Vector Processing

Manycores, SIMDs, and Cells (Oh my!)
Sodani HC 2015

Knights Landing (KNL): 2nd Generation Intel Xeon Phi Processor
Intel 80-core Teraflops Research Chip
“We can now put 80 cores on a chip. We just don’t have any idea of what to do with them.”
Intel MIC (Xeon Phi)

- Many Integrated Cores - after Larrabee GPU & 80 core
- Knights Ferry prototype with 32 cores, 1024 bit ring
- Knights Corner product
- 50 x86 (P54C) cores, 4-thread, superscalar, in-order
- 512 bit SIMD registers
  - 16x32 bit vectors, gather/scatter/mask
- Programmed with OpenMP, OpenCL, Intel Cilk Plus
- Claim up to 1.2 TFLOPS @ 300W
Knight’s Landing Phi

- 72 Silvermont Atom cores in 36 tiles
  - 2-wide OoO issue, 4-way threading, mesh connect
- Two 512-bit vector units
- Each pair shares 1MB cache
  - MESIF, directory-based coherence with other tiles
- Up to 215 Watts
Memory

- 16GB MCDRAM on same carrier (a la Pentium Pro)
- 3-modes: All cache, all RAM, part cache part RAM
- Specially allocated in software for critical data
- Other data in slower DDR
Interconnect

- Mesh of rings with three modes
- All-to-all: slowest but most general
- Quadrant: Directories for groups of 9 tiles (faster)
- Sub-NUMA: Quadrants are separate NUMA domains that allow explicit software optimization for locality (fastest, most programming effort)
- 25GB/s Omni-Path external ports
Performance

- 5X peak DP floating performance of Knights Corner (implies about 6TFLOPS)
- At 200W, gives about 30GFLOPS/W
- Knights Mill variant introduced 2017 with optimizations for machine learning
Discussion
Synergistic Processing in Cell’s Multicore Architecture

Gschwind, et. al.
Cell Broadband Engine Overview

- Heterogeneous Multicore Processor
- One Power-PC-based control processor (PPE)
- Eight “Synergistic” Vector-only Processors (SPE)
- Interconnect Bus (EIB)
- Common address space
As Figure 1 illustrates, the Cell BE implements a single-chip multiprocessor with nine processors operating on a shared, coherent system memory. The function of the processor elements is specialized into two types: the Power processor element (PPE) is optimized for control tasks and the eight synergistic processor elements (SPEs) provide an execution environment optimized for data processing. Figure 2 is a die photo of the Cell BE.

The design goals of the SPE and its architectural specification were to optimize for a low complexity, low area implementation. The PPE is built on IBM's 64-bit Power Architecture with 128-bit vector media extensions and a two-level on-chip cache hierarchy. It is fully compliant with the 64-bit Power Architecture specification and can run 32-bit and 64-bit operating systems and applications.

The SPEs are independent processors, each running an independent application thread. The SPE design is optimized for computation-intensive applications. Each SPE includes a private local store for efficient instruction and data access, but also has full access to the coherent shared memory, including the memory-mapped I/O space. Both types of processor cores share access to a common address space, which includes main memory, and address ranges corresponding to each SPE's local store, control registers, and I/O devices.
PPE Functionality

- Runs Operating System
- General Purpose (scalar heavy) computation
- Organizes structure of global address space
- Coordinates distribution and collection of data
SPE Functionality

- Vector/SIMD Instruction Set
- 128 128-bit Registers
  - Register Contents are polymorphic
- 256 KB Local Store
- 2-way Specialized Pipeline
SPE Programming (I)

- SPE has no native scalar processing
  - No scalar registers
  - Registers can hold vectors of ints and floats
- Instead, SPE Scalar processing is folded into vector processing
  - Manual alignment via software
  - Aim is for compiler to pick appropriate alignment
SPE Programming (II)

- Scalar Layering
  - Sequential scalar operations on a vector machine
  - Large register file can help (more scratch space)

- Data-parallel conditional execution
  - Branches expensive in SPEs
  - Minimal Branch Prediction (Hints)
  - Convert if-then to vector select
Scalar Layering

To illustrate how scalar layering works, consider the operation of SIMD data-parallel execution pipelines as described earlier on a four-element vector consisting of one word each. Figure 4 illustrates how a processor executes a SIMD instruction by performing the same operation—in parallel—on each element. In the example, the SIMD instruction sources two vector registers containing elements $x_0$, $x_1$, $x_2$, and $x_3$ and $y_0$, $y_1$, $y_2$, and $y_3$, respectively, and yields four results: $z_0 = x_0 - y_0; z_1 = x_1 - y_1; z_2 = x_2 - y_2; and z_3 = x_3 - y_3$.
Data-parallel Selection

for (i = 0; i< VL; i++)
    if (a[i]>b[i])
        m[i] = a[i]*2;
    else
        m[i] = b[i]*3;

(a)

Figure 6. The use of data-parallel select to exploit data parallelism. (a) Conditional operations are integrated into SIMD-based computation. (b) Using traditional code generation techniques, the source code is turned into a sequence of test and conditional branch instructions for each vector element. High branch misprediction rates of data-dependent branches and data conversion between vector and scalar representations incur long schedules. (c) Exploiting data-parallel conditional execution with data-parallel select allows the processing of conditional operations concurrently on multiple vector elements. In addition to exploiting data parallelism, data-parallel select purges hard-to-predict data-dependent branches from the instruction mix.
SPE Arithmetic and Local Store

- Emphasis on Non-saturating integer and single-precision FP
- Local Store is NOT a cache
  - Simpler Hardware
  - Deterministic Timing
- YOU have to perform the DMAs yourself (explicit prefetch for next thread)
- Local Store holds code, too!
SPE Pipelines

- Dual-pipelines, statically scheduled
  - Even = Integer and FP operations
  - Odd = Memory, Branch, and Data Formatting
- Explicit branch prefetcher instruction
SPE Pipelines

In the Cell BE, the SPU front end implements statically scheduled instruction fetch to reduce the cost of dynamic instruction scheduling hardware. However, the SPU architecture is not limited to implementations using static scheduling. The SPU architecture is bundle-oriented and supports the delivery of up to two instructions per cycle to the data-parallel back end. All instructions respect their dependencies to ensure sequential program semantics for future architectural compatibility and to ensure good code density.

Exploiting wide accesses for both instruction and data accesses decreases the necessary accesses and improves power efficiency.

### Microarchitecture for the SPE pipeline

<table>
<thead>
<tr>
<th>Type</th>
<th>IF</th>
<th>IB</th>
<th>ID</th>
<th>IS</th>
<th>RF</th>
<th>EX</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch instruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Permute instruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load/store instruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fixed-point instruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating-point instruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### SPU pipeline front end

- IF: Instruction fetch
- IB: Instruction buffer
- ID: Instruction decode
- IS: Instruction issue
- RF: Register file access
- EX: Execution
- WB: Write back
Summary of Techniques

- Turn scalar operations into shuffles
- Turn branches into selects (when possible)
- Plan local store resources carefully
- Balance your pipeline allocations carefully
- The point:
  - Hardware is simpler, but
  - Much more is exposed to the software
Discussion
Single Instruction Multiple Data

Obvious and simple rarely is either
Bit-Serial SIMD: CM-1 (&2), late 1980's
Sort of SIMD: CM-5 (SPMD), early 1990’s
Maspar (late 1980s)

- Grew out of project at Digital Equipment Corp
- 32 4-bit PEs per chip
- 8-way grid interconnect
- Up to 16K processors
Many others of that ilk

- Goodyear Staran, MPP, ASPRO (bit-serial SIMD)
- CMU/Intel Warp and iWarp systolic arrays
- nCUBE (SPMD)
- Inmos Transputer (CSP array)
- UMass/Hughes IUA
  - heterogeneous bit-serial SIMD, 32-bit SPMD, 32-bit CC-NUMA SMP
Why SIMD?

- Conceptually simple form of parallelism
  - Parallel vector operations common in mathematics
  - Also common in image processing, signal processing, database, etc.
- Serial program with parallel data type and operations
- Efficient silicon implementation (many ALUs under a single control unit)
Why not SIMD?

- Size of problem almost never matches hardware
- If smaller, then a fraction of the array is idle
- If larger, then the array has to be virtualized
  - Virtualization has to swap contexts
  - Handle communication across virtual tiles boundaries
  - More fractional arrays at edge of virtual array
Why not SIMD?

- Branches have to serialize — times are additive
- IF (A < B) implies elements that meet the condition take the branch, those that fail take the ELSE clause
- Only one source of instructions for all elements
  - Select A<B, issue instructions
  - Select A≥B, issue instructions
- Multiway branches further divide the elements
Why not SIMD?

- Instruction distribution is hard to scale up
- Assumes a globally synchronous clock
- Broadcast has to be balanced for simultaneous arrival
- Instruction generation takes multiple operations, so has to run faster than consumption, which is simpler
  - Makes clock scaling difficult
- Scaling in size consumes much more power
Why not SIMD?

- Collective operations can be slow
- Need feedback from computations for global branches
- Fan-in from thousands to millions of elements is a multi-stage process (can be pipelined but doesn’t hide latency)
- Array either sits idle during collective, or control is much more complex
Why not SIMD?

- Context switching is expensive
- Elements typically lack enough space to hold multiple contexts
- Entire array has massive amount of data
- Switching moves whole context out, new context in
- Array is idle for long period during context switch
Why not SIMD?

- Can only address some issues by multithreading
  - Hide collective delay, hide issue latency
- Local expansion of instructions
  - Allows asynchronous clocking, communication
  - Needs rate buffers between asynchronous sections
  - More complex hardware, fewer elements
- Optimal use exposes threading in programming model
Discussion