ int  SharedVar;         // in per-block shared memory
```

- Extend function invocation syntax for parallel kernel launch

```
KernelFunc<<<500, 128>>>(...);     // 500 blocks, 128 threads each
```

- Special variables for thread identification in kernels

```
dim3 threadIdx;  dim3 blockIdx;  dim3 blockDim;
```

- Intrinsics that expose specific operations in kernel code

```
__syncthreads();                   // barrier synchronization
```

CUDA: Features available on GPU

- Double and single precision (IEEE compliant)

- Standard mathematical functions
  - `sinf`, `powf`, `atanf`, `ceil`, `min`, `sqrtf`, etc.

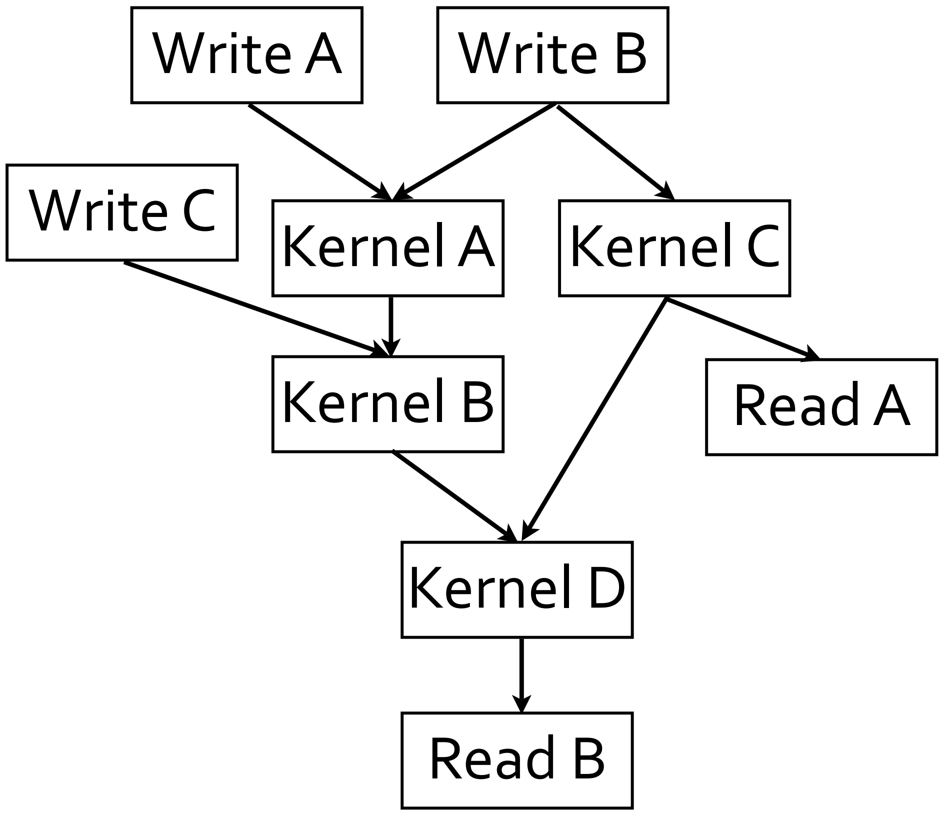
- Atomic memory operations
  - `atomicAdd`, `atomicMin`, `atomicAnd`, `atomicCAS`, etc.
- These work on both global and shared memory

CUDA: Runtime support

- Explicit memory allocation returns pointers to GPU memory
  - `cudaMalloc()`, `cudaFree()`
- Explicit memory copy for host ↔ device, device ↔ device
  - `cudaMemcpy()`, `cudaMemcpy2D()`, ...
- Texture management
  - `cudaBindTexture()`, `cudaBindTextureToArray()`, ...
- OpenGL & DirectX interoperability
  - `cudaGLMapBufferObject()`, `cudaD3D9MapVertexBuffer()`, ...

OpenCL

- OpenCL is supported by AMD {CPUs, GPUs} and Nvidia
  - Intel, Imagination Technologies (purveyor of GPUs for iPhone/OMAP/etc.) are also on board
- OpenCL's data parallel execution model mirrors CUDA, but with different terminology
- OpenCL has rich task parallelism model
- Runtime walks a dataflow DAG of kernels/memory transfers



CUDA and OpenCL correspondence

CUDA		OpenCL
Thread	⟷	Work-item
Thread-block	⟷	Work-group
Global memory	⟷	Global memory
Constant memory	⟷	Constant memory
Shared memory	⟷	Local memory
Local memory	⟷	Private memory
__global__ function	⟷	__kernel function
__device__ function	⟷	no qualification needed
__constant__ variable	⟷	__constant variable
__device__ variable	⟷	__global variable
__shared__ variable	⟷	__local variable

OpenCL and SIMD

- SIMD issues are handled separately by each runtime
- AMD GPU
  - Vectorize over 64-way SIMD, but not over 4/5-way VLIW
    - Use float4 vectors in your code
- AMD CPU
  - No vectorization
    - Use float4 vectors in your code (float8 when AVX appears?)
- Nvidia GPU
  - Full vectorization, like CUDA
    - Prefers scalar code per work-item

Imperatives for Efficient CUDA Code

- Expose abundant fine-grained parallelism
  - need 1000's of threads for full utilization
- Maximize on-chip work
  - on-chip memory orders of magnitude faster
- Minimize execution divergence
  - SIMT execution of threads in 32-thread warps
- Minimize memory divergence
  - warp loads and consumes complete 128-byte cache line

Mapping CUDA to Nvidia GPUs

- CUDA is designed to be functionally forgiving
  - First priority: make things work. Second: get performance.
- However, to get good performance, one must understand how CUDA is mapped to Nvidia GPUs
- Threads: each thread is a SIMD vector lane
- Warps: A SIMD instruction acts on a “warp”
  - Warp width is 32 elements: ***LOGICAL*** SIMD width
- Thread blocks: Each thread block is scheduled onto an SM
  - Peak efficiency requires multiple thread blocks per SM

Mapping CUDA to a GPU, *continued*

- The GPU is very deeply pipelined to maximize throughput
- This means that performance depends on the number of thread blocks which can be allocated on a processor
- Therefore, resource usage costs performance:
  - More registers => Fewer thread blocks
  - More shared memory usage => Fewer thread blocks
- It is often worth trying to reduce register count in order to get more thread blocks to fit on the chip
  - For Fermi, target 20 registers or less per thread for full occupancy

Occupancy (Constants for Fermi)

- The Runtime tries to fit as many thread blocks simultaneously as possible on to an SM
  - The number of simultaneous thread blocks (B) is ≤ 8
- The number of warps per thread block (T) ≤ 32
- B * T ≤ 48 (Each SM has scheduler space for 48 warps)
- The number of threads per warp (V) is 32
- B * T * V * Registers per thread ≤ 32768
- B * Shared memory (bytes) per block ≤ 49152/16384
  - Depending on Shared memory/L1 cache configuration

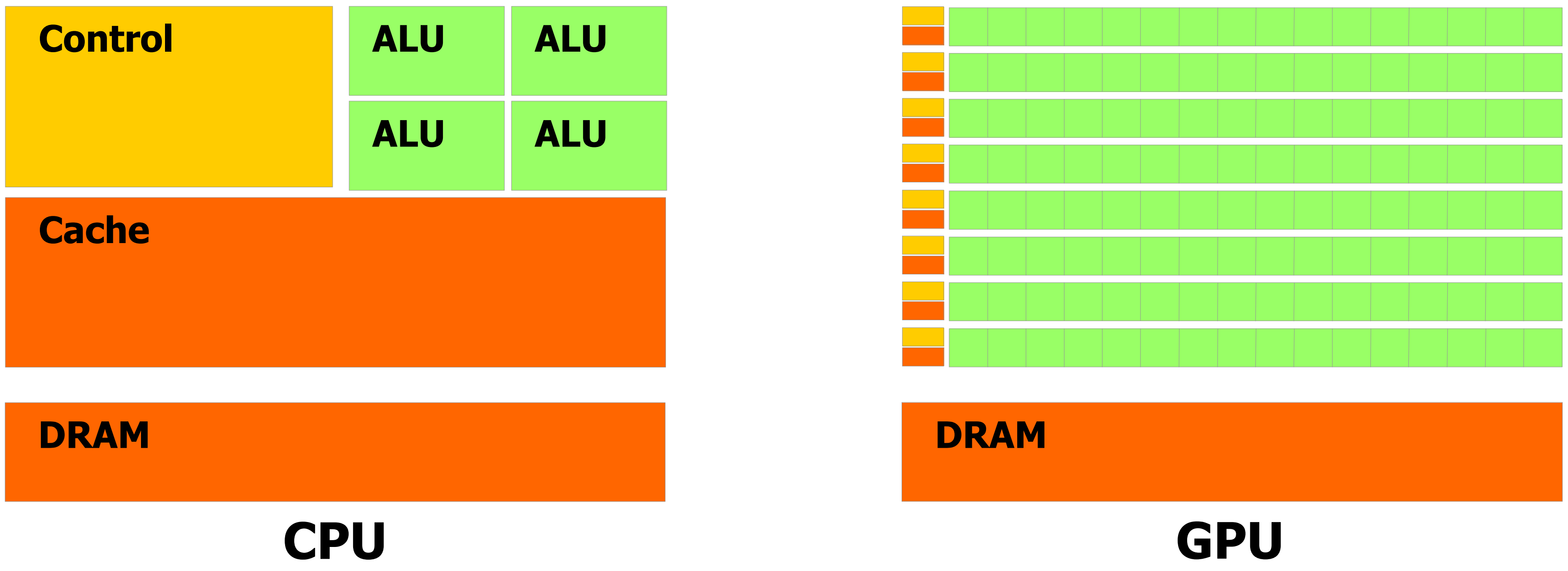
- Occupancy is reported as B * T / 48

SIMD & Control Flow

- Nvidia GPU hardware handles control flow divergence and reconvergence
  - Write scalar SIMD code, the hardware schedules the SIMD execution
  - One caveat: __syncthreads() can't appear in a divergent path
    - This will cause programs to hang
  - Good performing code will try to keep the execution convergent within a warp
    - Warp divergence only costs because of a finite instruction cache

Memory, Memory, Memory

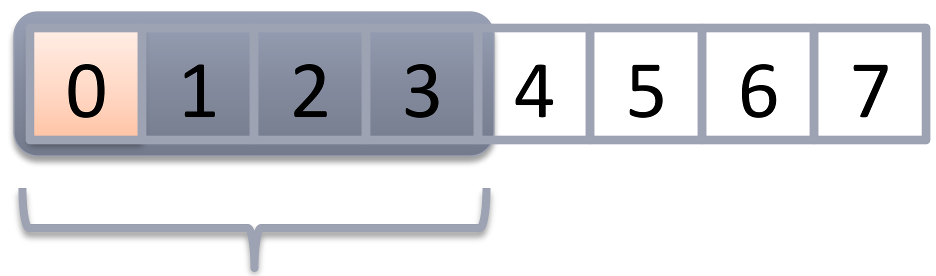
- A many core processor ≡ A device for turning a compute bound problem into a memory bound problem



- Lots of processors, only one socket
- Memory concerns dominate performance tuning

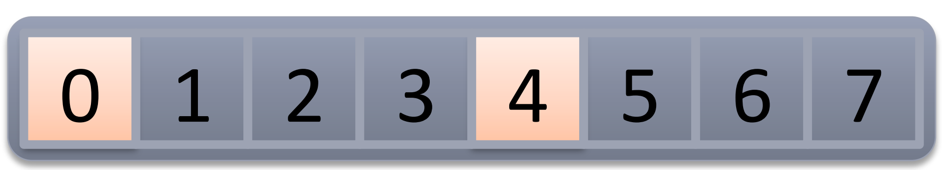
Memory is SIMD too

- Virtually all processors have SIMD memory subsystems



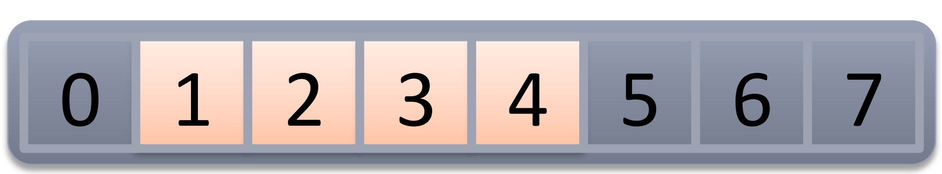
cache line width

- This has two effects:
  - Sparse access wastes bandwidth



2 words used, 8 words loaded:
¼ effective bandwidth

  - Unaligned access wastes bandwidth

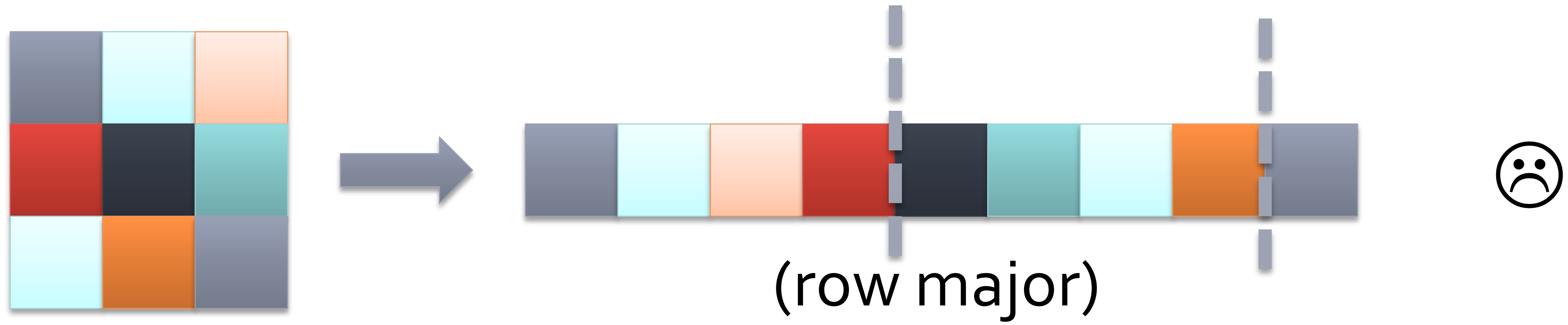


4 words used, 8 words loaded:
½ effective bandwidth

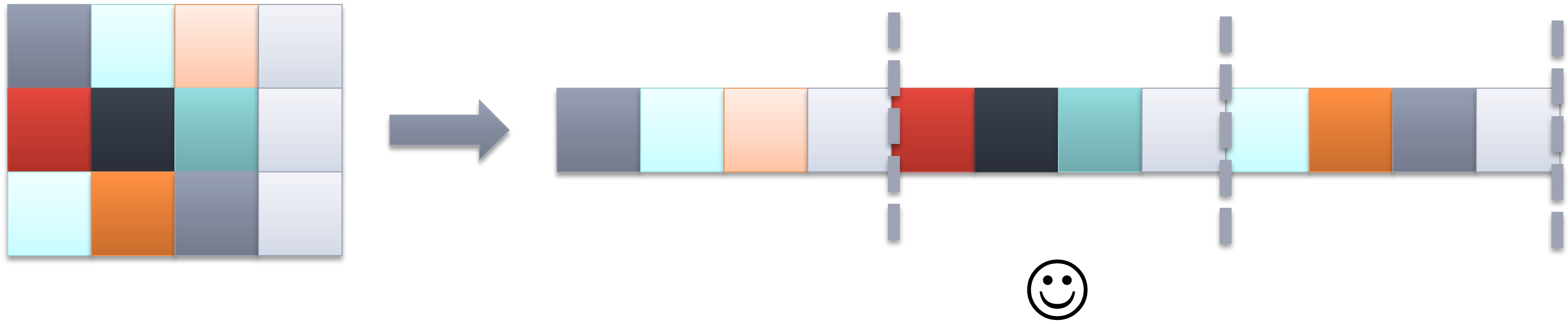
Coalescing

- GPUs and CPUs both perform memory transactions at a larger granularity than the program requests ("cache line")
- GPUs have a "coalescer", which examines memory requests dynamically and coalesces them
- To use bandwidth effectively, when threads load, they should:
  - Present a set of unit strided loads (dense accesses)
  - Keep sets of loads aligned to vector boundaries

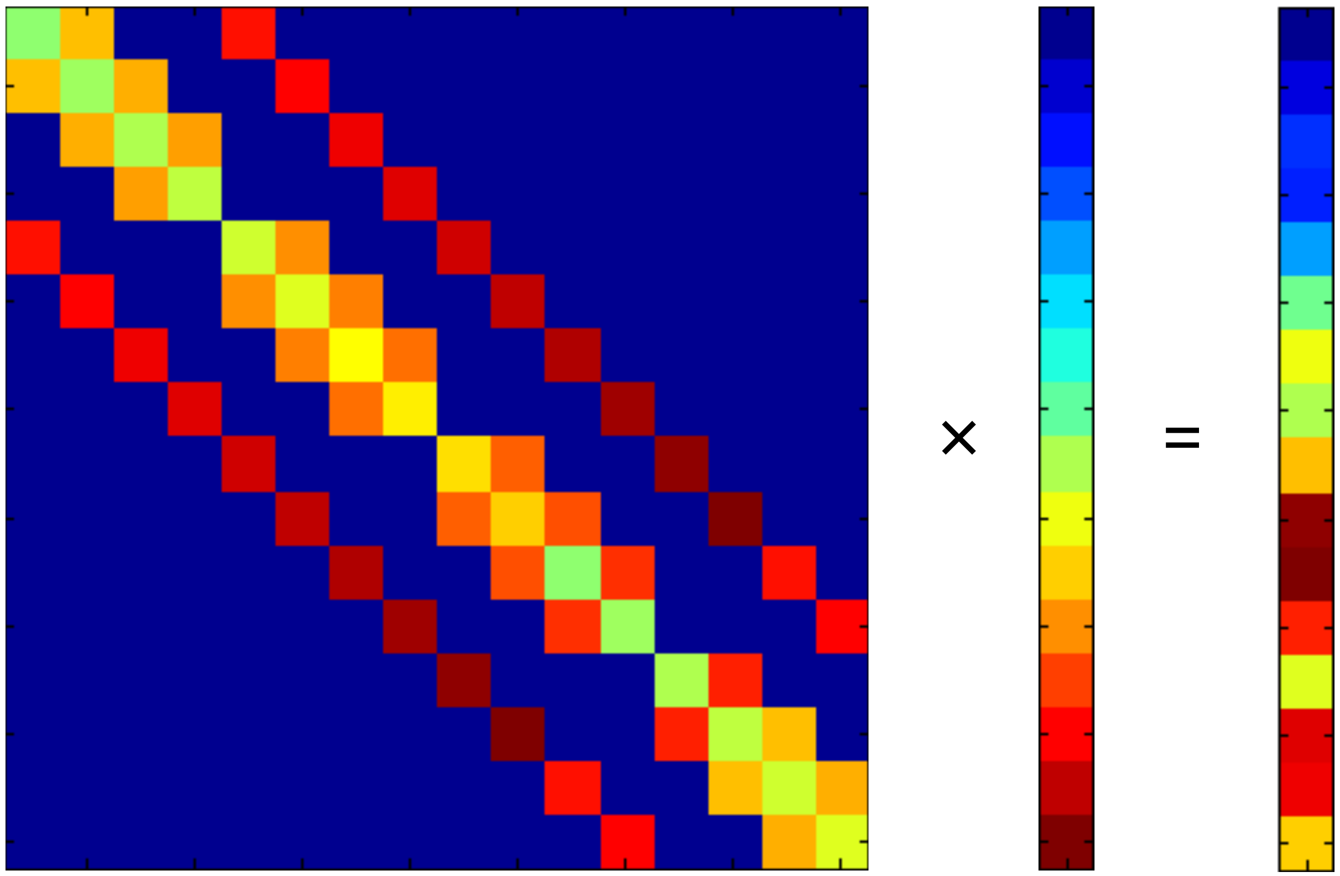
Data Structure Padding



- Multidimensional arrays are usually stored as monolithic vectors in memory
- Care should be taken to assure aligned memory accesses for the necessary access pattern

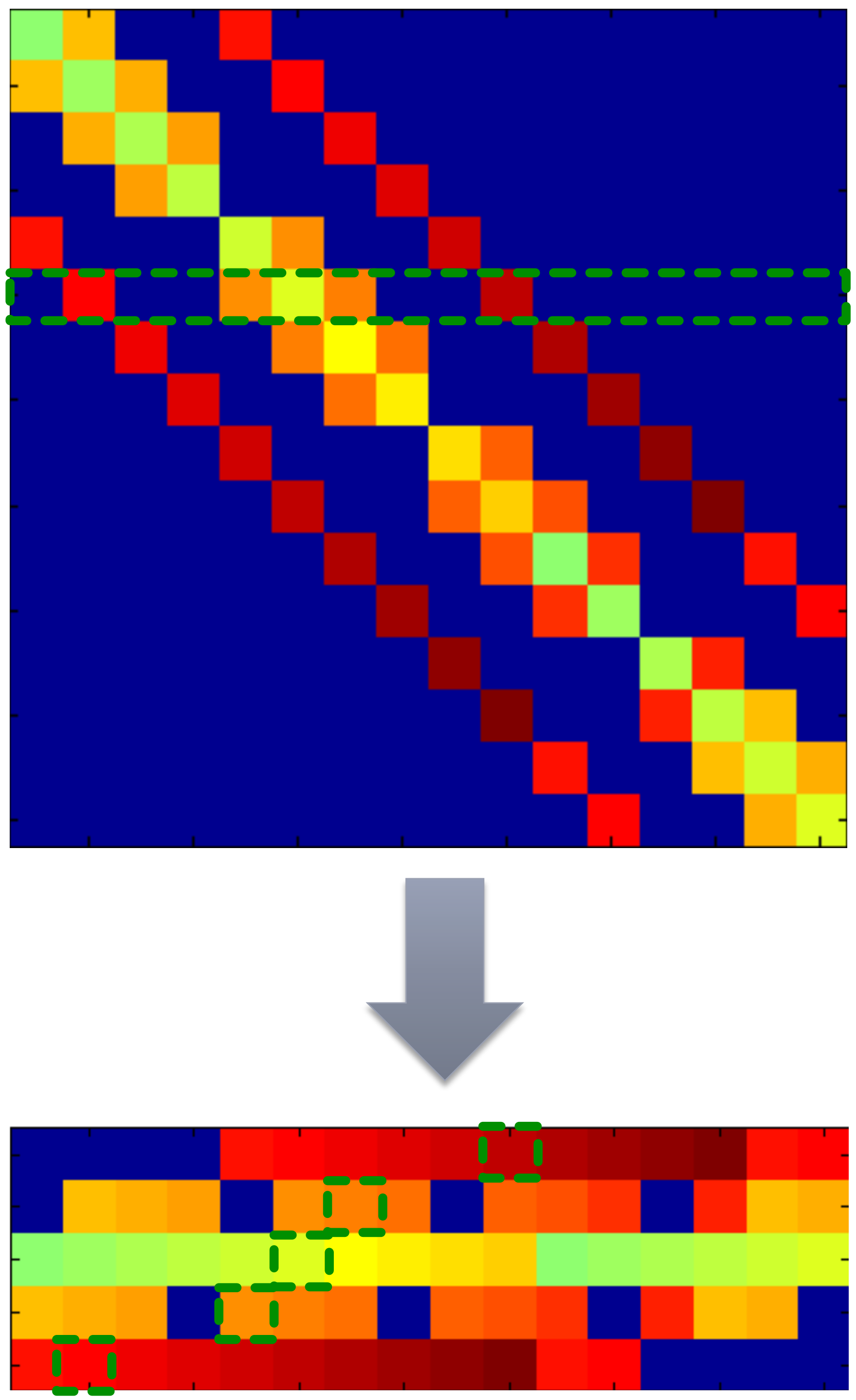


Sparse Matrix Vector Multiply



- Problem: Sparse Matrix Vector Multiplication
- How should we represent the matrix?
  - Can we take advantage of any structure in this matrix?

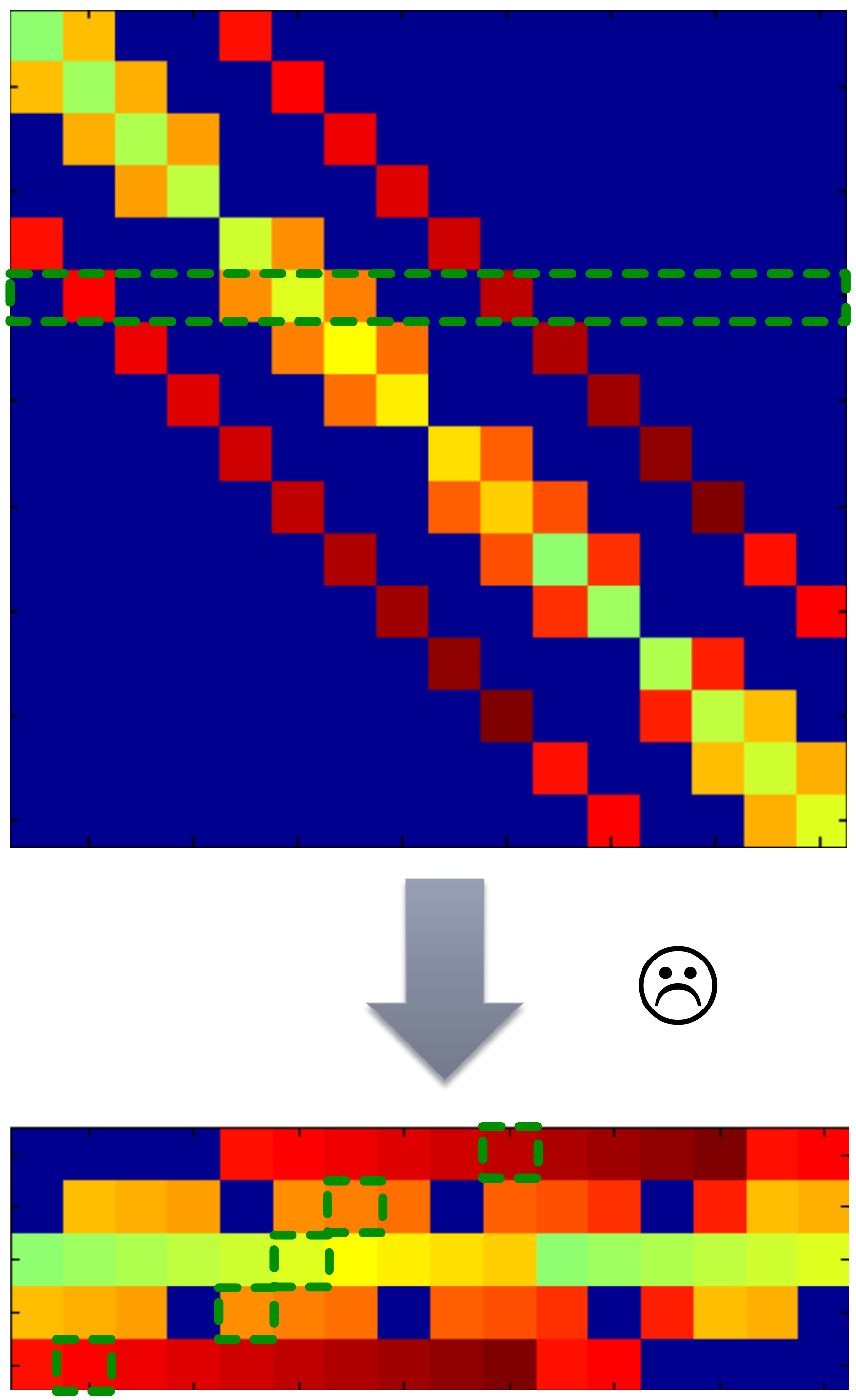
Diagonal representation

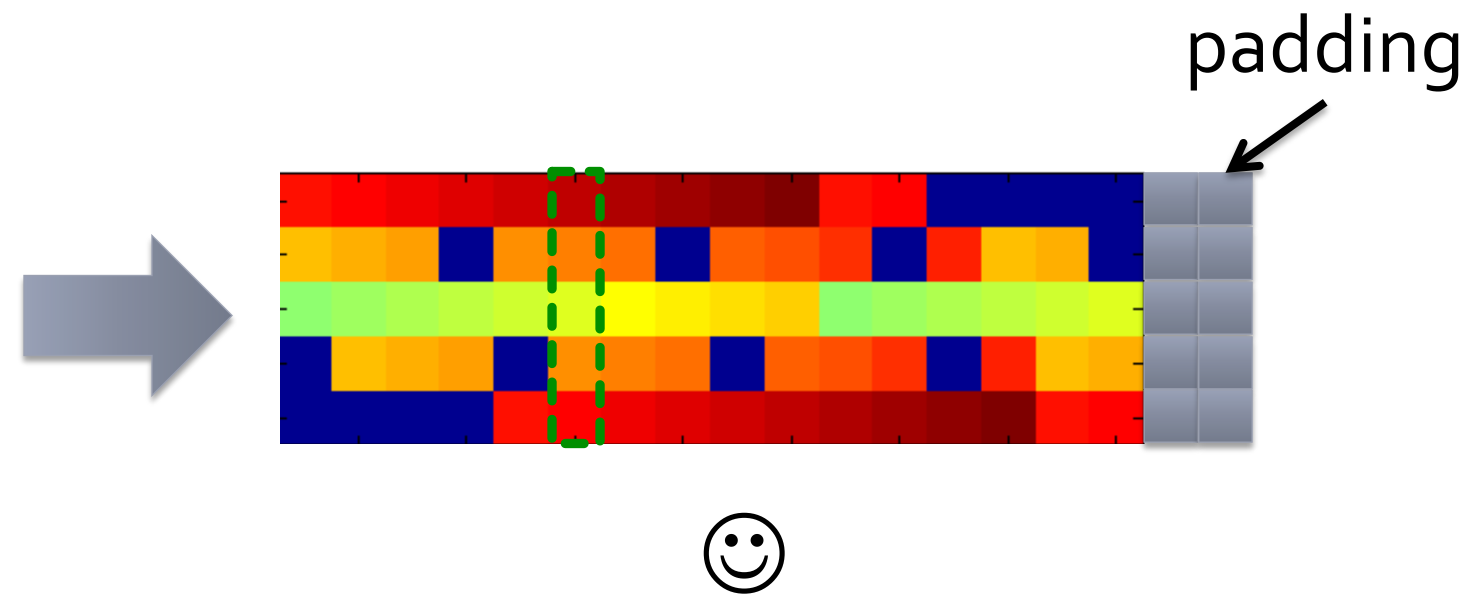


- Since this matrix has nonzeros only on diagonals, let's project the diagonals into vectors
- Sparse representation becomes dense
- Launch a thread per row
- Are we done?

- The straightforward diagonal projection is not aligned

Optimized Diagonal Representation

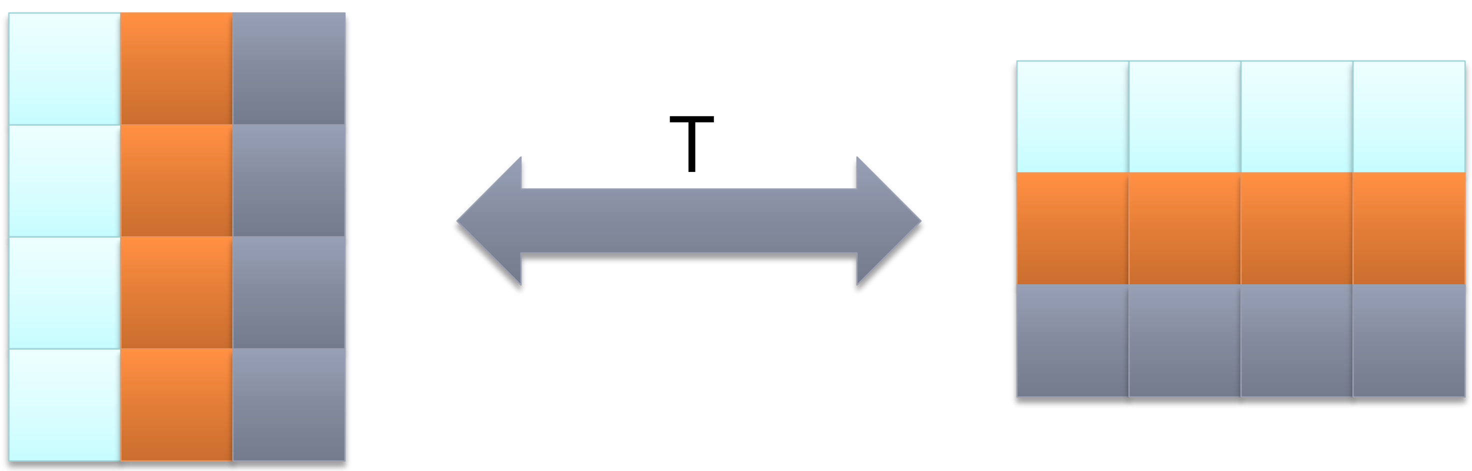




- Skew the diagonals again
- This ensures that all memory loads from matrix are coalesced
- Don't forget padding!

SoA, AoS

- Different data access patterns may also require transposing data structures



Array of Structs　　　　Structure of Arrays

- The cost of a transpose on the data structure is often much less than the cost of uncoalesced memory accesses
- Use shared memory to handle block transposes

- There exist many tools and libraries for GPU programming
- Thrust is now part of the CUDA SDK
- C++ libraries for CUDA programming, inspired by STL
- Many important algorithms:
  - reduce, sort, reduce_by_key, scan, ...
- Dramatically reduces overhead of managing heterogeneous memory spaces
- Includes OpenMP backend for multicore programming

Hello World of Thrust

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <cstdlib>

int main(void)
{
    // generate 32M random numbers on the host
    thrust::host_vector<int> h_vec(32 << 20);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device (846M keys per sec on GeForce GTX 480)
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
```

saxpy in Thrust

```
// C++ functor replaces __global__ function
struct saxpy
{
    float a;

    saxpy(float _a) : a(_a) {}

    __host__ __device__
    float operator()(float x, float y)
    {
        return a * x + y;
    }
};

transform(x.begin(), x.end(), y.begin(), y.begin(), saxpy(a));
```

Summary

- Manycore processors provide useful parallelism
- Programming models like CUDA and OpenCL enable productive parallel programming
- They abstract SIMD, making it easy to use wide SIMD vectors
- CUDA and OpenCL encourages SIMD friendly, highly scalable algorithm design and implementation
- Thrust is a productive C++ library for CUDA development

Questions?

Bryan Catanzaro

bcatanzaro@nvidia.com

http://research.nvidia.com