A Quick Review: Classes of Parallelism

- ILP:
  - Run multiple instructions from one stream in parallel (e.g. pipelining)
- TLP:
  - Run multiple instruction streams simultaneously (e.g. openMP)
- DLP:
  - Run the same operation on multiple data at the same time (e.g. SSE intrinsics)

GPUs are here

GPUs

- Hardware specialized for graphics calculations
- Originally developed to facilitate the use of CAD programs
- Graphics calculations are extremely data parallel
- e.g. translate every vertex in a 3D model to the right
- Programmers found that that could rephrase some of their problems as graphics manipulations and run them on the GPU
- Incredibly burdensome for the programmer to use
- More usable these days – openCL, CUDA

CPU vs. GPU

- Latency optimized
- A couple threads of execution
- Each thread executes quickly
- Serial code
- Lots of caching

- Throughput optimized
- Many, many threads of execution
- Each thread executes slowly
- Parallel code
- Lots of memory bandwidth

OpenCL and CUDA

- Extensions to C which allow for relatively easy GPU programming
- CUDA is NVIDIA proprietary
  - NVIDIA cards only
- OpenCL is opensource
  - Can be used with NVIDA or ATI cards
  - Intended for general heterogeneous computing
    - Means you can use it with stuff like FPGAs
    - Also means it's relatively clunky

- Similar tools, but different jargon

Kernels

- Kernels define the computation for one array index
- The GPU runs the kernel on each index of a specified range
- Similar functionality to map, but you get to know the array index <u>and</u> the array value.
- Call the work at a given index a *work-item, a cuda thread*, or a *μthread.*
- The entire range is called an *index-space* or *grid.*

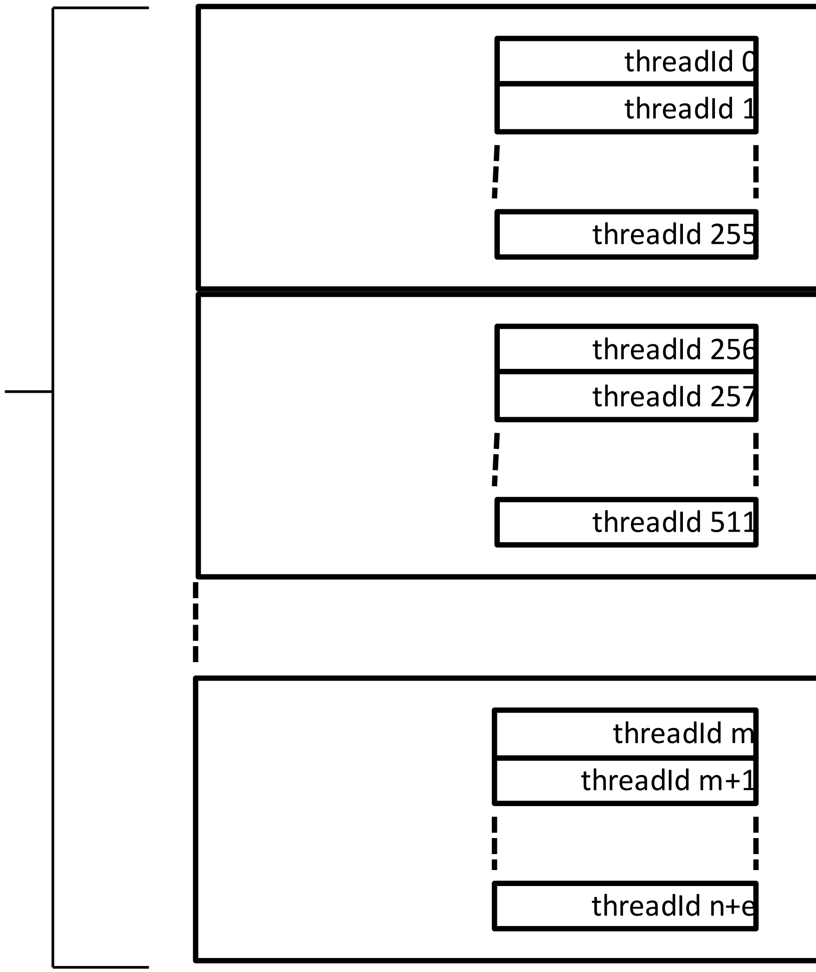
OpenCL vvadd

```
/* C version. */

void vvadd(float *dst, float *a, float *b, unsigned n) {

for(int i = 0; i < n; i++)

dst[i] = a[i] + b[i]

}

/* openCL Kernel. */

__kernel void vvadd(__global float *dst, __global float *a,
__global float *b, unsigned n) {

unsigned tid = get_global_id(0);

if (tid < n)

dst[tid] = a[tid] + b[tid];

}
```

Programmer's View of Execution

Create enough work groups to cover input vector

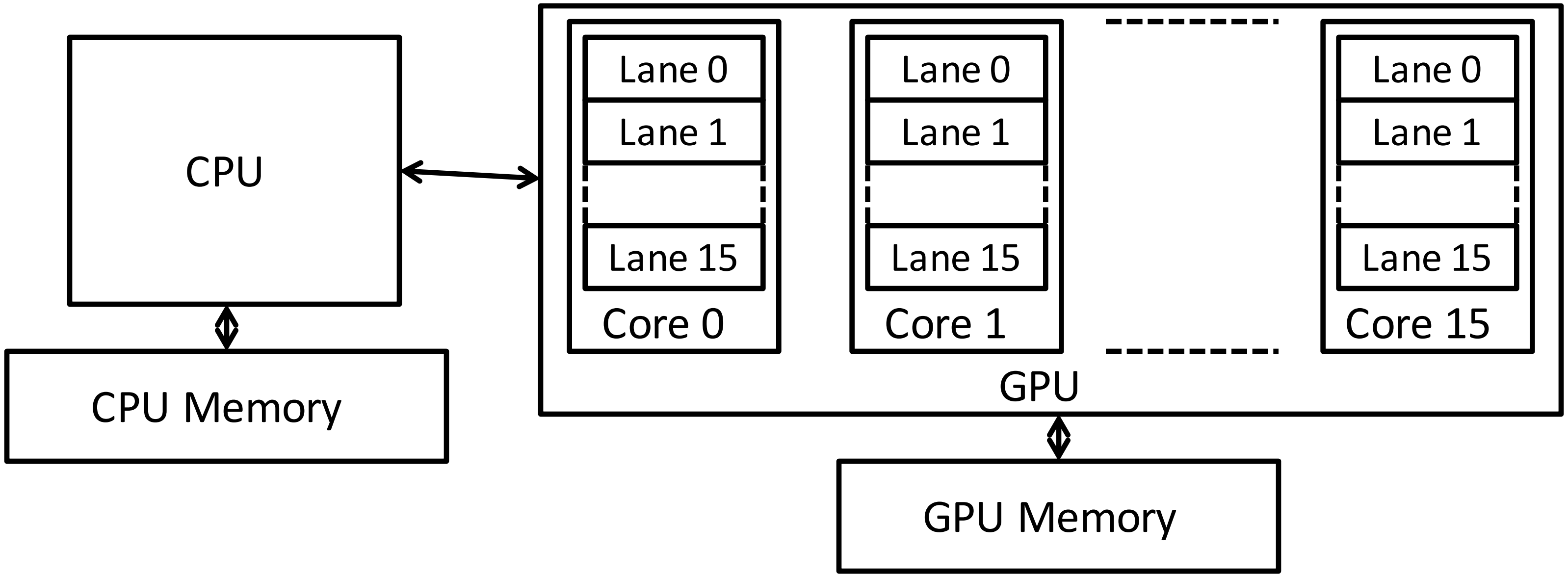
(openCL calls this ensemble of work groups an index space, can be 3-dimensional in openCL, 2 dimensional in CUDA)



Local work size (programmer can choose)

Conditional **(i<n)** turns off unused threads in last block

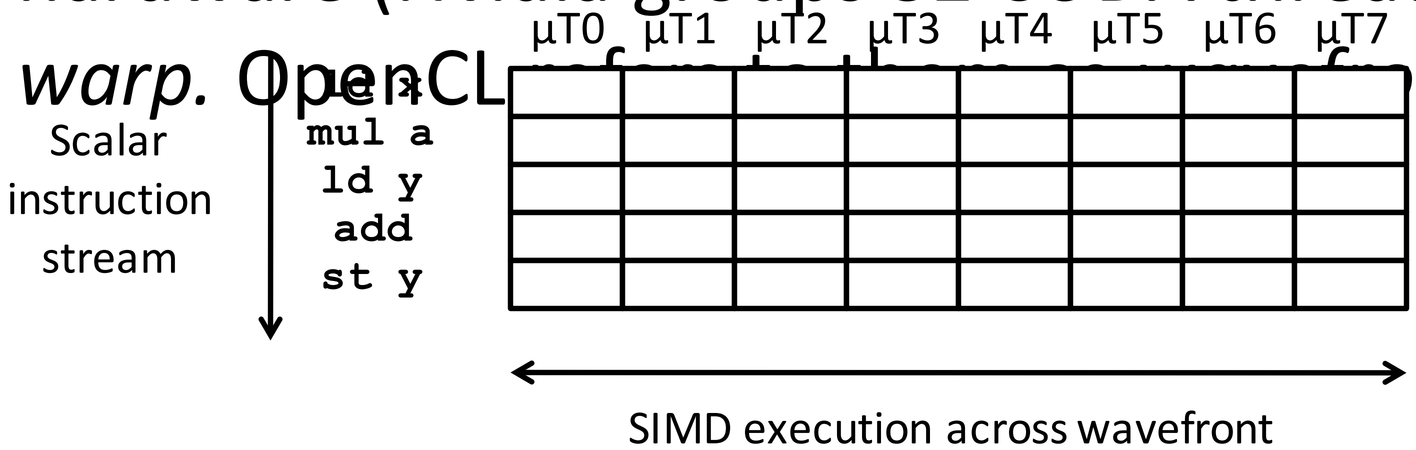
Hardware Execution Model



- GPU is built from multiple parallel cores, each core contains a multithreaded SIMD processor.
- CPU sends whole index-space over to GPU, which distributes work-groups among cores (each work-group executes on one core)

“Single Instruction, Multiple Thread”

- GPUs use a SIMT model, where individual scalar instruction streams for each work item are grouped together for SIMD execution on hardware (Nvidia groups 32 CUDA threads into a *warp. *OpenCL refers to them as *wavefronts*.)



Teminology Summary

- Kernel: The function that is mapped across the input.
- Work-item: The basic unit of execution. Takes care of one index. Also called a microthread or cuda thread.
- Work-group/Block: A group of work-items. Each work-group is sent to one core in the GPU.
- Index-space/Grid: The range of indices over which the kernel is applied.
- Wavefront/Warp: A group of microthreads (work-items) scheduled to be SIMD executed with eachother.

Administrivia

- Homework 4 is due Sunday (April 6th)

Conditionals in the SIMT Model

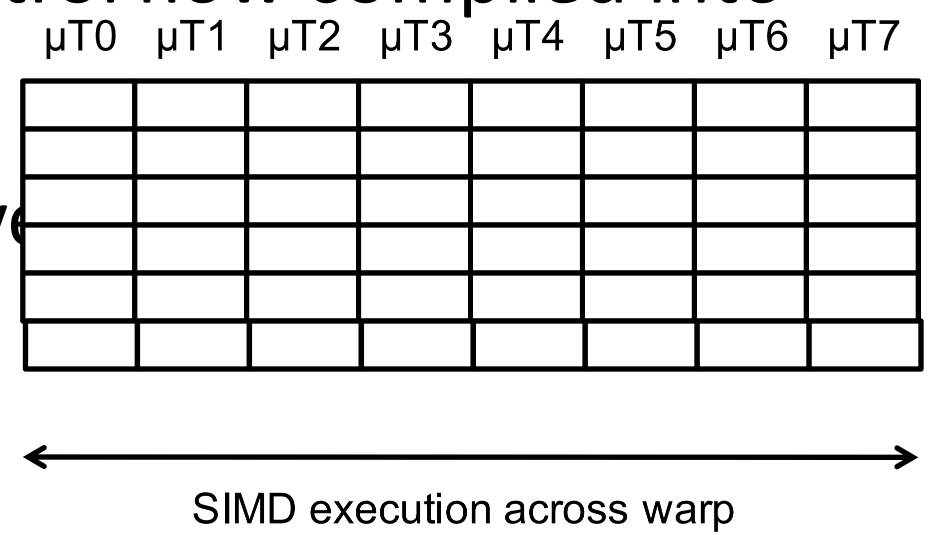
- Simple if-then-else are compiled into predicated execution, equivalent to vector masking

- More complex control flow compiled into branches

- How to execute a ve

Scalar instruction stream

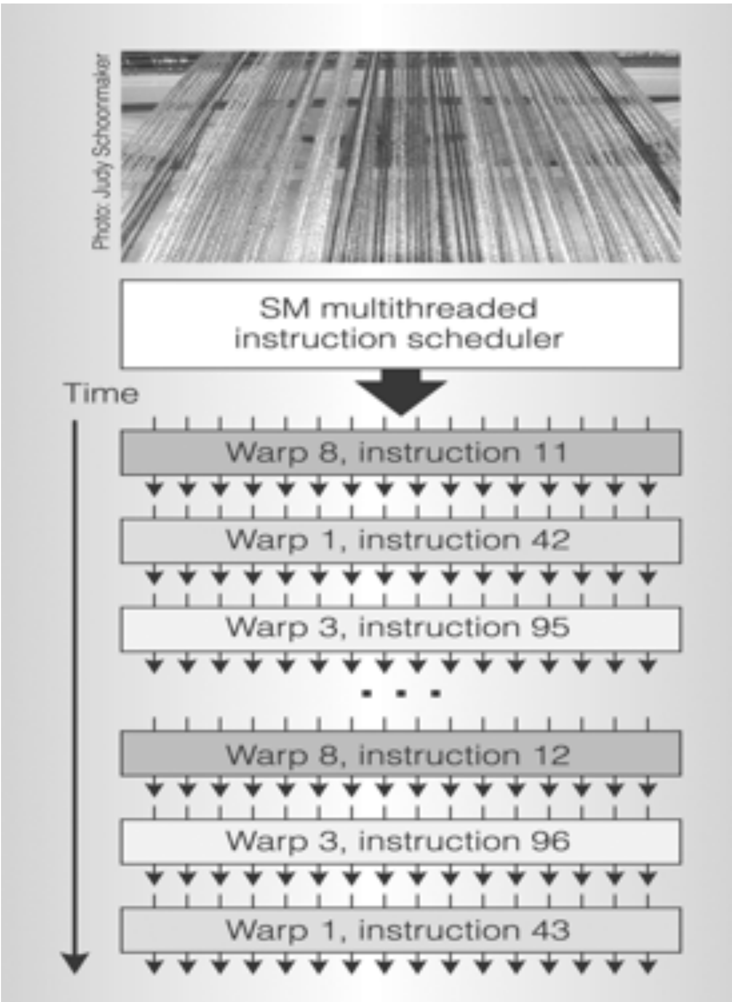
```
tid=threadid
If (tid >= n) skip
   Call func1
      add
      st y
skip:
```



Branch Divergence

- Hardware tracks which µthreads take or don’t take branch
- If all go the same way, then keep going in SIMD fashion
- If not, create mask vector indicating taken/not-taken
- Keep executing not-taken path under mask, push taken branch PC+mask onto a hardware stack and execute later
- When can execution of µthreads in warp

Warps (wavefronts) are multithreaded on a single core

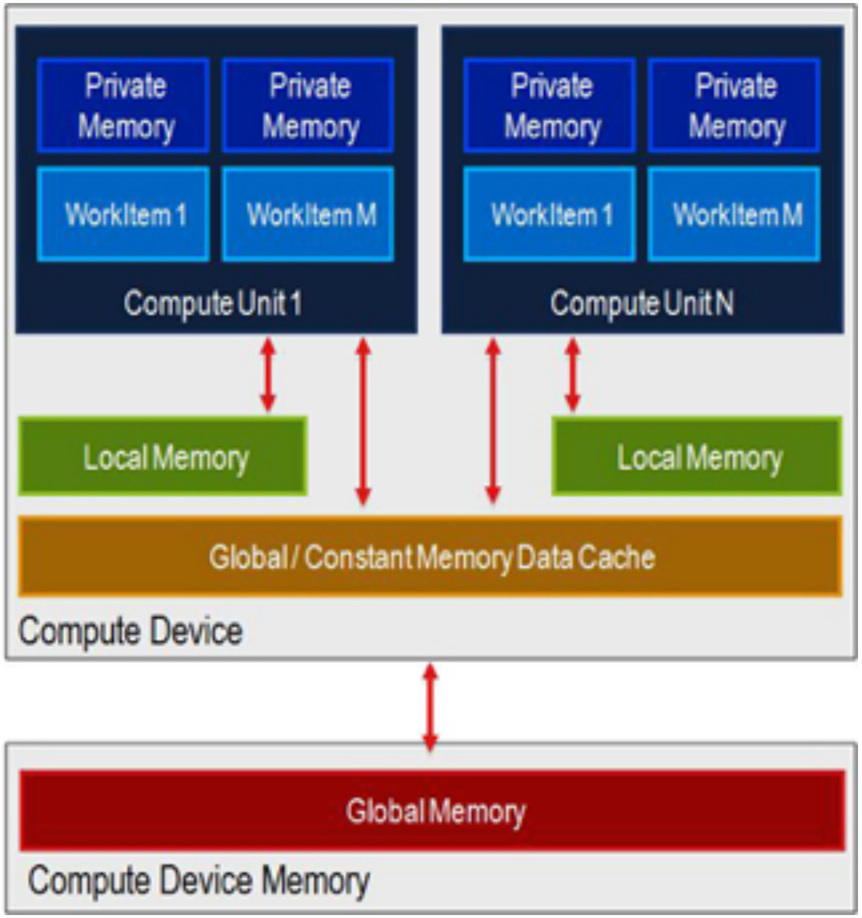


[Nvidia, 2010]

- One warp of 32 µthreads is a single thread in the hardware
- Multiple warp threads are interleaved in execution on a single core to hide latencies (memory and functional unit)
- A single thread block can contain multiple warps (up

OpenCL Memory Model

- Global – read and write by all work-items and work-groups
- Constant – read-only by work-items; read and write by host
- Local – used for data sharing; read/write by work-items in the same work group



SIMT

- Illusion of many independent threads
- But for efficiency, programmer must try and keep µthreads aligned in a SIMD fashion
- Try to do unit-stride loads and store so memory coalescing kicks in
- Avoid branch divergence so most instruction slots execute useful work and are not masked off

VVADD

```
/* C version. */
void vvadd(float *dst, float *a, float *b, unsigned n) {
#pragma omp parallel for
for(int i = 0; i < n; i++)
dst[i] = a[i] + b[i]
}

/* openCL Kernel. */
__kernel void vvadd(__global float *dst, __global float *a,
unsigned tid = get_global_id(0);
if (tid < n)
dst[tid] = a[tid] + b[tid];
}
```

A: CPU faster

B: GPU faster

VVADD

```
/* C version. */
void vvadd(float *dst, float *a, float *b, unsigned n) {
#pragma omp parallel for
for(int i = 0; i < n; i++)
dst[i] = a[i] + b[i]
}
```

- Only 1 flop per three memory accesses => mem
- “A many core processor ≡ A device for turning a con

VECTOR_COP

```
/* C version. */
void vector_cop(float *dst, float *a, float *b, unsigned n) {
#pragma omp parallel for
for(int i = 0; i < n; i++) {
dst[i] = 0;
for (int j = 0; j < A_LARGE_NUMBER; j++)
dst[i] += a[i]*2*b[i] – a[i]*a[i] – b[i]*b[i];
}
}

/* OpenCL kernel. */
__kernel void vector_cop(__global float *dst, __global float *a,
                         __global float *b, unsigned n) {
unsigned i = get_global_id(0);
if (tid < n) {
dst[i] = 0;
for (int j = 0; j < A_LARGE_NUMBER; j++)
dst[i] += a[i]*2*b[i] – a[i]*a[i] – b[i]*b[i];
•}
•}
```

A: CPU faster

B: GPU faster

GP-GPU in the future

- High-end desktops have separate GPU chip, but trend towards integrating GPU on same die as CPU (already in laptops, tablets and smartphones)
- Advantage is shared memory with CPU, no need to transfer data
- Disadvantage is reduced memory bandwidth compared to dedicated smaller-capacity specialized memory system
  - Graphics DRAM (GDDR) versus regular DRAM (DDR3)

Acknowledgements

- These slides contain materials developed and copryright by
- Krste Asanovic (UCB)
- AMD
- codeproject.com

And in conclusion…

- GPUs thrive when
- The calculation is data parallel
- The calculation is CPU-bound
- The calculation is large
- CPUs thrive when
- The calculation is largely serial
- The calculation is small
- The programmer is lazy

Bonus

- OpenCL source code for vvadd and vector_cop demos available at
  - http://www-inst.eecs.berkeley.edu/~cs61c/sp13/lec/39/demo.tar.gz