Vector, SIMD Extensions and GPU

COMP 4611

Tutorial 12

Introduction

- **SIMD architectures can exploit significant data-level parallelism for:**
  - Matrix-oriented scientific computing
  - Media-oriented image and sound processors
- **SIMD can be more energy efficient than MIMD**
  - Only needs to fetch one instruction per data operation
  - Makes SIMD attractive for personal mobile devices
- **SIMD allows programmer to continue to think sequentially**

SIMD Parallelism

- Vector architectures
- SIMD extensions
- Graphics Processor Units (GPUs)

- For x86 processors:
  - In the previous decade, roughly
    - Two additional cores per chip per year
    - SIMD width doubles every four years
  - If the rates are the similar in the next decade
    - Potential speedup from SIMD to be twice that from MIMD

Vector Architectures

- Basic idea:
  - Read sets of data elements into "vector registers"
  - Operate on those registers
    - A single instruction -> dozens of operations on independent data elements
  - Disperse the results back into memory

- Registers are controlled by compiler
  - Used to hide memory latency
  - Leverage memory bandwidth

VMIPS

Example architecture: VMIPS

- Loosely based on Cray-1
- Vector registers
  - Each register holds a 64-element, 64 bits/element vector
  - Register file has 16 read ports and 8 write ports
- Vector functional units
  - Fully pipelined
  - Data and control hazards are detected
- Vector load-store unit
  - Fully pipelined
  - One word per clock cycle after initial latency
- Scalar registers
  - 32 general-purpose registers
  - 32 floating-point registers

VMIPS Instructions

- ADDVV.D: add two vectors
- MULVS.D: multiply each element of a vector by a scalar
- LV/SV: vector load and vector store from address

- Example: DAXPY
  - **Y = a x X + Y**, X and Y are vectors, a is a scalar

```
L.D     F0,a        ; load scalar a
LV      V1,Rx       ; load vector X
MULVS.D V2,V1,F0    ; vector-scalar multiply
LV      V3,Ry       ; load vector Y
ADDVV   V4,V2,V3    ; add
SV      Ry,V4       ; store the result
```

- Requires 6 instructions vs. almost 600 for MIPS

Vector Execution Time

- Execution time depends on three factors:
  - Length of operand vectors
  - Structural hazards
  - Data dependencies
- VMIPS functional units consume one element per clock cycle
  - Execution time is approximately the vector length
- *Convoy*
  - Set of vector instructions that could potentially execute together

Chimes

- Sequences with read-after-write dependency hazards can be in the same convoy via *chaining*
- *Chaining*
  - Allows a vector operation to start as soon as the individual elements of its vector source operand become available
- *Chime*
  - Unit of time to execute one convoy
  - *m* convoys executes in *m* chimes
  - For vector length of *n*, requires *m* x *n* clock cycles

Example

```
LV      V1,Rx     ;load vector X
MULVS.D V2,V1,F0  ;vector-scalar multiply
LV      V3,Ry     ;load vector Y
ADDVV.D V4,V2,V3  ;add two vectors
SV      Ry,V4     ;store the sum
```

Convoys:

```
1  LV      MULVS.D
2  LV      ADDVV.D
3  SV
```

3 chimes, 2 FP ops per result, cycles per FLOP = 1.5

For 64 element vectors, requires 64 x 3 = 192 clock cycles

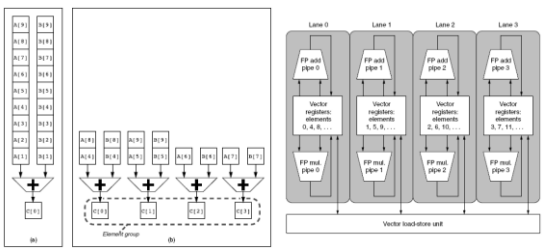
Challenges

- **Start up time**
  - Latency of vector functional unit
  - Assume the same as Cray-1
    - Floating-point add => 6 clock cycles
    - Floating-point multiply => 7 clock cycles
    - Floating-point divide => 20 clock cycles
    - Vector load => 12 clock cycles
- **Improvements:**
  - **> 1 element per clock cycle**
  - **IF statements in vector code**
  - Non-64 wide vectors
  - Memory system optimizations to support vector processors
  - Multiple dimensional matrices
  - Sparse matrices
  - Programming a vector computer

Multiple Lanes

Element *n* of vector register *A* is "hardwired" to element *n* of vector register *B*

- Allows for multiple hardware lanes



Vector Mask Registers

- Consider:

```
for (i = 0; i < 64; i=i+1)
    if (X[i] != 0)
        X[i] = X[i] – Y[i];
```

- Use vector mask register to "disable" elements:

```
LV      V1,Rx     ;load vector X into V1
LV      V2,Ry     ;load vector Y
L.D     F0,#0     ;load FP zero into F0
SNEVS.D V1,F0     ;sets VM(i) to 1 if V1(i)!=F0
SUBVV.D V1,V1,V2  ;subtract under vector mask
SV      Rx,V1     ;store the result in X
```

- Compilers manipulate mask registers explicitly

SIMD Extensions

- Media applications operate on data types narrower than the native word size
  - Example: disconnect carry chains to "partition" 64-bit adder --> eight 8-bit elements
- Limitations, compared to vector instructions:
  - Number of data operands encoded into op code
  - No sophisticated addressing modes
  - No mask registers

SIMD Implementations

Implementations:

- Intel MMX (1996)
  - Eight 8-bit integer ops or four 16-bit integer ops
- Streaming SIMD Extensions (SSE) (1999)
  - Eight 16-bit integer ops
  - Four 32-bit integer/fp ops or two 64-bit integer/fp ops
- Advanced Vector Extensions (2010)
  - Four 64-bit integer/fp ops

Graphical Processing Units

Given the hardware invested to do graphics well,

how can we supplement it to improve performance of a wider range of applications?

GPU Accelerated or Many-core Computation

- NVIDIA GPU
  - Reliable performance, communication between CPU and GPU using the PCIE channels
  - Tesla M2090
    - 512 CUDA cores (16 Streaming Multiprocessor x 32 cores/SM)
    - 1331 GFLOPs (single precision)
    - 6GB memory with 177GB/sec bandwidth

GPU Accelerated or Many-core Computation (cont.)

- APU (Accelerated Processing Unit) – AMD
  - CPU and GPU functionality on a single chip
  - AMD FirePro A320
    - 4 CPU cores and 384 GPU processors
    - 736 GFLOPs (single precision) @ 100W
    - Up to 2GB dedicated memory

GPU Accelerated or Many-core Computation (cont.)

- MIC (Many Integrated Core) – Intel
  - Many CPU cores on a single chip
  - Intel Xeon Phi Coprocessor 5110P
    - 60 cores/1.053 GHz/240 threads
    - 8 GB memory and 320 GB/s bandwidth
    - 829 GFLOPs (double precision) @ 195W (Intel Xeon Phi coprocessor SE10P)

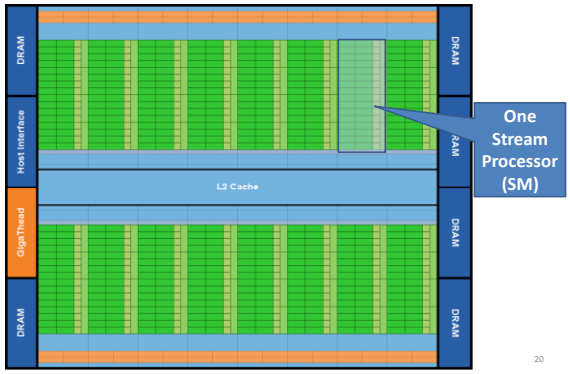
Easier programming, lower power consumption, and better integration of the CPU and GPU capabilities

Programming the NVIDIA GPU

We use NVIDIA GPU as the example for discussion

- Heterogeneous execution model
  - CPU is the *host*, GPU is the *device*
- Develop a programming language (CUDA) for GPU
- Unify all forms of GPU parallelism as *CUDA thread*
- Programming model is "Single Instruction Multiple Thread"

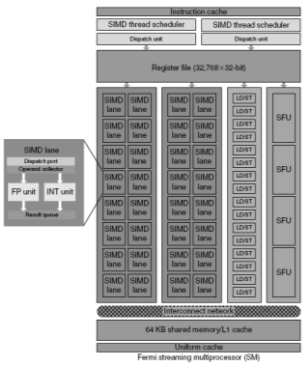
Fermi GPU



NVIDIA GPU Memory Structures

- Each SIMD Lane has private section of off-chip DRAM
  - "Private memory"
- Each multithreaded SIMD processor also has local memory
  - Shared by SIMD lanes
- Memory shared by SIMD processors is GPU Memory
  - Host can read and write GPU memory

Fermi Multithreaded SIMD Proc.



- Each Streaming Processor (SM)
  - –32 SIMD lanes
  - –16 load/store units
  - –32,768 registers
- Each SIMD thread has up to
  - –64 vector registers of 32 32-bit elements

GPU Compared to Vector

- Similarities to vector machines:
  - Works well with data-level parallel problems
  - Mask registers
  - Large register files
- Differences:
  - No scalar processor
  - Uses multithreading to hide memory latency
  - Has many functional units, as opposed to a few deeply pipelined units like a vector processor

Introduction to Fermi from NVIDIA

http://www.nvidia.com/object/fermi-architecture.html

Questions?