Data-Parallel Architectures

Nima Honarmand
Overview

• **Data-Level Parallelism (DLP) vs. Thread-Level Parallelism (TLP)**
  – In DLP, parallelism arises from independent execution of the same code on a large number of data objects
  – In TLP, parallelism arises from independent execution of different threads of control

• Hypothesis: many applications that use massively parallel machines exploit data parallelism
  – Common in the Scientific Computing domain
  – Also, multimedia (image and audio) processing
  – And more recently data mining and AI
Interlude: Flynn’s Taxonomy (1966)

• Michael Flynn classified parallelism across two dimensions: Data and Control
  – Single Instruction, Single Data (SISD)
    • Our uniprocessors
  – Single Instruction, Multiple Data (SIMD)
    • Same inst. executed by different “processors” using different data
    • Basis of DLP architectures: vector, SIMD extensions, GPUs
  – Multiple Instruction, Multiple Data (MIMD)
    • TLP architectures: SMPs and multi-cores
  – Multiple Instruction, Single Data (MISD)
    • Just for the sake of completeness, no real architecture

• DLP originally associated w/ SIMD; now SIMT is also common
  – SIMT: Single Instruction Multiple Threads
  – SIMT found in NVIDIA GPUs
Examples of Data-Parallel Code

- **SAXPY**: $Y = a \times X + Y$
  
  ```
  for (i = 0; i < n; i++)
      Y[i] = a * X[i] + Y[i]
  ```

- **Matrix-Vector Multiplication**: $A_{m \times 1} = M_{m \times n} \times V_{n \times 1}$
  
  ```
  for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
  ```
Overview

• Many incarnations of DLP architectures over decades
  – Vector processors
    • Cray processors: Cray-1, Cray-2, ..., Cray X1
  – SIMD extensions
    • Intel MMX, SSE* and AVX* extensions
  – Modern GPUs
    • NVIDIA, AMD, Qualcomm, ...

• General Idea: use statically-known DLP to achieve higher throughput
  – instead of discovering parallelism in hardware as OOO super-scalars do
  – Focus on throughput rather than latency
Vector Processors
Vector Processors

• Basic idea:
  – Read sets of data elements into “vector registers”
  – Operate on those registers
  – Disperse the results back into memory

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

• Hide memory latency by:
  – Issuing all memory accesses for a vector load/store together
  – Using chaining (later) to compute on earlier vector elements while waiting for later elements to be loaded
Scalar processors operate on single numbers (scalars)
Vector processors operate on linear sequences of numbers (vectors)
Components of a Vector Processor

• A scalar processor (e.g. a MIPS processor)
  – Scalar register file (32 registers)
  – Scalar functional units (arithmetic, load/store, etc)

• A vector register file (a 2D register array)
  – Each register is an array of elements
  – E.g. 32 registers with 32 64-bit elements per register
  – MVL = maximum vector length = max # of elements per register

• A set of vector functional units
  – Integer, FP, load/store, etc
  – Some times vector and scalar units are combined (share ALUs)
Simple Vector Processor Organization
## Basic Vector ISA

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>vadd.vv</td>
<td>v1=v2+v3</td>
<td>vector + vector</td>
</tr>
<tr>
<td>vadd.sv</td>
<td>v1=r0+v2</td>
<td>scalar + vector</td>
</tr>
<tr>
<td>vmul.vv</td>
<td>v1=v2*v3</td>
<td>vector x vector</td>
</tr>
<tr>
<td>vmul.sv</td>
<td>v1=r0*v2</td>
<td>scalar x vector</td>
</tr>
<tr>
<td>vld</td>
<td>v1=m[r1...r1+63]</td>
<td>load, stride=1</td>
</tr>
<tr>
<td>vlds</td>
<td>v1=m[r1...r1+63*r2]</td>
<td>load, stride=r2</td>
</tr>
<tr>
<td>vldx</td>
<td>v1=m[r1+v2[i], i=0..63]</td>
<td>indexed load (gather)</td>
</tr>
<tr>
<td>vst</td>
<td>m[r1...r1+63]=v1</td>
<td>store, stride=1</td>
</tr>
<tr>
<td>vsts</td>
<td>v1=m[r1...r1+63*r2]</td>
<td>store, stride=r2</td>
</tr>
<tr>
<td>vstx</td>
<td>v1=m[r1+v2[i], i=0..63]</td>
<td>indexed store (scatter)</td>
</tr>
</tbody>
</table>

+ regular scalar instructions
SAXPY in Vector ISA vs. Scalar ISA

• For now, assume array length = vector length (say 32)

```
loop:
  fld f1, 0(x5) # load x[i]
  fmul f1, f1, f0 # a * X[i]
  fld f2, 0(x6) # load Y[i]
  fadd f2, f2, f1 # a * X[i] + Y[i]
  fst f2, 0(x6) # store Y[i]
  addi x5, x5, 4 # increment X index
  addi x6, x6, 4 # increment Y index
  bne x28, x5, loop # check if done
```

```
fld f0, a # load scalar a
addi x28, x5, 4*32 # last addr to load

fld f0, a # load scalar a
vld v0, x5 # load vector X
Vmul v1, f0, v0 # vector-scalar multiply
vld v2, x6 # load vector Y
vadd v3, v1, v2 # vector-vector vector add
vst v3, x6 # store the sum in Y
```
Vector Length (VL)

- Usually, array length not equal to (or a multiple of) maximum vector length (MVL)
- Can **strip-mine** the loop to make inner loops a multiple of MVL, and use an explicit VL register for the remaining part

```c
for (j = 0; j < n; j += mvl)
    for (i = j; i < mvl; i++)
        Y[i] = a * X[i] + Y[i];
for (; i < n; i++)
    Y[i] = a * X[i] + Y[i];

Strip-mined C code
```

```assembly
fld f0, a   # load scalar a
Loop:       setvl x1   # set VL = min(n, mvl)
vld v0, x5  # load vector X
Vmul v1, f0, v0  # vector-scalar multiply
vld v2, x6  # load vector Y
vadd v3, v1, v2  # vector-vector add
vst v3, x6  # store the sum in Y
// decrement x1 by VL
// increment x5, x6 by VL
// jump to Loop if x1 != 0
```

Strip-mined Vector code
Advantages of Vector ISA

• **Compact:** single instruction defines N operations
  – Amortizes the cost of instruction fetch/decode/issue
  – Also reduces the frequency of branches

• **Parallel:** N operations are (data) parallel
  – No dependencies
  – No need for complex hardware to detect parallelism
  – Can execute in parallel assuming N parallel functional units

• **Expressive:** memory operations describe patterns
  – Continuous or regular memory access pattern
  – Can prefetch or accelerate using wide/multi-banked memory
  – Can amortize high latency for 1st element over large sequential pattern
Optimization 1: Chaining

- Consider the following code:
  
  ```
  vld v3, r4
  vmul.sv v6, r5, v3  # very long RAW hazard
  vadd.vv v4, v6, v5  # very long RAW hazard
  ```

  - **Chaining:**
    - v1 is not a single entity but a group of individual elements
    - vmul can start working on individual elements of v1 as they become ready
    - Same for v6 and vadd

- Can allow any vector operation to chain to any other active vector operation
  - By having register files with many read/write ports

```
Unchained Execution
```

```
Chained Execution
```

```
vmul  vadd
```

```
vmul  vadd
```

```
vmul  vadd
```
Modular, scalable design

- Elements for each vector register interleaved across the lanes
- Each lane receives identical control
- Multiple element operations executed per cycle
- No need for inter-lane communication for most vector instructions
Chaining & Multi-lane Example

VL=16, 4 lanes, 2 FUs, 1 LSU

chaining -> 12 ops/cycle

Just 1 new instruction issued per cycle

!!!

Element Operations: ◆ ▼ ▲

Instr. Issue: ◄
Optimization 3: Vector Predicates

• Suppose you want to vectorize this:
  for (i=0; i<N; i++)
    if (A[i]!= B[i]) A[i] -= B[i];

• Solution: vector conditional execution (predication)
  – Add vector flag registers with single-bit elements (masks)
  – Use a vector compare to set the a flag register
  – Use flag register as mask control for the vector sub
    • Do subtraction only for elements w/ corresponding flag set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operations</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>vld</td>
<td>v1, x5</td>
<td># load A</td>
</tr>
<tr>
<td>vld</td>
<td>v2, x6</td>
<td># load B</td>
</tr>
<tr>
<td>vcmp.neq.vv</td>
<td>m0, v1, v2</td>
<td># vector compare</td>
</tr>
<tr>
<td>vsub.vv</td>
<td>v1, v1, v2, m0</td>
<td># conditional vsub</td>
</tr>
<tr>
<td>vst</td>
<td>v1, x5, m0</td>
<td># store A</td>
</tr>
</tbody>
</table>
Strided Vector Load/Stores

• Consider the following matrix-matrix multiplication:

```plaintext
for (i = 0; i < 100; i=i+1)
    for (j = 0; j < 100; j=j+1)
        A[i][j] = 0.0;
    for (k = 0; k < 100; k=k+1)
```

• Can vectorize multiplication of rows of B with columns of D
  – D’s elements have non-unit stride
  – Use normal vld for B and vlds (strided vector load) for D
Indexed Vector Load/Stores

• A.k.a, *gather* (indexed load) and *scatter* (indexed store)

• Consider the following *sparse* vector-vector addition:

```
for (i = 0; i < n; i=i+1)
    A[K[i]] = A[K[i]] + C[M[i]];
```

• Can vectorize the addition operation?
  – Yes, but need a way to vector load/store to random addresses
  – Use indexed vector load/stores

```c
vld v0, x7  # load K[]
vldx v1, x5, v0  # load A[K[]]
vld v2, x28  # load M[]
vldx v3, x6, v2  # load C[M[]]
vadd v1, v1, v3  # add
vstx v1, x5, v0  # store A[K[]]
```
Memory System Design

• DLP workload are very memory intensive
  – Because of large data sets
  – Caches and compiler optimizations can help but not enough

• Supporting strided and indexed vector loads/stores can generate many parallel memory accesses
  – How to support efficiently?

• Banking: spread memory across many banks w/ fine interleaving
  – Can access all banks in parallel if no bank conflict; otherwise will need to stall (structural hazard)

• Example:
  – 32 processors, each generating 4 loads and 2 stores/cycle
  – Processor cycle time is 2.25 ns, Memory cycle time is 15 ns
  – How many memory banks needed?
SIMD ISA Extensions
SIMD Extensions (1)

• SIMD extensions are a smaller version of vector processors
  – Integrated with ordinary scalar processors
  – E.g., MMX, SSE and AVX extensions for x86

• The original idea was to use a functional unit built for a single large operation for many parallel smaller ops
  – E.g., using one 64-bit adder to do eight 8-bit addition by partitioning the carry chain

• Initially, they were not meant to focus on memory-intensive data-parallel applications, but rather digital signal-processing (DSP) applications
  – DSP apps are more compute-bound than memory-bound
  – DSP apps usually use smaller data types

Hiding memory-latency was not originally an issue!
SIMD Extensions (2)

• SIMD extensions were slow to add vector ideas such as vector length, strided and indexed load/stores, predicated execution, etc.

• Things are changing now because of Big Data applications that are memory bound

• E.g., AVX-512 (available in recent Intel processors)
  – Has vectors of 512 bits (8 64-bit elements or 64 8-bit elements)
  – Supports all of the above vector load/stores and other features
SIMD Example: Intel Xeon Phi

- Multi-core chip with Pentium-based SIMD processors
  - Targeting HPC market (Goal: high GFLOPS, GFLOPS/Watt)

- 4 hardware threads + wide SIMD units
  - Vector ISA: 32 vector registers (512b), 8 mask registers, scatter/gather

- In-order, short pipeline
  - Why in-order?
GPUs
Graphics Processing Unit (GPU)

• An architecture for compute-intensive, highly data-parallel computation
  – Exactly what graphics rendering is about
  – Transistors devoted to data processing rather than caching and flow control
Data Parallelism in GPUs

• GPUs take advantage of massive DLP to provide very high FLOP rates
  – More than 1 Tera DP FLOP in NVIDIA GK110

• SIMT execution model
  – Single instruction multiple threads
  – Trying to distinguish itself from both “vectors” and “SIMD”
  – A key difference: better support for conditional control flow

• Program it with CUDA or OpenCL (among other things)
  – Extensions to C
  – Perform a “shader task” (a snippet of scalar computation) over many elements
  – Internally, GPU uses scatter/gather and vector-mask-like operations
CUDA

- Extension of the C language

- Function types
  - *Device code* (kernel): run on the GPU
  - *Host code*: run on the CPU and calls device programs

- Extensions / API
  - Function type: __global__, __device__, __host__
  - Variable type: __shared__, __constant__
    - Affects allocation of variable in different types of memory
  - cudaMalloc(), cudaFree(), cudaMemcpy(),...
  - __syncthreads(), atomicAdd(),...
CUDA Software Model

- A kernel is executed as a **grid of thread blocks**
  - Per-thread register and local-memory space
  - Per-block shared-memory space
  - Shared global memory space

- Blocks are considered **cooperating** arrays of threads
  - Share memory
  - Can synchronize

- Blocks within a grid are independent
  - can execute concurrently
  - No cooperation across blocks
SAXPY in CUDA

```c
__global__ void saxpy(int n, float a, float *x, float *y) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

// Perform SAXPY on with 512 threads/block
int block_cnt = (N + 511) / 512;
saxpy<<<block_cnt,512>>>(N, 2.0, x, y);
```

- Each CUDA thread operates on one data element
  - That’s the reason behind **MT** in **SIMT**

- Hardware tries to execute these threads in lock-step as long as they all execute the same instruction together
  - That’s the **SI** part in **SIMT**

- We’ll see how shortly
Heterogeneous Programming

- Use the right processor for the right job

Serial Code

Parallel Kernel

```c
foo<<< nBlk, nTid >>>(args);
```

Serial Code

Parallel Kernel

```c
bar<<< nBlk, nTid >>>(args);
```
Compiling CUDA

- nvcc
  - Compiler driver
  - Invoke cudacc, g++, cl

- PTX
  - Parallel Thread eXecution

```
ld.global.v4.f32 {f1,f3,f5,f7}, [$r9+0];
mad.f32 f1, f5, f3, f1;
```

Courtesy NVIDIA
CUDA Hardware Model

• Follows the software model closely

• Each thread block executed by a single multiprocessor
  – Synchronized using shared memory

• Many thread blocks assigned to a single multiprocessor
  – Executed concurrently in a FGMT fashion
  – Keep GPU as busy as possible

• Running many threads in parallel can hide DRAM memory latency
  – Global memory access can be several hundred cycles
Example: NVIDIA Kepler GK110

- 15 SMX processors, shared L2, 6 memory controllers
  - 1 TFLOP dual-precision FP
- HW thread scheduling
  - No OS involvement in scheduling

Source: NVIDIA’s Next Generation CUDA Compute Architecture: Kepler GK110
Streaming Multiprocessor (SMX)

• Capabilities
  – 64K registers
  – 192 simple cores
    • Int and SP FPU
  – 64 DP FPUs
  – 32 LD/ST Units (LSU)
  – 32 Special Function Units (FSU)

• Warp Scheduling
  – 4 independent warp schedulers
  – 2 inst dispatch per warp

Source: NVIDIA’s Next Generation CUDA Compute Architecture: Kepler GK110
Latency Hiding with “Thread Warps”

- **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 branch prediction)
  - Interleave warp execution to hide latencies

- Register values of all threads stay in register file
  - No OS context switching

Slide credit: Tor Aamodt
Warp-based SIMT vs. Traditional SIMD

- Traditional SIMD consists of a single thread
  - SIMD Programming model (no threads) \(\rightarrow\) SW needs to know vector length
  - ISA contains vector/SIMD instructions

- Warp-based SIMT consists of multiple scalar threads
  - Same instruction executed by all threads
    - Does not have to be lock step
  - Each thread can be treated individually
    - i.e., placed in a different warp \(\rightarrow\) programming model not SIMD
    - SW does not need to know vector length
    - Enables memory and branch latency tolerance
  - ISA is scalar \(\rightarrow\) vector instructions formed dynamically
Warp Scheduling in Kepler

- 64 warps per SMX
  - 32 threads per warp
  - 64K registers/SMX
  - Up to 255 registers per thread

- Scheduling
  - 4 schedulers select 1 warp per cycle each
  - 2 independent instructions issued per warp
  - Total bandwidth = 4 * 2 * 32 = 256 ops/cycle

- Register Scoreboarding
  - To track ready instructions for long latency ops

- Compiler handles scheduling for fixed-latency operations
  - Binary incompatibility?

Source: NVIDIA’s Next Generation CUDA Compute Architecture: Kepler GK110
What about branching?

```cpp
if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
```

<resume unconditional shader code>
What about branching?

```plaintext
if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
<resume unconditional shader code>
```
What about branching?

Not all ALUs do useful work!
Worst case: 1/8 performance

```cpp
if (x > θ) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = θ;
    refl = Ka;
}
```
What about branching?

```c
if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
```

<unconditional shader code>

<resume unconditional shader code>
Memory Hierarchy

- Each SMX has 64KB of memory
  - Split between shared mem and L1 cache
    - 16/48, 32/32, 48/16
    - 256B per access

- 48KB read-only data cache
  - Compiler controlled

- 1.5MB shared L2

- Support for atomic operations
  - atomicCAS, atomicADD, ...

- Throughput-oriented main memory
  - Memory coalescing
  - *Graphics DDR (GDDR)*
    - Very wide channels: 256 bit vs. 64 bit for DDR
    - Lower clock rate than DDR

Source: NVIDIA’s Next Generation CUDA Compute Architecture: Kepler GK110