An initial encounter

Sverre Jarp

CERN openlab Minor Review Meeting – 16 June 2009
Disclaimer

- This is a small digest of what is publicly available on the Web

In particular:
- “A First Look at the LRBni” by Michael Abrash, Dr. Dobb’s Journal (Apr 01, 2009)

But,
- The presentation is principally limited to the instruction set
Today’s agenda

- What is Larrabee, anyway?
- Architecture
  - Chip, Processor, Vector
- Instruction overview
  - Vector load, store
  - Vector masks
  - Vector arithmetic, logical, shift
  - Others
- A simple code snippet
What is Larrabee?

- It is an architecture
  - Actually, it is a “RISC-like” architectural extension of x86

- Implementation:
  - Per core:
    - Multiple threads (currently 4)
    - Vector unit with lots of new instructions
    - In-order execution
  - Many cores
    - Interconnected via a coherent bus (or “ring”)

- “The architecture will first be used in GPUs, and could be used in CPUs as well”
Sverre’s 7 dimensions

- Several are rather different in Larrabee
  - Compared to Xeon:
    - Superscalar (-)
    - SIMD width (++)
    - Multithreading (+)
    - Multicore (++)
Schematic chip overview

- Conceptually, it looks like this:
Scalar and vector units are separate

- Communication via registers (and flags)

- **L1-cache**
  - 32 KB each

- **L2-cache**
  - 256 KB each subset
Vector-related registers

- 32 vector registers with 512 bits
  - v0 – v31
- Each can hold:
  - 16 floats, 16 int32s, 8 doubles, 8 int64s, etc.

- 8 mask registers with 16 bits
  - k0 – k7
Instruction classes

Six major groupings:

1) Vector arithmetic, logical and shift
2) Vector compare
3) Vector mask
4) Vector load/store
5) Misc. vector
6) Misc. scalar

Convention for mnemonics:

- \texttt{vxxxpt} (vector instruction packed type)
- \texttt{kxxx} (mask instruction)
Standard instruction format

- Typically “vop v1{k1}, v2, (v3/src) “
  - Ternary (3 sources)
  - Target same as first source
  - Third source (but only this one):
    - Also addresses memory, as in: [rbx + rcx*4]
  - Mask register:
    - Predicates updates of target
    - Maintains “state”
## Data element types

**Quick overview:**

<table>
<thead>
<tr>
<th>Size</th>
<th>Ld/St</th>
<th>Signed</th>
<th>Unsigned</th>
<th>FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>dword</td>
<td>int32</td>
<td>uint32</td>
<td>single</td>
</tr>
<tr>
<td>64</td>
<td>qword</td>
<td></td>
<td></td>
<td>double</td>
</tr>
</tbody>
</table>

So, an instruction may operate on **all** or only on some types:

- `vminp{disu}`
- `vsllpi`
Arithmetic, Logical and Shift

- Summarized in the backup section
- Normal collection of
  - Logical (and, or, xor, not, etc.)
  - Shift (Shift left logical, shift right arithmetic, etc.)
  - Lots of convert variants
  - Add, subtract, multiply
  - Min, max
  - Scale, round, etc.
- Some more exotic ones

Sverre Jarp - LRBni - June 09
Vsllpi (shift i32 vector left logical)

- **vsllpi**  v1, v2, v3
  - Shift each i32 element in v2 left logical according to values in v3.
  - Store in v1

v2:

\[
\begin{array}{cccccccccccc}
5 & 14 & 13 & 12 & 11 & 10 & 5 & 6 & 2 & 5 & 7 & 9 & 14 & 4 & 4 & 4 \\
\end{array}
\]

\[<<\]

v3:

\[
\begin{array}{cccccccccccc}
2 & 1 & 0 & 3 & 0 & 0 & 3 & 2 & 1 & 0 & 3 & 1 & 0 & 2 & 1 & 0 \\
\end{array}
\]

= 

v1:

\[
\begin{array}{cccccccccccc}
20 & 28 & 13 & 96 & 11 & 10 & 40 & 24 & 4 & 5 & 56 & 18 & 14 & 16 & 8 & 4 \\
\end{array}
\]
Standard Math Instructions

- 24 fused multiply-add/subtract instructions of the format:
  - “vm fff nnn p t”
    - fff (function): add, addn (negate), sub, subr (reverse)
    - nnn (sequence of operands): 132, 213, 231
    - t: {ds}

- For instance:
  - vmadd231ps
Multiply Add (231ps)

- \texttt{vmadd231ps} \quad v1, v2, v3
  - Multiply f32 v2 and f32 v3, add the result to f32 v1.

\begin{align*}
\text{v2:} & \quad 2.0 \quad 1.0 \quad 3.0 \quad 4.0 \quad 8.0 \quad 9.0 \quad 4.0 \quad 1.0 \quad 1.0 \quad 5.0 \quad 6.0 \quad 2.5 \quad 3.5 \quad 2.0 \quad 0.0 \quad 0.0 \\
\text{v3:} & \quad 2.5 \quad 2.6 \quad 3.0 \quad E12 \quad E11 \quad E10 \quad E9 \quad E8 \quad E7 \quad E6 \quad E5 \quad E4 \quad E3 \quad E2 \quad E1 \quad E0 \\
\text{v1:} & \quad 4.1 \quad 2.4 \quad 4.0 \quad E12 \quad E11 \quad E10 \quad E9 \quad E8 \quad E7 \quad E6 \quad E5 \quad E4 \quad E3 \quad E2 \quad E1 \quad E0 \\
\text{v1:} & \quad 9.1 \quad 5.0 \quad 13.0 \quad E12 \quad E11 \quad E10 \quad E9 \quad E8 \quad E7 \quad E6 \quad E5 \quad E4 \quad E3 \quad E2 \quad E1 \quad E0
\end{align*}
Fused multiply-Add

- Combines one multiplication and one addition
  - Without any loss of accuracy

- Good for cases like:
  - $X^3A + X^2B + X^2C + D = X(X(XA + B) + C) + D$
Vector compare instructions

- Syntax:
  - “vcmpp{disu} k2{k1}, v1, v2, cc”
  - cc:
    - eq, neq
    - lt, nlt
    - le, nle
    - ord, unord

- Results are always stored in a mask register
Mask instructions

- **kand**
  - Bitwise logical-and
- **kandn**
  - Bitwise logical-and-not
- **kandnr**
  - And-not reverse
- **knot**
  - Bitwise logical-not
- **kor**
  - Bitwise logical-or
- **kxnor**
  - Bitwise logical-xnor
- **kxor**
  - Bitwise logical-xor

- **kortest**
  - Set ZF if OR results in all ‘0’, CF if all ‘1’

- **kmov**
  - Move vector masks
- **kmovlhb**
  - Move low to high byte
- **kswapb**
  - Swap and merge high byte portion
Vector Load/Store instructions

- **vgatherd**
  - Gather vector (32-bit elements)

- **vgatherpfd**
  - Prefetch vector (in gather form)

- **vload{dq}**
  - Load vector

- **vexpand{dq}**
  - Load unaligned and expand to vector

- **vcompress{dq}**
  - Compress and store unaligned from vector

- **vscatterd**
  - Scatter vector

- **vscatterpfd**
  - Prefetch vector (in scatter from)

- **vstore{dq}**
  - Store vector

← “d” for “dword”, not “double”
Two common data scenarios

- Work with SOAs
  - Structures of Arrays or simply Arrays

- Work with AOSs
  - Arrays of Structures
Structures Of Arrays

Typical sequence:
- Load all data
  - vloadd or vexpanddd
- Work
- Perform tests
- Mask out irrelevant elements
- More work
- Store modified elements
  - vstored or vcompressd
- Masking (predication) ensures algorithmic optimization, possibly also vexpand

Sverre Jarp – LRBni – June 09
Arrays of Structures

- Typical sequence:
  - Gather all data (vgatherd)
  - Work
  - Perform tests
  - Mask out irrelevant elements
  - More work
  - Store modified elements (vscatterd)

- Gather and masking (predication) should ensure even better algorithmic optimization
From Dr. Dobbs:

- \(v0\) accumulates the results
- \(v2\) keeps addressing offsets

```plaintext
vxorpi v0, v0, v0
;
ChecksumLoop:
  vgatherd v1{k0}, [rbx + v2]
vaddpi v0, v0, v1
vaddpi v2,v2, [Mem_Structure_Sizes]
dec rcx
jnz ChecksumLoop
```
Simple example (Navigation)

Physics example:
- Is the particle inside a box or not?

```c
if (abs(point[0] - origin[0]) > xhalfsz) return FALSE;
if (abs(point[1] - origin[1]) > yhalfsz) return FALSE;
if (abs(point[2] - origin[2]) > zhalfsz) return FALSE;
return true;
```

- Can now be handled inside one vector register
  - This will also be true for AVX
  - Next year’s extension to Xeon
Andrzej’s usage comparisons

- Do you want (computing) power, flexibility, or ease of use?

**ASM**

- Power
- Flexibility
- Ease of use

**Intrinsics**

- Power
- Flexibility
- Ease of use

**Autovectorization**

- Power
- Flexibility
- Ease of use

**High level**

- Power
- Flexibility
- Ease of use

Sverre Jarp - LRBni - June 09
Conclusions

- Larrabee exploits Moore’s law in several dimensions:
  - Long vectors
    - Coupled with sophisticated instruction set
  - Four threads
  - Large (double-digit) core count

- Consequently, applications need to expose:
  - Data parallelism
  - Task parallelism
Further reading

On the Web:

- “A First Look at the LRBni” by Michael Abrash, Dr. Dobb’s Journal (Apr 01, 2009)
  - http://www.ddj.com/architect/216402188
  - http://software.intel.com/file/18198/
- Collection of articles from Intel, including:
  - “Game Physics Performance on the Larrabee Architecture”, by A.Bader et al.
  - “Rasterization on Larrabee” by Michael Abrash

Sverre Jarp - LRBni - June 09
BACKUP
Arithmetic, Logical and Shift (1)

- **vadcpi**
  - Add vectors with carry (in and out)
- **vaddnp{ds}**
  - Add and negate vectors
- **vaddp{dis}**
  - Add vectors
- **vaddsetcpi**
  - Add vectors and set mask to carry
- **vaddsetsp{is}**
  - Add vectors and set mask to sign
- **vandnp{iq}**
  - Bitwise logical-and-not vectors
- **vandp{iq}**
  - Bitwise logical-and vectors
- **vclampzp{is}**
  - Clamp vector between value and zero
- **vcvtpd2p{isu}**
  - Convert vector of double
- **vcvtpi2p{ds}**
  - Convert vector of int32
- **vcvtps2p{diu}**
  - Convert vector of double
- **vcvtps2srgb8**
  - Convert single to sRGB8
- **vcvtpu2p{ds}**
  - Convert vector of uint32
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vmaxabsps</code></td>
<td>Absolute maximum of singles</td>
</tr>
<tr>
<td><code>vmaxp{disu}</code></td>
<td>Maximum</td>
</tr>
<tr>
<td><code>vminp{disu}</code></td>
<td>Minimum</td>
</tr>
<tr>
<td><code>vmulhp{iu}</code></td>
<td>Multiply and store high</td>
</tr>
<tr>
<td><code>vmulpi</code></td>
<td>Multiply and store low</td>
</tr>
<tr>
<td><code>vmulp{ds}</code></td>
<td>Multiply</td>
</tr>
<tr>
<td><code>vorp{iq}</code></td>
<td>Bitwise logical-or</td>
</tr>
<tr>
<td><code>vroundps</code></td>
<td>Round vector</td>
</tr>
<tr>
<td><code>vsbbpi</code></td>
<td>Subtract with borrow (in and out)</td>
</tr>
<tr>
<td><code>vsbbri</code></td>
<td>Reverse subtract with borrow (in and out)</td>
</tr>
<tr>
<td><code>vscaleps</code></td>
<td>Scale vector</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------------------------------------</td>
</tr>
<tr>
<td><code>vsl</code>pi</td>
<td>Shift left logical</td>
</tr>
<tr>
<td><code>vsra</code>pi</td>
<td>Shift right arithmetic</td>
</tr>
<tr>
<td><code>vsrl</code>pi</td>
<td>Shift right logical</td>
</tr>
<tr>
<td><code>vsub</code>p{dis}</td>
<td>Subtract</td>
</tr>
<tr>
<td><code>vsubr</code>p{dis}</td>
<td>Reverse subtract</td>
</tr>
<tr>
<td><code>vsubrsetb</code>pi</td>
<td>Reverse subtract and set borrow</td>
</tr>
<tr>
<td><code>vsubsetb</code>pi</td>
<td>Subtract and set borrow</td>
</tr>
<tr>
<td><code>vxor</code>p{iq}</td>
<td>Bitwise logical-xor</td>
</tr>
</tbody>
</table>
Special madd/msub cases

- **vmadd231p{d}s**
  - Variant with int32

- **vmadd233p{is}**
  - $v1 = (v2 \times \text{ExtractScaleElement}(v3)) + \text{ExtractOffsetElement}(v3)$

- **vmsubr23c1p{ds}**
  - $v1 = 1.0 - (v2 \times v3)$