Outline

- GPU history
- A short introduction to CUDA

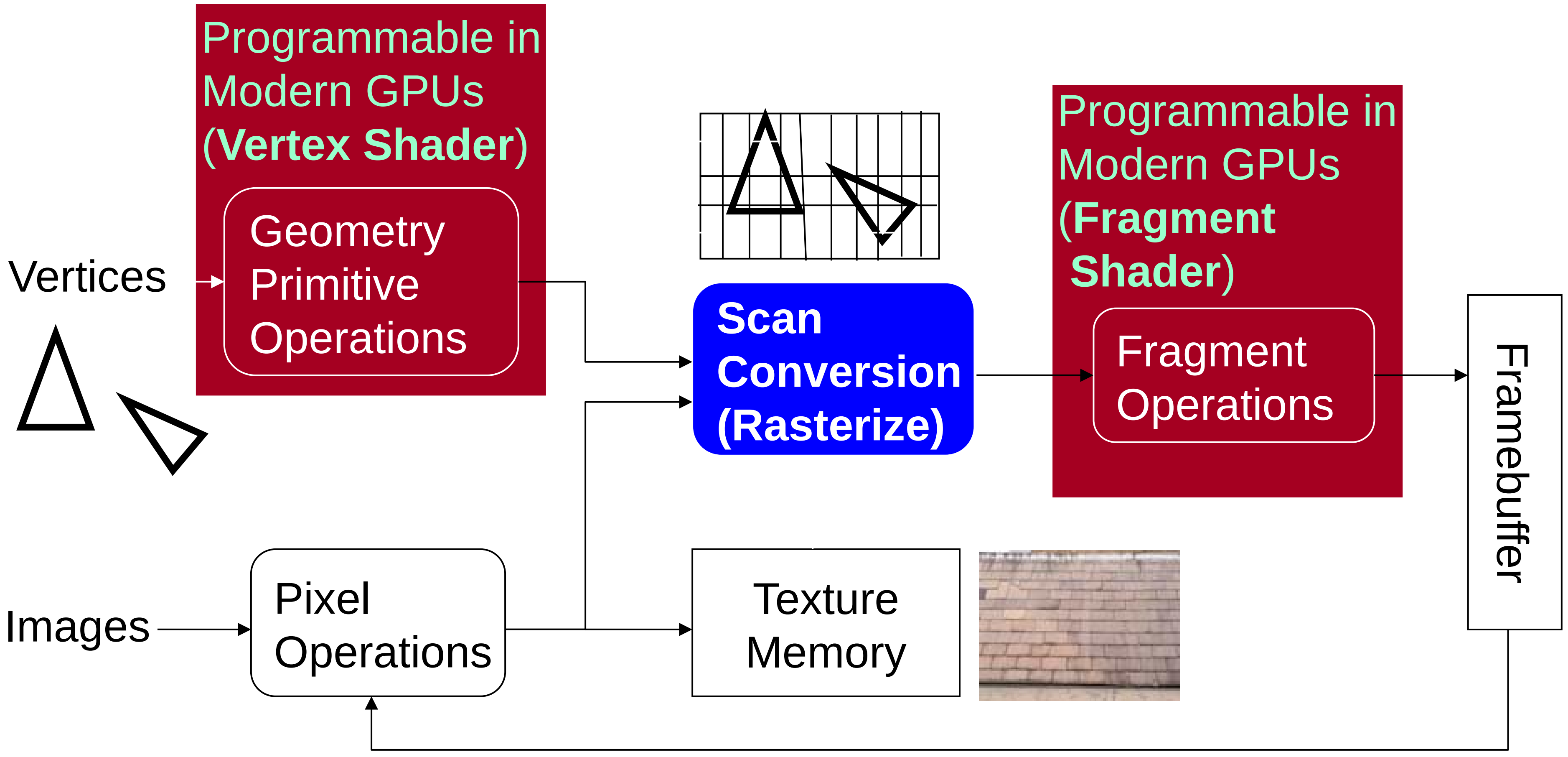
HPC Energy consumption

- At ~$1M per MW, energy costs are substantial
  - 1 petaflop in 2008 used 3 MW
  - 1 exaflop was projected for 2018 at 200 MW “usual chip scaling” so the goal was 20 MW
  - Reality will probably be close to 50 MW in 2022
- Example machines:
  - Tihanhe-2 at 18MW (2013)
  - TaihuLight at 15 MW (2016)
  - Fugaku at 28 MW (2020)

Brief history

- Late 80s-early 90s: “golden age” for supercomputing
  - Companies: Thinking Machines, MasPar, Cray
  - Relatively fast processors (vs memory)
  - Lots of academic interest and development
  - But got hard to compete with commodity hardware
    - Scientific computing is not a market driver!
- 90s-early 2000s: age of the cluster
  - Beowulf, grid computing, etc.
  - “Big iron” also uses commodity chips (better interconnect)
- Past few years
  - CPU producers move to multicore
  - High-end graphics becomes commodity HW
    - Gaming is a market driver!
  - GPU producers realize their many-core designs can apply to general purpose computing

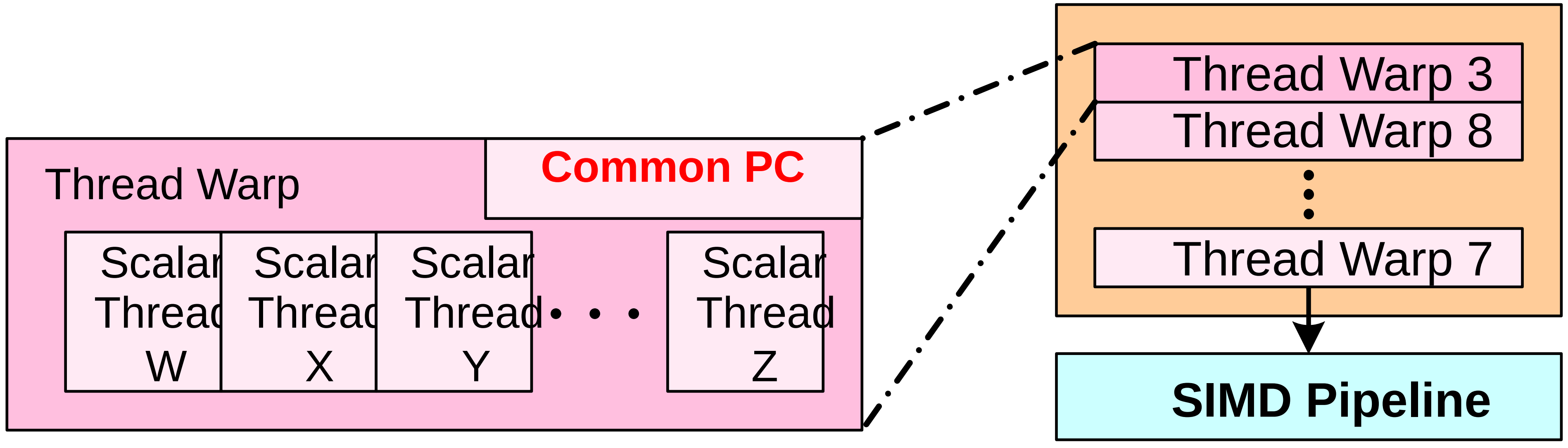
GPU Programmable Shaders


Programmable in Modern GPUs (**Vertex Shader**)
Vertices
Geometry Primitive Operations
Scan Conversion (Rasterize)
Programmable in Modern GPUs (**Fragment Shader**)
Fragment Operations
Framebuffer
Images
Pixel Operations
Texture Memory

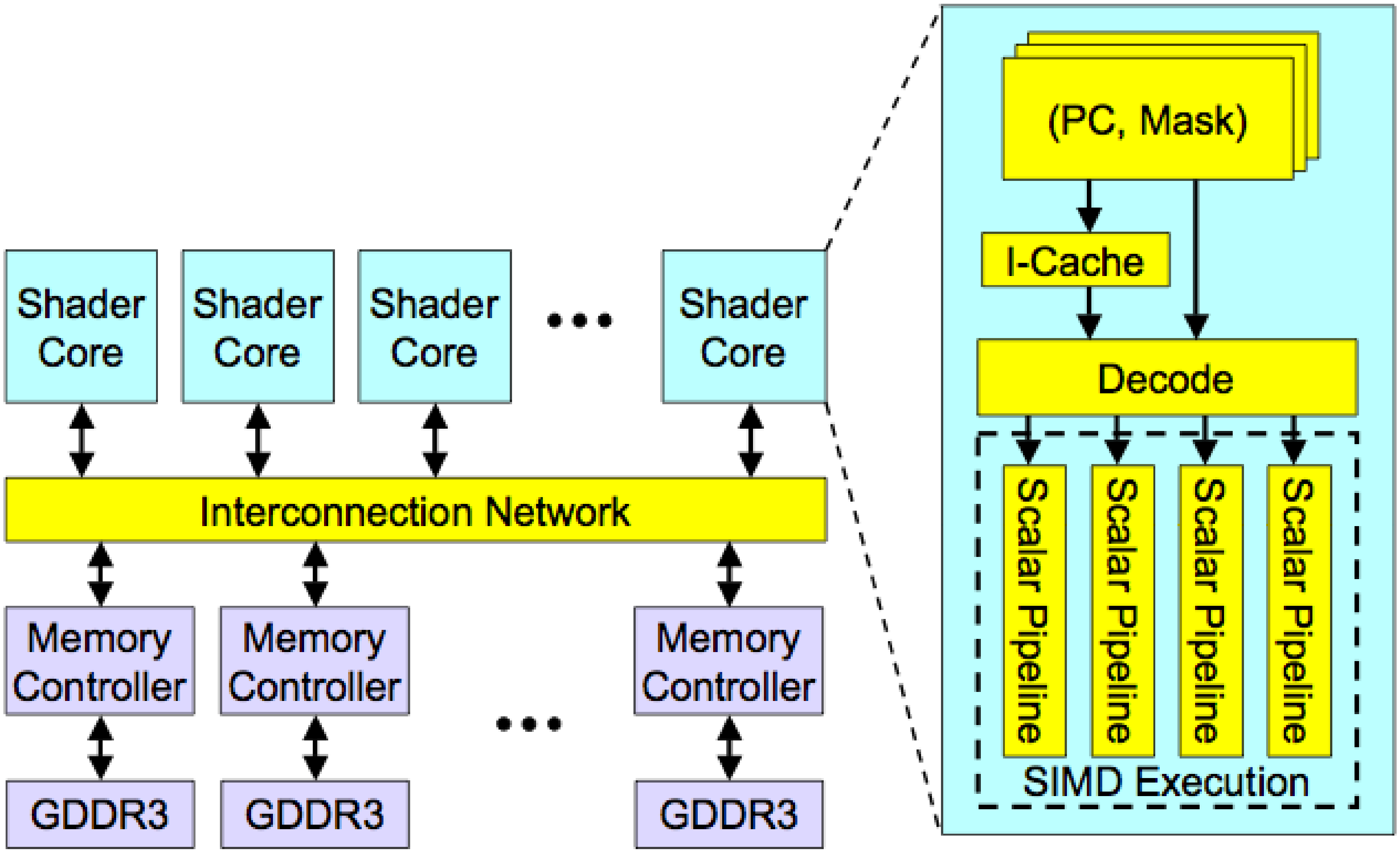
Traditional Approach: Fixed function pipeline (state machine)
New Development (2003-): Programmable pipeline

Warps and Warp-Level Fine-Grain Multithreaded Execution

- Warp: A set of threads that execute the same instruction (on different data elements)
- All threads run the same code
  - Warp: The threads that run lengthwise in a woven fabric …

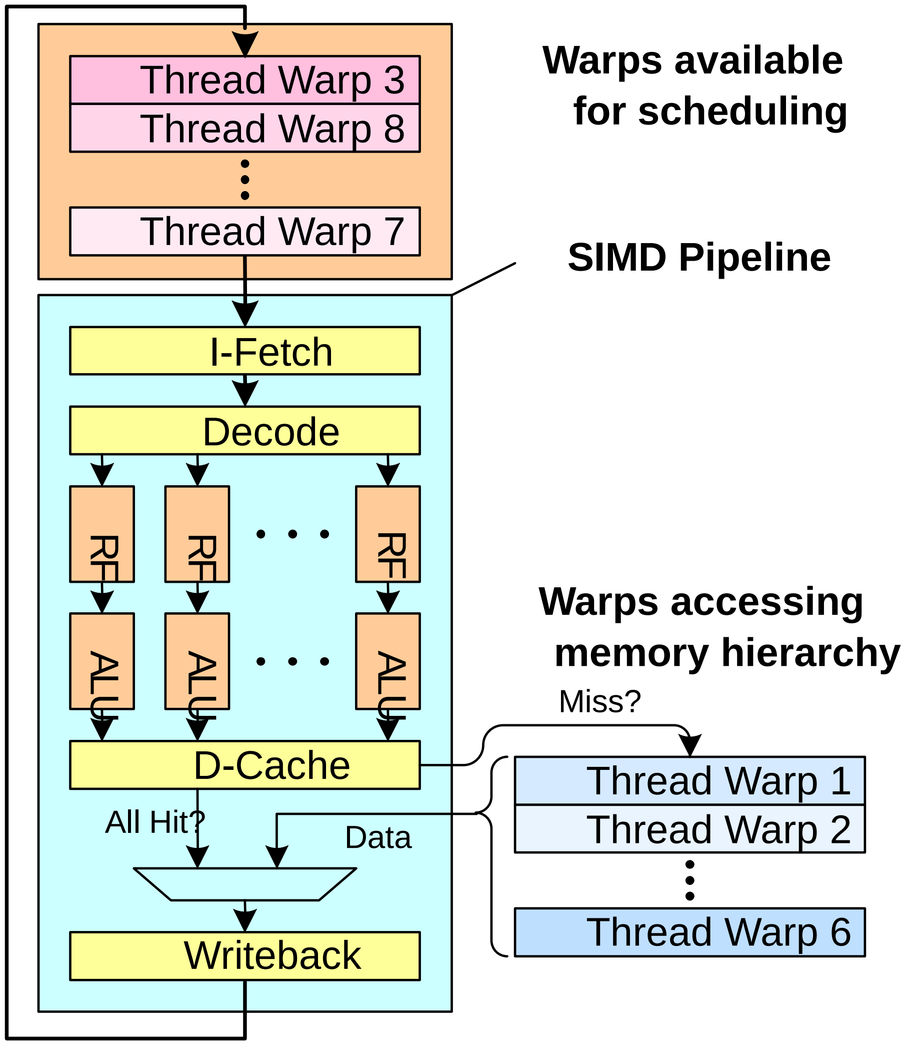


High-Level View of a GPU



Latency Hiding via Warp-Level Fine Grain Multithreading

- Warp: A set of threads that execute the same instruction (on different data elements)
- Fine-grained multithreading
  - One instruction per thread in pipeline at a time (No interlocking)
  - Interleave warp execution to hide latencies
- Register values of all threads stay in register file
- Fine-Grain multithreading enables long latency tolerance
  - Millions of pixels



Threads

- Threads on desktop CPUs
  - Implemented via lightweight processes (for example)
  - General system scheduler
  - Thrashing when more active threads than processors
- An alternative approach
  - Hardware support for many threads / CPU
    - Modest example: hyperthreading
    - More extreme: Cray MTA-2 and XMT
  - Hide memory latency by thread switching
  - Want many more independent threads than cores
- GPU programming
  - Thread creation / context switching are basically free
  - Want lots of threads (thousands for efficiency?!)

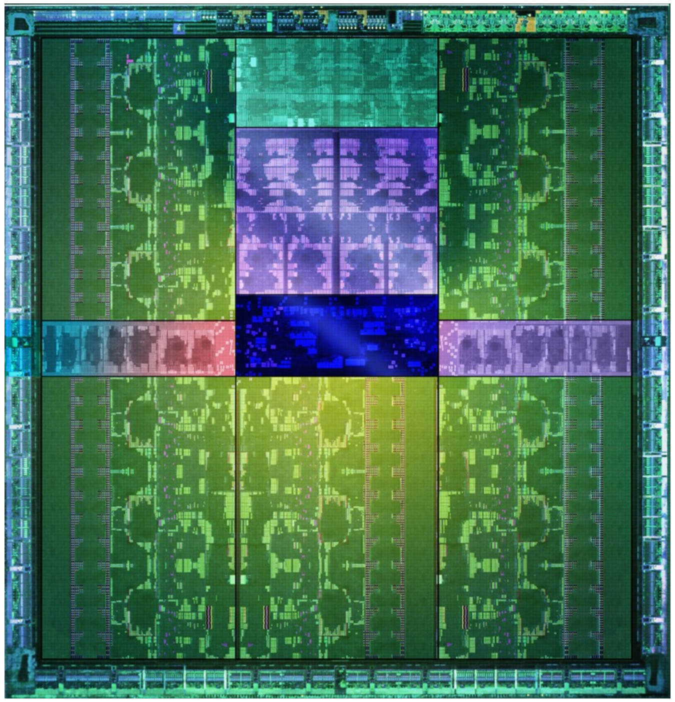
Throughput vs. Latency

- GPU goal: maximum throughput
  - massively-multithreaded architecture, use very large register file
- CPU goal: minimum latency
  - use tiny register file and much larger caches to optimize latency

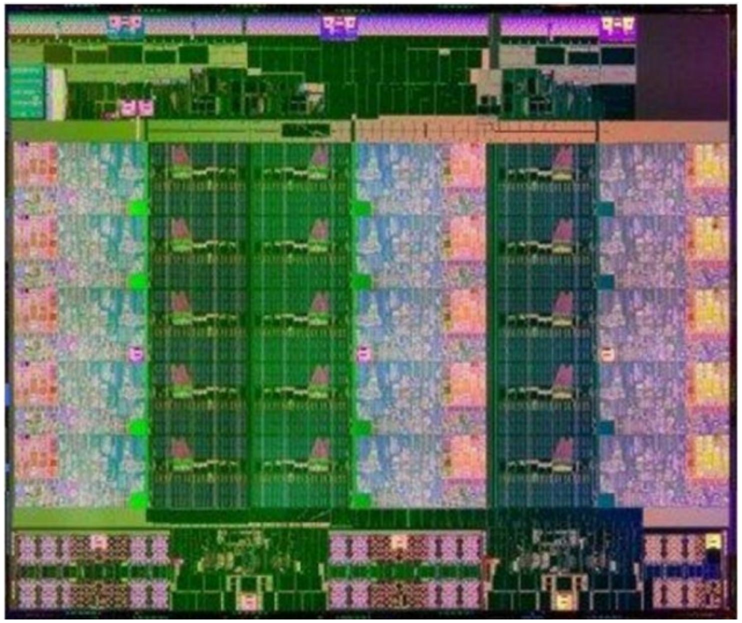
Specifications	Ivy Bridge EX (Xeon E7-8890v2)	Kepler (Tesla K40)
Processing Elements	15 cores, 2 issue, 8 way SIMD @**2.8** GHz	15 SMs, 6 issue, 32 way SIMD @**745** MHz
Resident Strands/Threads (max)	15 cores, 2 threads, 8 way SIMD: **240** strands	15 SMs, 64 SIMD vectors, 32 way SIMD: **30720** threads
SP GFLOP/s	672	4291
Memory Bandwidth	85 GB/s	288 GB/s
Register File	xx kB (?)	3.75 MB
Local Store/L1 Cache	960 kB	960 kB
L2 Cache	3.75 MB	1.5 MB

GPU vs. CPU masks

- Kepler



- Ivy Bridge



Throughput vs. Latency

- Different goals produce different designs
  - Throughput cores: assume work load is highly parallel
  - Latency cores: assume workload is mostly sequential
- Latency goal: minimize latency experienced by 1 thread
  - lots of big on-chip caches
  - extremely sophisticated control
- Throughput goal: maximize throughput of all threads

SIMD: Parallel data

- OpenMP / Pthreads / MPI all neglect SIMD parallelism
- Because 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

- Most common solution:
  - Either forget about SIMD (And maybe the autovectorizer likes you)
  - Or instantiate intrinsics (assembly language)
  - Requires a new code version for every SIMD extension

General-purpose GPU programming

- Old GPGPU model: use texture mapping interfaces
  - People got good performance!
  - But too clever by half
- CUDA (Compute Unified Device Architecture)
  - More natural general-purpose programming model
  - Initial release in 2007; now in version 11.7
- OpenCL
  - Relatively new (late 2009); in Apple’s Snow Leopard
  - Open standard

CUDA

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

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

CUDA Threads

- Independent thread of execution
  - has its own program counter, variables (registers),
    processor state, etc.
  - no implication about how threads are scheduled
- CUDA threads might be physical threads
  - as mapped onto GPUs
- CUDA threads might be virtual threads
  - might pick 1 block = 1 physical thread on multicore CPU

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 Thread blocks

- A 1D Grid of 1D Blocks:
  - int threadId = blockIdx.x *blockDim.x + threadIdx.x;
- A 2D Grid of 1D Blocks:
  - int blockId = blockIdx.y * gridDim.x + blockIdx.x;
  - int threadId = blockId * blockDim.x + threadIdx.x;
- A 2D Grid of 2D Blocks:
  - int blockId = blockIdx.x + blockIdx.y * gridDim.x;
  - int threadId = blockId * (blockDim.x * blockDim.y) + (threadIdx.y * blockDim.x) + threadIdx.x;

CUDA parallelism

- 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);
```

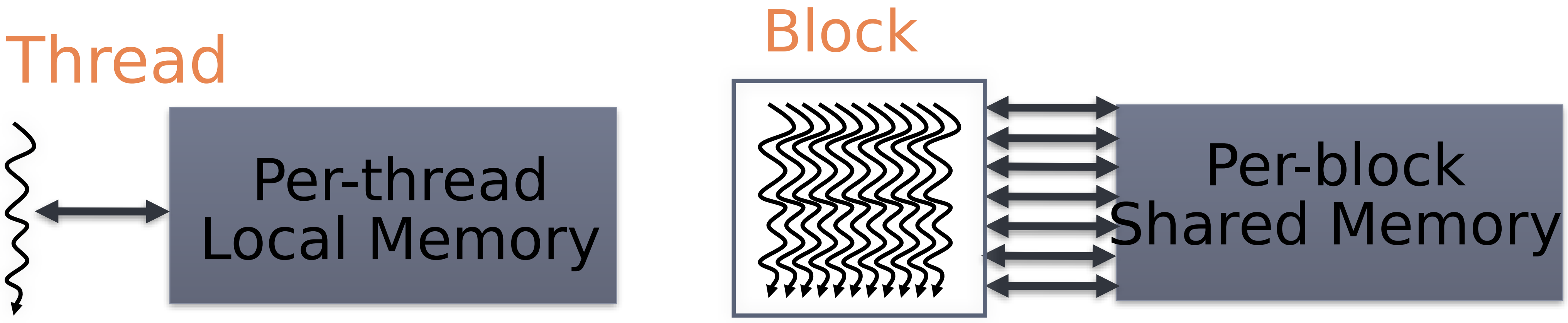
Independence

- 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

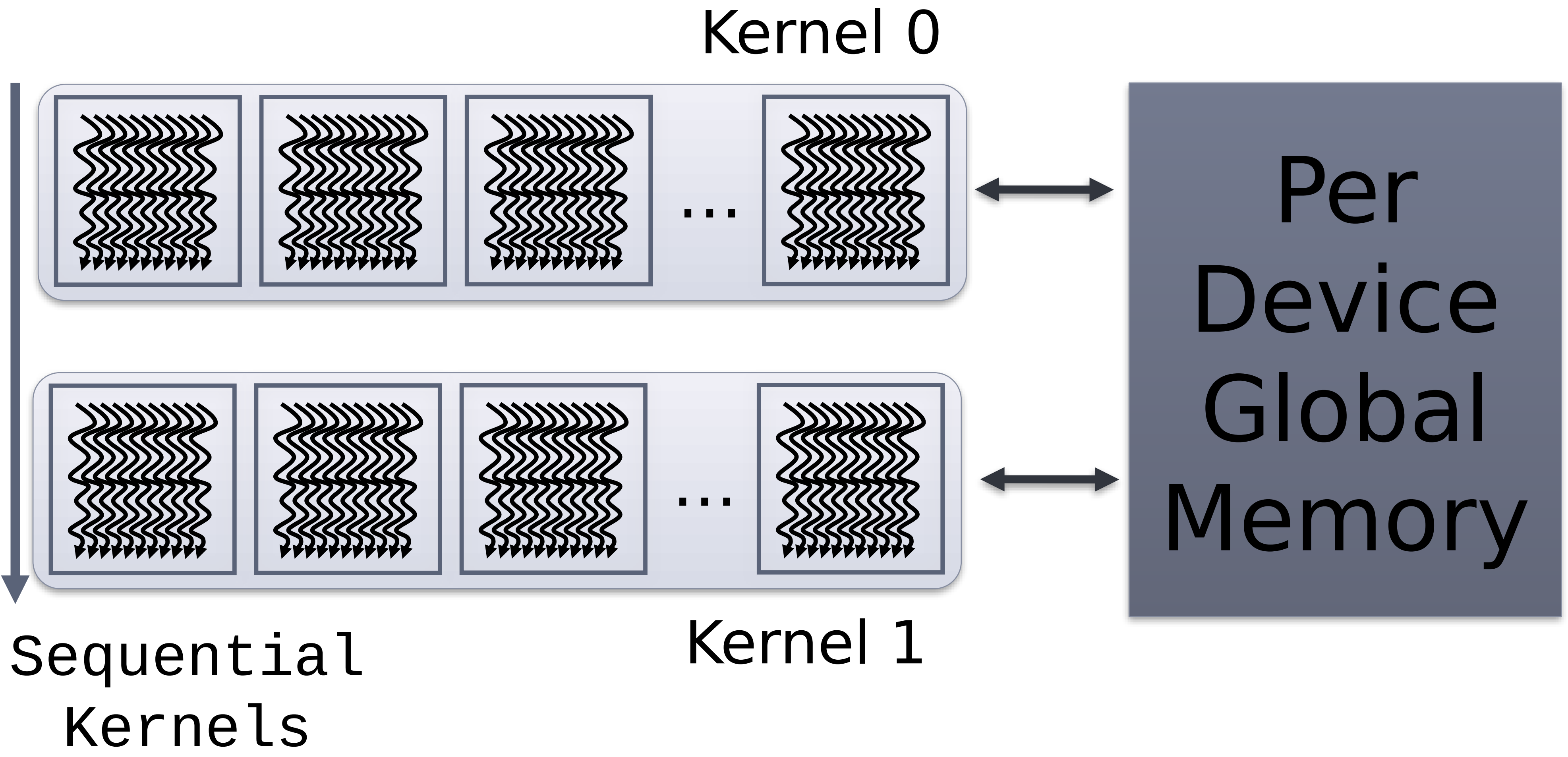
- Blocks may coordinate but not synchronize
  - shared queue pointer: OK
  - shared lock: BAD … can easily deadlock

- Independence requirement gives *scalability*

Memory Model



Memory Model



Vector Addition (CPU)

```
#include <iostream>
int main(void) {
  int N = 1<<20; // 1M elements
  float *x = new float[N]; // Allocate memory
  float *y = new float[N];
  // initialize x and y on the CPU
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f; y[i] = 2.0f;
  }
  // Run on 1M elements on the CPU
  add(N, x, y);
  // Free memory
  delete [] x; delete [] y;
  return 0;
}
```

Running code on a GPU

1) Allocate memory on GPU
2) Copy data to GPU
3) Execute GPU program
4) Wait for completion
5) Copy results back to CPU

Running code on a serial GPU

```
float *x = new float[N];
float *y = new float[N];
int size = N*sizeof(float);
float *d_x, *d_y; // device copies of x y
cudaMalloc((void **)&d_x, size);
cudaMalloc((void **)&d_y, size);
// Run kernel on GPU
add<<<1,1>>>(d_x, d_y); // Only 1 thread
// Copy result back to host
cudaMemcpy(y, d_y, size, cudaMemcpyDeviceToHost);
// Free memory
cudaFree(d_x); cudaFree(d_y);
delete [] x; delete [] y;
```

Minimal extensions to 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
```

Special variables for thread identification in kernels

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

Intrinsics that expose specific operations in kernel code

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

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

Runtime functions

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(),

Running code on a parallel GPU

```
  float *x = new float[N];
  float *y = new float[N];
  int size = N*sizeof(float);
  float *d_x, *d_y; // device copies of x y
  cudaMalloc((void **)&d_x, size);
  cudaMalloc((void **)&d_y, size);
  // Run kernel on GPU
  add<<<1,256>>>(d_x, d_y); // 1 block of 256 threads *** Architecture bound!
  // Copy result back to host
  cudaMemcpy(y, d_y, size, cudaMemcpyDeviceToHost);
 // Free memory
  cudaFree(d_x); cudaFree(d_y);
  delete [] x; delete [] y;

// GPU function to add two vectors
__global__
void add(int n, float *x, float *y) {
  int index = threadIdx.x;
  y[index] = x[index] + y[index];
}
```

Hierarchical Parallelism Strategy

- Use both blocks and threads – Why?

- Hardware limit on maximum number of threads/block
- Threads alone won’t work for large arrays
- Fast shared memory only between threads
- Blocks alone are slower

Mapping CUDA to a GPU (1)

- 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 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 Streaming Multiprocessor (SM)
  - Peak efficiency requires multiple thread blocks per SM

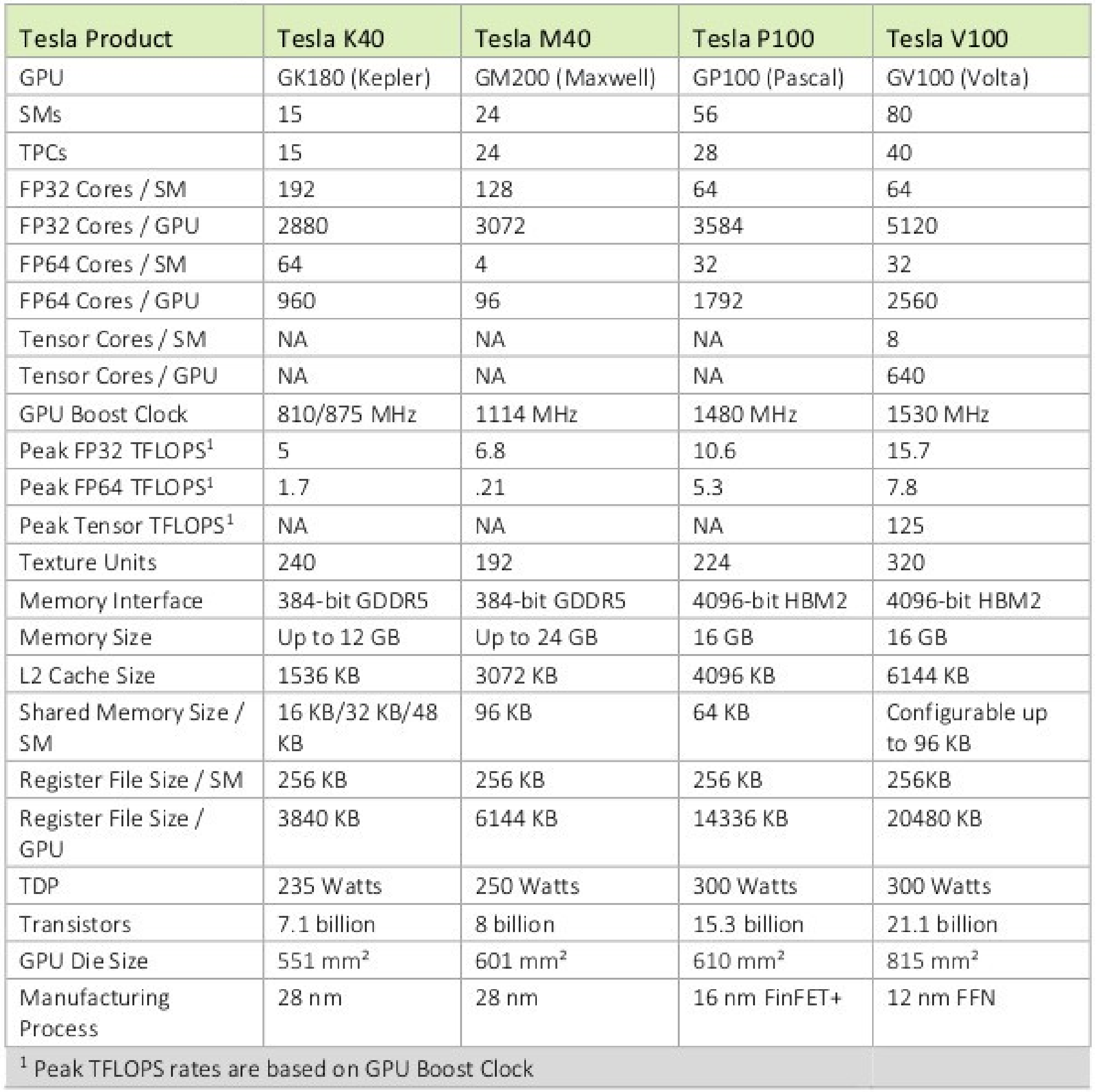
Mapping CUDA to a GPU (2)

- 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 Kepler, target 32 registers or less per thread for full occupancy

Occupancy (on Kepler)

- 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
- Each SM has scheduler space for 64 warps (W)
  - B * T ≤ W=64
- The number of threads per warp (V) is 32
  - B * T * V * Registers per thread ≤ 65536
  - B * Shared memory (bytes) per block ≤ 49152/16384
- Depending on Shared memory/L1 cache configuration
- Occupancy is reported as B * T / W

Nvidia Tesla series



Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	NA	NA	8
Tensor Cores / GPU	NA	NA	NA	640
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1530 MHz
Peak FP32 TFLOPS[1]	5	6.8	10.6	15.7
Peak FP64 TFLOPS[1]	1.7	.21	5.3	7.8
Peak Tensor TFLOPS[1]	NA	NA	NA	125
Texture Units	240	192	224	320
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB	6144 KB
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB	Configurable up to 96 KB
Register File Size / SM	256 KB	256 KB	256 KB	256KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB	20480 KB
TDP	235 Watts	250 Watts	300 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion	21.1 billion
GPU Die Size	551 mm²	601 mm²	610 mm²	815 mm²
Manufacturing Process	28 nm	28 nm	16 nm FinFET+	12 nm FFN

[1] Peak TFLOPS rates are based on GPU Boost Clock

GPU Memory

- Registers per thread
- Local cached memory per thread
- Shared memory (shared in block)
  - Declare using __shared__, allocated per block
  - Fast on-chip memory, user-managed
  - Not visible to threads in other blocks

- Global device level shared
- Constant cache shared by threads
- Texture cache shared by all blocks

- CPU access to global, constant and texture

Memory bounds

- “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
- Cache access patterns matter
  - Sparse access
  - Unaligned access

Memory 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* from different SIMD lanes 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 structures

- Multidimensional arrays are usually stored as monolithic vectors in memory
- Care should be taken to assure aligned memory accesses for the necessary access pattern
- Different data access patterns may also require transposing data structures (Arrays, structs)
- 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

Another example: 1D stencil

y[i] = x[i] + x[i-2] + x[i-1] + x[i+2] + x[i+1]

1D 5-point stencil (with a “radius” of 2)

- Each thread processes one output element
  - blockDim.x elements per block
- Input elements are read several times:
  - Radius of 2, each input element is read 5 times
  - Radius of 3, each input element is read 7 times

1D stencil: GPU thread strategy

- Divide output array into blocks, each assigned to a thread block
  - Each element within is assigned to a thread
  - Compute blockDim.x output elements
  - Write blockDim.x output elements to global memory
- Cache (manually) input data in shared memory
  - Have each block read (blockDim.x + 2 * radius) input elements from global memory to shared memory
  - Each block needs a *ghost region* of radius elements at each boundary

1D stencil: kernel

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;
  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {  // fill in ghost regions
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
  } // temp avoids using global memory over and over
 // Apply the stencil
  int result = 0;
  for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];
  // Store the result
  out[gindex] = result;
}
```

Race conditions: Synchronization

Suppose thread 7 (of 8) reads the ghost region before thread 0 has filled it in?

- Synchronizes all threads within a block
    void __syncthreads();
- Used to prevent RAW / WAR / WAW hazards
- All threads in the block must reach the barrier
- If used inside a conditional, the condition must be uniform across the block

Additional GPU Functions

- Double and single precision
- 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

GPU Conclusions

- GPUs gain efficiency from simpler cores and more parallelism
  - Very wide SIMD (SIMT) for parallel arithmetic and latency-hiding
- Heterogeneous programming with manual offload
  - CPU to run OS, etc. GPU for compute
- Massive (mostly data) parallelism required
  - Memmory coalescing helps
- Threads in block share faster memory and barriers
  - Blocks in kernel share slow device memory and atomics