CS 152 Computer Architecture and Engineering

Lecture 16: Graphics Processing Units (GPUs)

Krste Asanovic
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~krste
http://inst.cs.berkeley.edu/~cs152
Last Time: Vector Computers

- Vectors provide efficient execution of data-parallel loop codes
- Vector ISA provides compact encoding of machine parallelism
- Vector ISA scales to more lanes without changing binary code
- Vector registers provide fast temporary storage to reduce memory bandwidth demands, & simplify dependence checking between vector instructions
- Scatter/gather, masking, compress/expand operations increase set of vectorizable loops
- Requires extensive compiler analysis (or programmer annotation) to be certain that loops can be vectorized
- Full “long” vector support (vector length control, scatter/gather) still only in supercomputers (NEC SX9, Cray X1E); microprocessors have limited packed or subword-SIMD operations
  - Intel x86 MMX/SSE/AVX
  - IBM/Motorola PowerPC VMX/Altivec
Multimedia Extensions (aka SIMD extensions)

<table>
<thead>
<tr>
<th>64b</th>
<th>32b</th>
<th>32b</th>
</tr>
</thead>
<tbody>
<tr>
<td>16b</td>
<td>16b</td>
<td>16b</td>
</tr>
<tr>
<td>8b</td>
<td>8b</td>
<td>8b</td>
</tr>
</tbody>
</table>

- Very short vectors added to existing ISAs for microprocessors
- Use existing 64-bit registers split into 2x32b or 4x16b or 8x8b
  - Lincoln Labs TX-2 from 1957 had 36b datapath split into 2x18b or 4x9b
  - Newer designs have wider registers
    - 128b for PowerPC Altivec, Intel SSE2/3/4
    - 256b for Intel AVX
- Single instruction operates on all elements within register

4x16b adds
Multimedia Extensions versus Vectors

• Limited instruction set:
  – no vector length control
  – no strided load/store or scatter/gather
  – unit-stride loads must be aligned to 64/128-bit boundary

• Limited vector register length:
  – requires superscalar dispatch to keep multiply/add/load units busy
  – loop unrolling to hide latencies increases register pressure

• Trend towards fuller vector support in microprocessors
  – Better support for misaligned memory accesses
  – Support of double-precision (64-bit floating-point)
  – New Intel AVX spec (announced April 2008), 256b vector registers (expandable up to 1024b)
Types of Parallelism

- **Instruction-Level Parallelism (ILP)**
  - Execute independent instructions from one instruction stream in parallel (pipelining, superscalar, VLIW)

- **Thread-Level Parallelism (TLP)**
  - Execute independent instruction streams in parallel (multithreading, multiple cores)

- **Data-Level Parallelism (DLP)**
  - Execute multiple operations of the same type in parallel (vector/SIMD execution)

- Which is easiest to program?
- Which is most flexible form of parallelism?
  - i.e., can be used in more situations
- Which is most efficient?
  - i.e., greatest tasks/second/area, lowest energy/task
Resurgence of DLP

• Convergence of application demands and technology constraints drives architecture choice

• New applications, such as graphics, machine vision, speech recognition, machine learning, etc. all require large numerical computations that are often trivially data parallel

• SIMD-based architectures (vector-SIMD, subword-SIMD, SIMT/GPUs) are most efficient way to execute these algorithms
DLP important for conventional CPUs too

- Prediction for x86 processors, from Hennessy & Patterson, 5th edition
  - Note: Educated guess, not Intel product plans!
- TLP: 2+ cores / 2 years
- DLP: 2x width / 4 years
- DLP will account for more mainstream parallelism growth than TLP in next decade.
  - SIMD – single-instruction multiple-data (DLP)
  - MIMD - multiple-instruction multiple-data (TLP)
Graphics Processing Units (GPUs)

• Original GPUs were dedicated fixed-function devices for generating 3D graphics (mid-late 1990s) including high-performance floating-point units
  – Provide workstation-like graphics for PCs
  – User could configure graphics pipeline, but not really program it

• Over time, more programmability added (2001-2005)
  – E.g., New language Cg for writing small programs run on each vertex or each pixel, also Windows DirectX variants
  – Massively parallel (millions of vertices or pixels per frame) but very constrained programming model

• Some users noticed they could do general-purpose computation by mapping input and output data to images, and computation to vertex and pixel shading computations
  – Incredibly difficult programming model as had to use graphics pipeline model for general computation
General-Purpose GPUs (GP-GPUs)

• In 2006, Nvidia introduced GeForce 8800 GPU supporting a new programming language: CUDA
  – “Compute Unified Device Architecture”
  – Subsequently, broader industry pushing for OpenCL, a vendor-neutral version of same ideas.

• Idea: Take advantage of GPU computational performance and memory bandwidth to accelerate some kernels for general-purpose computing

• Attached processor model: Host CPU issues data-parallel kernels to GP-GPU for execution

• This lecture has a simplified version of Nvidia CUDA-style model and only considers GPU execution for computational kernels, not graphics
  – Would probably need another course to describe graphics processing
Simplified CUDA Programming Model

- Computation performed by a very large number of independent small scalar threads (CUDA threads or microthreads) grouped into thread blocks.

```c
// C version of DAXPY loop.
void daxpy(int n, double a, double* x, double* y)
{
    for (int i=0; i<n; i++)
        y[i] = a*x[i] + y[i];
}

// CUDA version.
__host__  // Piece run on host processor.
int nblocks = (n+255)/256; // 256 CUDA threads/block
daxpy<<<nblocks,256>>>(n,2.0,x,y);

__device__  // Piece run on GP-GPU.
void daxpy(int n, double a, double* x, double* y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i<n) y[i]=a*x[i]+y[i];
}```
Programmer’s View of Execution

Create enough blocks to cover input vector

(Nvidia calls this ensemble of blocks a Grid, can be 2-dimensional)

blockIdx 0
  threadId 0
  threadId 1
  threadId 255

blockIdx 1
  threadId 0
  threadId 1
  threadId 255

blockIdx (n+255/256)
  threadId 0
  threadId 1
  threadId 255

blockDim = 256 (programmer can choose)

Conditional (i < n) turns off unused threads in last block
GPU Hardware Execution Model

- GPU is built from multiple parallel cores, each core contains a multithreaded SIMD processor with multiple lanes but with no scalar processor
- CPU sends whole “grid” over to GPU, which distributes thread blocks among cores (each thread block executes on one core)
  - Programmer unaware of number of cores
“Single Instruction, Multiple Thread”

• GPUs use a SIMT model, where individual scalar instruction streams for each CUDA thread are grouped together for SIMD execution on hardware (Nvidia groups 32 CUDA threads into a warp)
Implications of SIMT Model

- All “vector” loads and stores are scatter-gather, as individual µthreads perform scalar loads and stores
  - GPU adds hardware to dynamically coalesce individual µthread loads and stores to mimic vector loads and stores
- Every µthread has to perform stripmining calculations redundantly ("am I active?") as there is no scalar processor equivalent
CS152 Administrivia

• Quiz 2 results
• Quiz 3, Complex Pipelining, L10-12, Lab 3, PS 3, readings
Conditionals in 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 vector of branches?

Scalar instruction stream

\[
\text{tid} = \text{threadid}
\]

If (\text{tid} \geq n) skip

Call \text{func1}

add

st y

skip:

SIMD execution across warp

\[
\begin{array}{cccccccc}
\mu T0 & \mu T1 & \mu T2 & \mu T3 & \mu T4 & \mu T5 & \mu T6 & \mu T7 \\
\end{array}
\]
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 reconverge?
Warps are multithreaded on core

- 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 to 512 μT max in CUDA), all mapped to single core
- Can have multiple blocks executing on one core

[Image: SIMT multithreaded instruction scheduler]

[Informative Diagram: Warps execution on a core]

[Source: Nvidia, 2010]
GPU Memory Hierarchy

[ Nvidia, 2010]
SIMT

- Illusion of many independent threads
- But for efficiency, programmer must try and keep threads aligned in a SIMD fashion
  - Try and 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
Nvidia Fermi GF100 GPU

[Nvidia, 2010]
Fermi “Streaming Multiprocessor” Core
Fermi Dual-Issue Warp Scheduler
Apple A5X Processor for iPad v3 (2012)

- 12.90mm x 12.79mm
- 45nm technology

[Source: Chipworks, 2012]
Historical Retrospective, Cray-2 (1985)

- 243MHz ECL logic
- 2GB DRAM main memory (128 banks of 16MB each)
  - Bank busy time 57 clocks!
- Local memory of 128KB/core
- 1 foreground + 4 background vector processors
GPU 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)

• Will GP-GPU survive? Or will improvements in CPU DLP make GP-GPU redundant?
  – On same die, CPU and GPU should have same memory bandwidth
  – GPU might have more FLOPS as needed for graphics anyway
Acknowledgements

- These slides contain material developed and copyright by:
  - Krste Asanovic (UCB)