SIMD Programming with Larrabee

Tom Forsyth

Larrabee Architect

VCG Visual Computing Group

What lies ahead

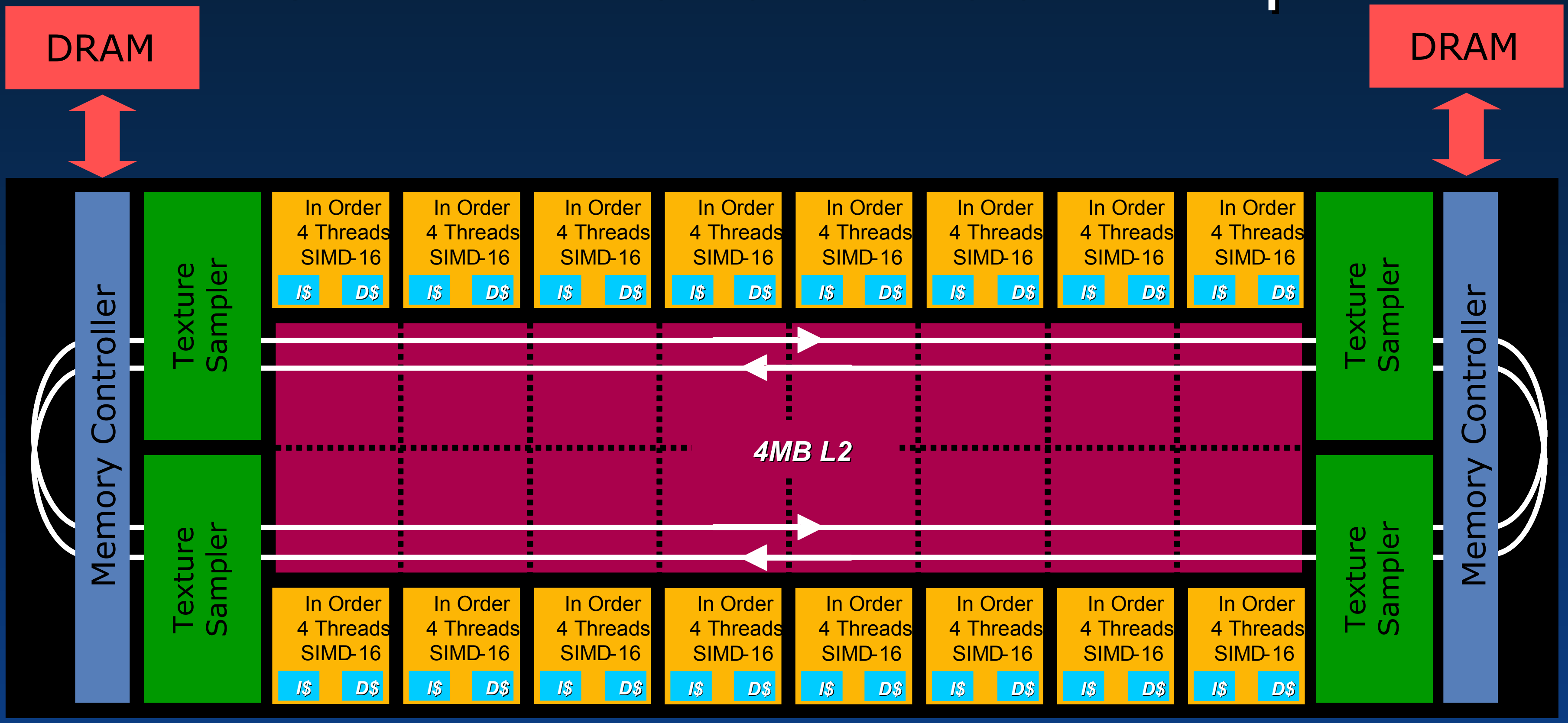
The Larrabee architecture

Larrabee New Instructions

Writing efficient code for Larrabee

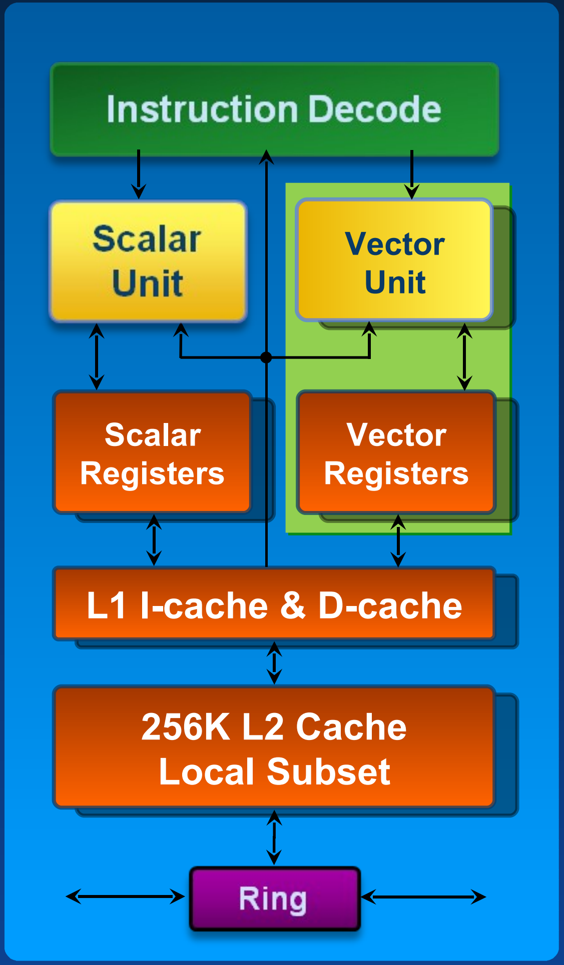
The rendering pipeline

Overview of a Larrabee chip



- CONCEPTUAL MODEL ONLY! Actual numbers of cores, texture units, memory controllers, etc will vary – a lot. Also, structure of ring & placement of devices on ring is more complex than shown

One Larrabee core



Larrabee based on x86 ISA

- All of the left "scalar" half
- Four threads per core
- No surprises, except that there's LOTS of cores and threads

New right-hand vector unit

- Larrabee New Instructions
- 512-bit SIMD vector unit
- 32 vector registers
- Pipelined one-per-clock throughput
- Dual issue with scalar instructions

Larrabee "old" Instructions

The x86 you already know

- Core originally based on Pentium 1
- Upgraded to 64-bit
- Full cache coherency preserved
- x86 memory ordering preserved
- Predictable in-order pipeline model

4 threads per core

- Fully independent "hyperthreads" – no shared state
- Typically run closely ganged to improve cache usage
- Help to hide instruction & L1$-miss latency

No surprises – "just works"

- "microOS" with pthreads, IO, pre-emptive multitasking, etc
- Compile and run any existing code in any language

Larrabee New Instructions

512-bit SIMD

- int32, float32, float64 ALU support
- Today’s talk focussed on the 16-wide float32 operations

Ternary, multiply-add

- Ternary = non-destructive ops = fewer register copies
- Multiply-add = more flops in fewer ops

Load-op

- Third operand can be taken direct from memory at no cost
- Reduces register pressure and latency

Larrabee New Instructions

Broadcast/swizzle

- Scalar->SIMD data broadcasts (e.g. constants, scales)
- Crossing of SIMD lanes (e.g. derivatives, horizontal ops)

Format conversion

- Small formats allow efficient use of caches & bandwidth
- Free common integer formats int8, int16
- Free common graphics formats float16, unorm8
- Built-in support for other graphics formats (e.g. 11:11:10)

Predication and gather/scatter

- Makes for a "complete" vector ISA
- A lot more on these in a bit

Larrabee New Instructions

Designed for software

- Not always the simplest hardware
- Compiler & code scheduler written during the design
- Anything the compiler couldn’t grok got fixed or killed

Very few special cases

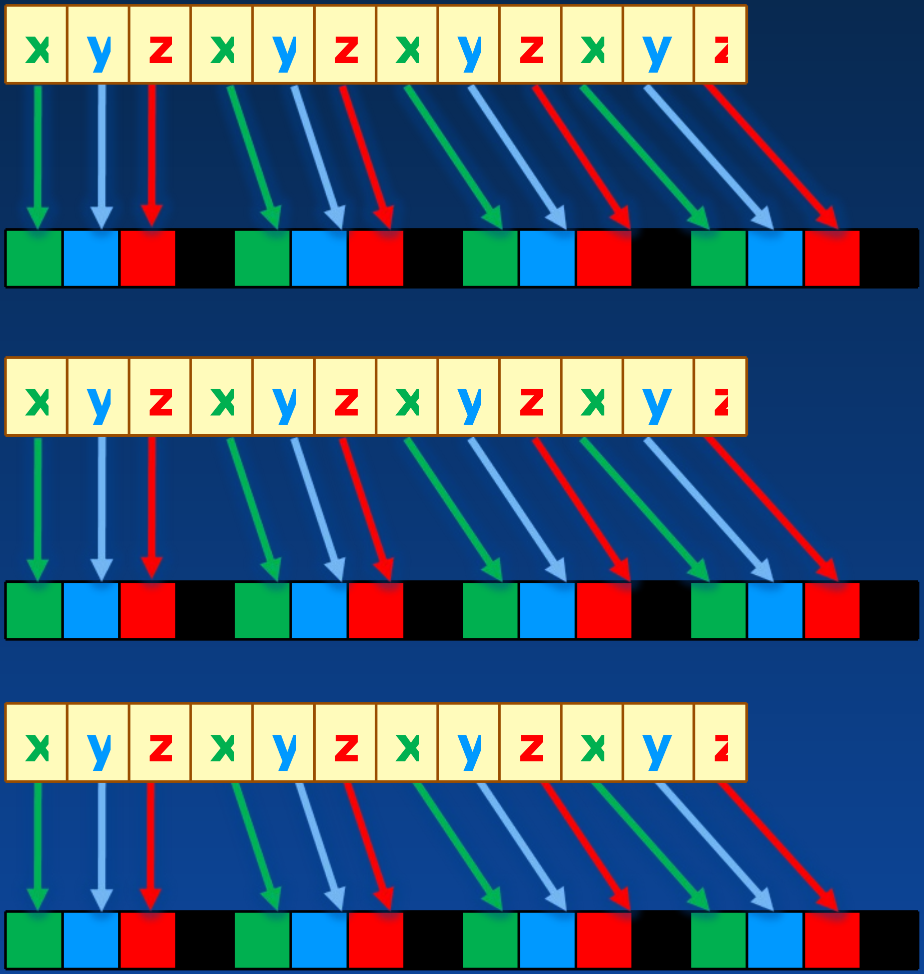
- Compilers don’t cope well with special cases
- e.g. no hard-wiring of register sources
- Most features work the same in all instructions

Targeted at graphics

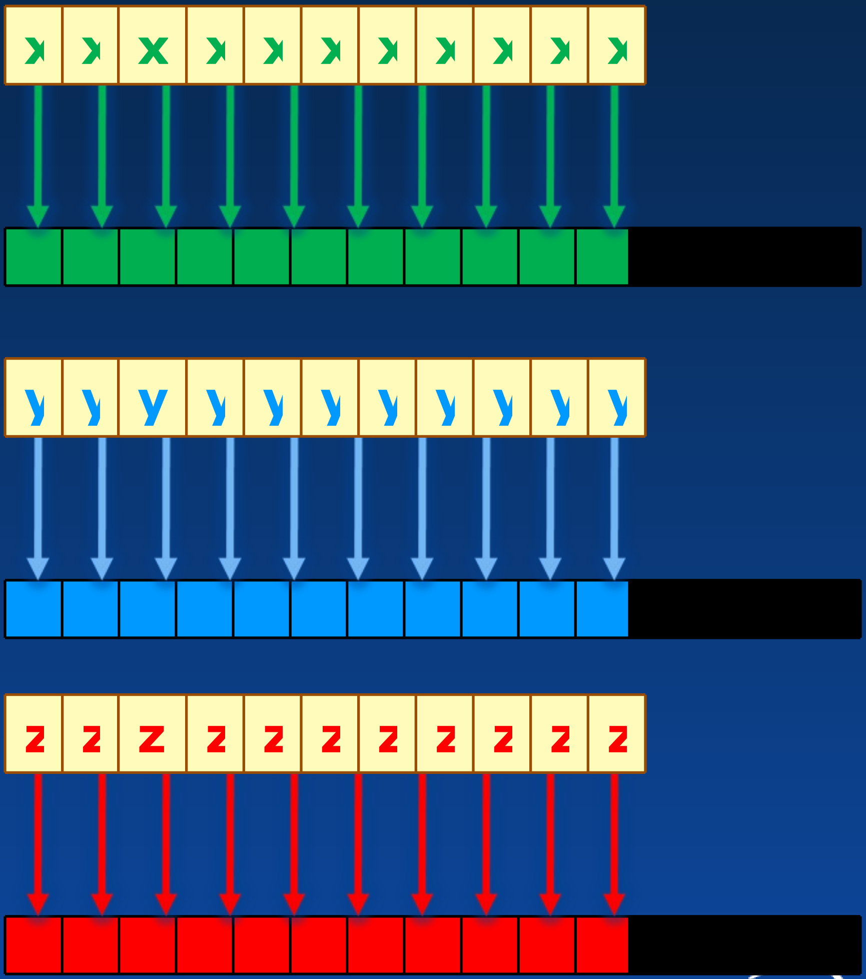
- Surprisingly, ended up with <10% graphics-specific stuff
- DX/OGL format support
- Rasterizer-specific instructions

16 wide SIMD – SOA vs AOS

Array of Structures

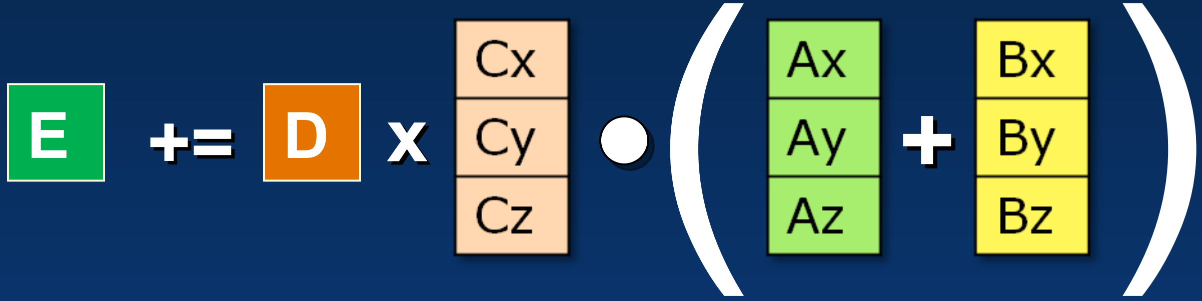


Structure of Arrays



Simple SOA example

```
e += d * dot(c.xyz, a.xyz + b.xyz);
```



E += D x
Cx
Cy
Cz
●
(
Ax
Ay
Az
+
Bx
By
Bz
)

First step is to "scalarize" the code

- Turn vector notation into scalars
- Remember that each "scalar" op is doing 16 things at once

Simple SOA example

```
e += d * dot(c.xyz, a.xyz + b.xyz);
```

```
// temp = a.xyz + b.xyz;
vec3 temp;
temp.x = a.x + b.x;
temp.y = a.y + b.y;
temp.z = a.z + b.z;
```

A vec3 add turns into 3 scalar adds

Simple SOA example

```
e += d * dot(c.xyz, a.xyz + b.xyz);
```

```
// temp = a.xyz + b.xyz;
vec3 temp;
temp.x = a.x + b.x;
temp.y = a.y + b.y;
temp.z = a.z + b.z;
```

```
// t = dot(c.xyz, temp.xyz);
float t = temp.x * c.x;
t      += temp.y * c.y;
t      += temp.z * c.z;
```

Note how the dot-product, which is complex in AOS code and requires horizontal adds or lane-shuffling, becomes easy in SOA code.

Simple SOA example

```
e += d * dot(c.xyz, a.xyz + b.xyz);
```

```
// temp = a.xyz + b.xyz;
vec3 temp;
temp.x = a.x + b.x;
temp.y = a.y + b.y;
temp.z = a.z + b.z;
// t = dot(c.xyz, temp.xyz);
float t = temp.x * c.x;
t      += temp.y * c.y;
t      += temp.z * c.z;
e += d * t;
```

Scalar operations stay scalar with no loss of efficiency in SOA

Now turn into LRBNI instructions

```
e += d * dot(c.xyz, a.xyz + b.xyz);
```

```
// temp = a.xyz + b.xyz;
vec3 temp;
temp.x = a.x + b.x;
temp.y = a.y + b.y;
temp.z = a.z + b.z;
// t = dot(c.xyz, temp.xyz);
float t = temp.x * c.x;
t      += temp.y * c.y;
t      += temp.z * c.z;
e += d * t;
```

Now turn into LRBNI instructions

```
e += d * dot(c.xyz, a.xyz + b.xyz);
```

```
// temp = a.xyz + b.xyz;
vec3 temp;
temp.x = a.x + b.x;
temp.y = a.y + b.y;
temp.z = a.z + b.z;
// t = dot(c.xyz, temp.xyz);
float t = temp.x * c.x;
t      += temp.y * c.y;
t      += temp.z * c.z;
e += d * t;
```

```
vaddps v20, v0, v3
vaddps v21, v1, v4
vaddps v22, v2, v5

vmulps  v23, v20, v6
vmaddps v23, v21, v7
vmaddps v23, v22, v8
vmaddps v10, v9, v23
```

… use names instead of numbers

```
e += d * dot(c.xyz, a.xyz + b.xyz);
```

```
// temp = a.xyz + b.xyz;
vec3 temp;
temp.x = a.x + b.x;
temp.y = a.y + b.y;
temp.z = a.z + b.z;
// t = dot(c.xyz, temp.xyz);
float t = temp.x * c.x;
t      += temp.y * c.y;
t      += temp.z * c.z;
e += d * t;
```

```
vaddps vTempx, vAx, vBx
vaddps vTempy, vAy, vBy
vaddps vTempz, vAz, vBz

vmulps  vT, vTempx, vCx
vmaddps vT, vTempy, vCy
vmaddps vT, vTempz, vCz
vmaddps vE, vD, vT
```

Predication

- 8 16-bit registers k0-k7
- Every instruction can take a mask
  - k0 has limited use – encoding often means “no mask”
- Act as write masks – bit=0 preserves dest

```
    vaddps v1{k6}, v2, v3
```

- Bits in k6 enable/disable writes to v1
  - Preserves existing register contents in bit=0 lanes
  - Usually also disables individual ALU lanes to save power
- Memory stores also take a write mask
  - Preserves existing values in memory

Predication

Predication allows per-lane conditional flow

Vector compare does 16 parallel compares

- Writes results into a write mask
- Mask can be used to protect some of the 16 elements from being changed by instructions

Simple predication example:

```
;if (v5<v6) {v1 += v3;}
vcmppi_lt k7, v5, v6
vaddpi v1{k7}, v1, v3
```

Predication

```
;if (v5<v6) {v1 += v3;}

v5 = 0 4 7 8 3 9 2 0 6 3 8 9 4 5 0 1
v6 = 9 4 8 2 0 9 4 5 5 3 4 6 9 1 3 0
vcmppi_lt k7, v5, v6
```

Predication

```
;if (v5<v6) {v1 += v3;}

v5 = 0 4 7 8 3 9 2 0 6 3 8 9 4 5 0 1
v6 = 9 4 8 2 0 9 4 5 5 3 4 6 9 1 3 0
vcmppi_lt k7, v5, v6
k7 = 1 0 1 0 0 0 1 1 0 0 0 0 1 0 1 0
```

Predication

```
;if (v5<v6) {v1 += v3;}

v5 = 0 4 7 8 3 9 2 0 6 3 8 9 4 5 0 1
v6 = 9 4 8 2 0 9 4 5 5 3 4 6 9 1 3 0
vcmppi_lt k7, v5, v6
k7 = 1 0 1 0 0 0 1 1 0 0 0 0 1 0 1 0

v3 = 5 6 7 8 5 6 7 8 5 6 7 8 5 6 7 8
v1 = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
vaddpi v1{k7}, v1, v3
```

Predication

```
;if (v5<v6) {v1 += v3;}

v5 = 0 4 7 8 3 9 2 0 6 3 8 9 4 5 0 1
v6 = 9 4 8 2 0 9 4 5 5 3 4 6 9 1 3 0
vcmppi_lt k7, v5, v6
k7 = 1 0 1 0 0 0 1 1 0 0 0 0 1 0 1 0

v3 = 5 6 7 8 5 6 7 8 5 6 7 8 5 6 7 8
v1 = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
vaddpi v1{k7}, v1, v3
v1 = 6 1 8 1 1 1 8 9 1 1 1 1 6 1 8 1
```

Existing values are preserved in disabled lanes

Predication - functional

```
;  {e+=d*dot(c.xyz,a.xyz+b.xyz);}



vaddps vTempx, vAx, vBx
vaddps vTempy, vAy, vBy
vaddps vTempz, vAz, vBz
vmulps  vT, vTempx, vCx
vmaddps vT, vTempy, vCy
vmaddps vT, vTempz, vCz

vmaddps vE,     vD, vT
```

…same dot-product SOA example code as before…

Predication - functional

```
;if (d > 0)
;   {e+=d*dot(c.xyz,a.xyz+b.xyz);}




vaddps vTempx, vAx, vBx
vaddps vTempy, vAy, vBy
vaddps vTempz, vAz, vBz
vmulps  vT, vTempx, vCx
vmaddps vT, vTempy, vCy
vmaddps vT, vTempz, vCz
vcmpps_gt kT, vD, [ConstZero]{1to16}
vmaddps vE{kT}, vD, vT
```

Add a conditional clause

Use load-op and broadcast to do a vector compare against a constant zero in memory.

Predicate the multiply-add.

Predication – early-out branches

```
;if (d > 0)
;  {e+=d*dot(c.xyz,a.xyz+b.xyz)}
vcmpps_gt kT, vD, [ConstZero]{1to16}
kortest kT, kT
jz skip_all_this
vaddps vTempx, vAx, vBx
vaddps vTempy, vAy, vBy
vaddps vTempz, vAz, vBz
vmulps  vT, vTempx, vCx
vmaddps vT, vTempy, vCy
vmaddps vT, vTempz, vCz

vmaddps vE{kT}, vD, vT
skip_all_this:
```

Move the compare earlier…

Sets the Z flag if kT is all-0

Early-out branch

All this code is completely skipped if all 16 values of d are <=0.

If only some lanes are zero, we run the code, but we still get the correct answers because of predication.

Predication – power-efficient

```
;if (d > 0)
;  {e+=d*dot(c.xyz,a.xyz+b.xyz)}
vcmpps_gt kT, vD, [ConstZero]{1to16}
kortest kT, kT
jz skip_all_this
vaddps vTempx{kT}, vAx, vBx
vaddps vTempy{kT}, vAy, vBy
vaddps vTempz{kT}, vAz, vBz
vmulps  vT{kT}, vTempx, vCx
vmaddps vT{kT}, vTempy, vCy
vmaddps vT{kT}, vTempz, vCz


vmaddps vE{kT}, vD, vT
skip_all_this:
```

…and we now add predication to all these instructions, not just the final one, which saves power by not computing results for lanes you won't use.

Predication - loops

- There is still a standard x86 loop
  - A label and a conditional jump
- A mask stores which lanes are still running
  - The mask predicates all operations inside the loop
  - Predicated lanes remain unchanged
- Do the end-of-loop condition once per lane
  - If a lane hits its end-of-loop, it clears the mask bit
  - That lane is now stopped – running more loops does not affect its results
- When all lanes have finished, stop the loop
  - Keep looping until the mask register is all-0

Predication - loops

- Vector compare can take a starting mask
  - Bits that are already zero will stay zero

```
; y=1; while(x>0){ y+=y; x--; };
```

Predication - loops

Vector compare can take a starting mask

- Bits that are already zero will stay zero

```
; y=1; while(x>0){ y+=y; x--; };

vloadpi vY, [ConstOne]{1to16}                y=1;
loop:

vaddpi vY     , vY, vY                       y = y + y;
vsubpi vX     , vX, [ConstOne]{1to16}        x = x - 1;

j?? loop
```

Predication - loops

Vector compare can take a starting mask

Bits that are already zero will stay zero

```
; y=1; while(x>0){ y+=y; x--; };
kxnor kL, kL
vloadpi vY, [ConstOne]{1to16}
loop:
vcmppi_gt kL{kL}, vX, [ConstZero]{1to16}
vaddpi vY{kL}, vY, vY
vsubpi vX{kL}, vX, [ConstOne]{1to16}
kortest kL, kL
jnz loop
```

Sets the loop mask to all-1s

x>0?

while(any x>0)

Predication – iteration 1

kL =	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
vY =	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
vX =	3	0	1	2	5	4	2	1	0	2	3	1	3	5	2	4

```
loop:
vcmppi_gt kL{kL}, vX, [ConstZero]{1to16}
vaddpi vY{kL}, vY, vY
vsubpi vX{kL}, vX, [ConstOne]{1to16}
kortest kL, kL
jnz loop
```

kL =	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1

Predication – iteration 1

kL =	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
vY =	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
vX =	3	0	1	2	5	4	2	1	0	2	3	1	3	5	2	4

```
loop:
vcmppi_gt kL{kL}, vX, [ConstZero]{1to16}
vaddpi vY{kL}, vY, vY
vsubpi vX{kL}, vX, [ConstOne]{1to16}
kortest kL, kL
jnz loop
```

kL =	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1
vY =	2	1	2	2	2	2	2	2	1	2	2	2	2	2	2	2

Predication – iteration 1

kL =	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
vY =	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
vX =	3	0	1	2	5	4	2	1	0	2	3	1	3	5	2	4

```
loop:
vcmppi_gt kL{kL}, vX, [ConstZero]{1to16}
vaddpi vY{kL}, vY, vY
vsubpi vX{kL}, vX, [ConstOne]{1to16}
kortest kL, kL
jnz loop
```

kL =	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1
vY =	2	1	2	2	2	2	2	2	1	2	2	2	2	2	2	2
vX =	2	0	0	1	4	3	1	0	0	1	2	0	2	4	1	3

Predication – iteration 1

kL =	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
vY =	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
vX =	3	0	1	2	5	4	2	1	0	2	3	1	3	5	2	4

```
loop:
vcmppi_gt kL{kL}, vX, [ConstZero]{1to16}
vaddpi vY{kL}, vY, vY
vsubpi vX{kL}, vX, [ConstOne]{1to16}
kortest kL, kL
jnz loop
```

kL =	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1
vY =	2	1	2	2	2	2	2	2	1	2	2	2	2	2	2	2
vX =	2	0	0	1	4	3	1	0	0	1	2	0	2	4	1	3

kL is not all-0, so we continue the loop

Predication – iteration 2

kL =	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1
vY =	2	1	2	2	2	2	2	2	1	2	2	2	2	2	2	2
vX =	2	0	0	1	4	3	1	0	0	1	2	0	2	4	1	3

```
loop:
vcmppi_gt kL{kL}, vX, [ConstZero]{1to16}
vaddpi vY{kL}, vY, vY
vsubpi vX{kL}, vX, [ConstOne]{1to16}
kortest kL, kL
jnz loop
```

kL =	1	0	0	1	1	1	1	0	0	1	1	0	1	1	1	1
vY =	4	1	2	4	4	4	4	2	1	4	4	2	4	4	4	4
vX =	1	0	0	0	3	2	0	0	0	0	1	0	1	3	0	2

Only do the compare on unmasked lanes

Just like every other math instruction

Predication – iteration 3

kL =	1	0	0	1	1	1	1	0	0	1	1	0	1	1	1	1
vY =	4	1	2	4	4	4	4	2	1	4	4	2	4	4	4	4
vX =	1	0	0	0	3	2	0	0	0	0	1	0	1	3	0	2

```
loop:
vcmppi_gt kL{kL}, vX, [ConstZero]{1to16}
vaddpi vY{kL}, vY, vY
vsubpi vX{kL}, vX, [ConstOne]{1to16}
kortest kL, kL
jnz loop
```

kL =	1	0	0	0	1	1	0	0	0	0	1	0	1	1	0	1
vY =	8	1	2	4	8	8	4	2	1	4	8	2	8	8	4	8
vX =	0	0	0	0	2	1	0	0	0	0	0	0	0	2	0	1

Predication – iteration 4

kL =	1	0	0	0	1	1	0	0	0	0	1	0	1	1	0	1
vY =	8	1	2	4	8	8	4	2	1	4	8	2	8	8	4	8
vX =	0	0	0	0	2	1	0	0	0	0	0	0	0	2	0	1

```
loop:
vcmppi_gt kL{kL}, vX, [ConstZero]{1to16}
vaddpi vY{kL}, vY, vY
vsubpi vX{kL}, vX, [ConstOne]{1to16}
kortest kL, kL
jnz loop
```

kL =	0	0	0	0	1	1	0	0	0	0	0	0	0	1	0	1
vY =	8	1	2	4	16	16	4	2	1	4	8	2	8	16	4	16
vX =	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0

Predication – iteration 5

kL =	0	0	0	0	1	1	0	0	0	0	0	0	0	1	0	1
vY =	8	1	2	4	16	16	4	2	1	4	8	2	8	16	4	16
vX =	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0

```
loop:
vcmppi_gt kL{kL}, vX, [ConstZero]{1to16}
vaddpi vY{kL}, vY, vY
vsubpi vX{kL}, vX, [ConstOne]{1to16}
kortest kL, kL
jnz loop
```

kL =	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0
vY =	8	1	2	4	32	16	4	2	1	4	8	2	8	32	4	16
vX =	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Predication – iteration 6

kL =	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0
vY =	8	1	2	4	32	16	4	2	1	4	8	2	8	32	4	16
vX =	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

```
loop:
vcmppi_gt kL{kL}, vX, [ConstZero]{1to16}
vaddpi vY{kL}, vY, vY
vsubpi vX{kL}, vX, [ConstOne]{1to16}
kortest kL, kL
jnz loop
```

kL =	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
vY =	8	1	2	4	32	16	4	2	1	4	8	2	8	32	4	16
vX =	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

kL is now all-0, so we exit the loop

Predication – loops

```
    kxnor kL, kL
    vorpi vR, vRstart, vRstart
    vorpi vI, vIstart, vIstart
    vxorpi vIter, vIter, vIter
loop:
    vmulps vTemp{kL}, vR, vI
    vaddps vTemp{kL}, vTemp, vTemp
    vmadd213ps vR{kL}, vR, vRstart
    vmsub231ps vR{kL}, vR, vI
    vaddps vI{kL}, vTemp, vIstart
    vaddps vIter{kL}, vIter, [ConstOne]{1to16}
    vmulps  vTemp{kL}, vR, vR
    vmaddps vTemp{kL}, vI, vI
    vcmpps_le kL{kL}, vTemp, [ConstOne]{1to16}
    kortest kL, kL
    jnz loop
    ; Result iteration count in vIter
```

A Mandelbrot set generator. Again, notice the comparison that tests if the point is outside the unit circle. Once all 16 points are outside, the loop ends.

Gather/scatter

- Important part of a wide vector ISA
- SOA mode is difficult to get data into
  - Most data structures are AOS
  - Natural format for indirections – pointer to each structure
- Gather/scatter allows sparse read/write
  - Gather gets data into the 16-wide SOA format in registers
  - Process data 16-wide
  - Scatter stores data back out into AOS
- Temporaries stay as SOA
  - Gets the benefit of load-op and co-issue stores

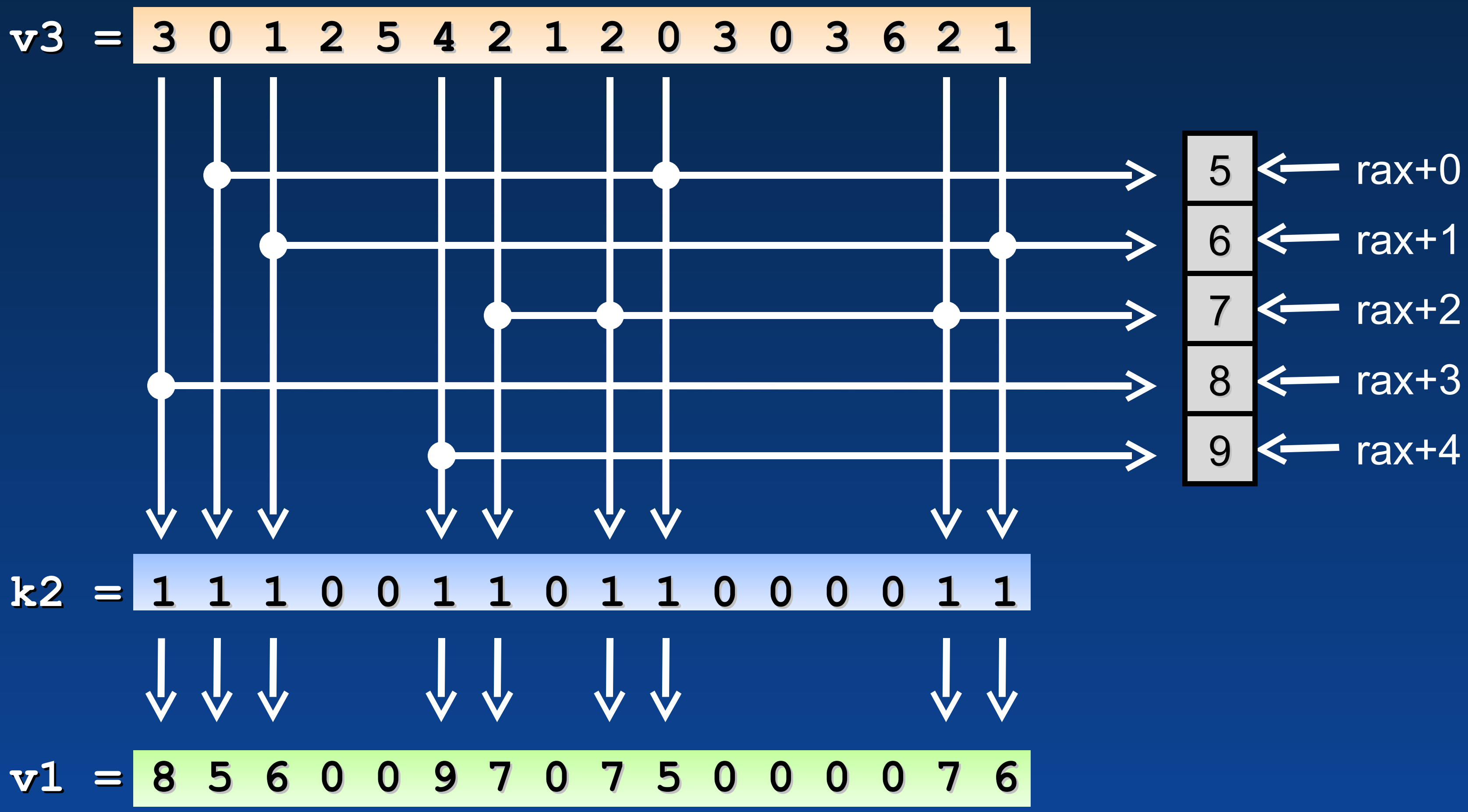
Gather

```
vgather v1{k2},[rax+v3]
```

- Gather is effectively 16 loads, one per lane
  - As usual, a mask register (k2) disables some lanes
- Vector of offsets (v3)
  - Normal x86 address mode would look like [rax+rbx]
  - But here v3 supplies 16 different offsets
  - Offsets may be optionally scaled by 2, 4 or 8 bytes
  - Added to a standard x86 base pointer (rax)
- Offsets can point anywhere in memory
  - Multiple offsets can point to the same place

16 independent offsets into memory

```
vgather v1{k2},[rax+v3]
```



v3 = 3 0 1 2 5 4 2 1 2 0 3 0 3 6 2 1

k2 = 1 1 1 0 0 1 1 0 1 1 0 0 0 0 1 1

v1 = 8 5 6 0 0 9 7 0 7 5 0 0 0 0 7 6

Scatter

```
vscatter [rax+v3]{k2}, v1
```

Same as gather, but in reverse

- Stores a vector of values to 16 different places in memory

If two offsets point to the same place, results are not obvious

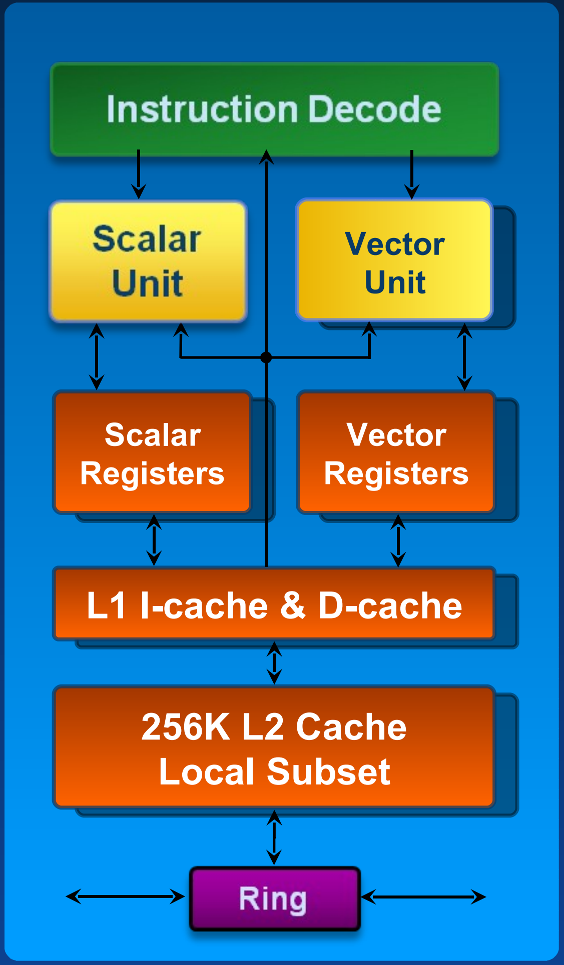
- One of them will “win”, but it’s difficult to know which
- Technically it is well-defined, but I advise not relying on it

Gather/scatter speed

- Gather/scatter limited by cache speed
- L1$ can only handle a few accesses per clock, not 16 different ones
  - Address generation and virtual->physical are expensive
  - Exact performance varies
- Offsets referring to the same cache line can happen on the same clock
  - A gather where all offsets point to the same cache line will be much faster than one where they point to 16 different cache lines
  - Gather/scatter allows SOA/AOS mixing, but data layout design is still important for top speed

Writing fast code for Larrabee

Performance



Two pipelines

- One x86 scalar pipe, one LNI vector
- Every clock, you can run an instruction on each
- Similar to Pentium U/V pairing rules
- Mask operations count as scalar ops

Vector stores are special

- They can run down the scalar pipe
- Can co-issue with a vector math op
- Since vector math instructions are all load-op, and vector stores co-issue, memory access is very cheap in this architecture

Low-level performance

- Almost all vector instructions take one clock
  - Gather/scatter exceptions already mentioned
  - Most instructions have 4 clocks latency (max is 9)
- 4 threads makes good code easy to write
  - If a thread misses the cache, it goes to sleep, and its cycles are given to other threads
  - When the data comes back, the thread wakes up again
  - Branch misprediction only needs to flush instructions from the same thread – typically only costs 1-2 cycles
  - Live threads help hide latency – with 3 threads running, 4-9 clocks of latency looks like 1-2 independent instructions

Memory performance

- Good code will usually be memory-limited
  - Roughly 2 bytes per core per clock of bandwidth
- For each source find best access pattern
  - Regular layouts benefit greatly from the right pattern
  - Semi-random accesses don’t care – LRU gets 90% benefit
  - “Use once” sources get zero benefit – always stream them
- Pick the ordering that gets most benefit
  - Focus on good L2$ residency
  - Let the four threads handle L1$ residency
- Explicitly manage streaming data
  - Prefetch ahead to hide miss latency
  - Use strong & weak evict hints behind to free up cache

High-level performance

In **this** order...

Architect for many cores & threads

- Amdahl’s law is always waiting to pounce
- “Task parallel” methods take advantage of high core-to-core bandwidth and coherent caches

Design data flow for memory bandwidth

- Aim for high cache locality on regular data sources
- Explicitly stream when appropriate: read, use, evict
- Try to get the 4 threads per core working on similar data
- Coherent caches avoid “cliff edges” in performance

Think about 16-wide SIMD

- “Complete” vector instruction set means lots of options – no need to turn everything into SOA

High-level performance

1. Architect for many cores & threads
2. Design data flow for memory bandwidth
3. Think about SIMD

If this all sounds familiar...

- The same advice works for almost all multi-core systems
- The thing we focus on as programmers that is new to Larrabee – the new wide SIMD instructions – are actually the simplest thing to architect code for
- ...but they're still great fun to talk about!

The rendering pipeline on Larrabee

The standard rendering pipeline

Input assembly

Vertex shading

Triangle clip/cull

Rasterize

Early Z/stencil

Pixel shade

Late Z/stencil

FB Blend

1. Architect for many cores & threads

Partition the scene into independent chunks

- Each chunk is processed by one core, preserving order
- But different chunks can be processed out of order

Partition by rendertarget

- Modern scenes have 10-100 rendertargets
- But they have “render to texture” dependencies
- Limited parallelism, but still useful
- Create DAG of rendertargets, exploit what there is

Partition each rendertarget

- Variety of methods, we chose binning/tiling
- Tiles can be processed out of order = good parallelism

2. Design data flow for memory BW

Input assembly: indices

Vertex shading: vertices & some textures

Triangle clip/cull: indices & vertices

Rasterize: vertex positions

Early Z/stencil: depth

Pixel shade: vertex attributes & textures

Late Z/stencil: depth

FB Blend: colour

...and remember everything wants good I$ locality!

2. Design data flow for memory BW

Input assembly & vertex processing

- Typically “use once” data – order doesn’t matter much
- Also have “constant” data – lights, animation bones, etc
- Best order is submission order

Adjacent DrawPrimitive calls share little data

- So we can parcel them out to different cores
- Can do vertex shading in any order
- But must preserve original order when doing Z/pixel blend

2. Design data flow for memory BW

In pixel shaders, textures & FB use most BW

- So which should we order by?

Textures are fairly order-insensitive

- Mipmapping causes about 1.25 texel misses per sample
- Small caches give most of the available benefit
- Need impractically large caches to give benefit beyond that
- Result – they don't care much about processing order

FB colour & depth do cache very well

- Load tile into L2$, do all processing, store the tile
- Choose a tile size that fits in L2$
- Depends on pixel format, # of channels, and antialias type
- Typically 32x32 up to 128x128 pixels

3. Think about SIMD

Pixel shaders require 2x2 blocks

- Needed for derivative calculations for mipmapping
- So that’s 4-wide SIMD already
- We can pack 4 2x2 blocks together to make 16-wide SIMD
- Typically (but not always) a 4x4 block of pixels
- This also keeps texture accesses coherent enough to cache

Rasterisation uses a hierarchical descent

- Each level of the descent is 16x fewer pixels than the last
- Extensive details in Michael Abrash’s GDC 2009 talk

Other parts of the pipeline are simple

- Shade 16 vertices, setup 16 triangles, etc...
- (though the details can get a little tricky)

Our binning/tiling architecture

“Front end”

- Input assembly
- Vertex shading
- Triangle culling

Binning

- Split rendertarget into tiles
- Decide which tile(s) each triangle hits
- Make a “bin” for each tile – the list of tris that hit that tile

“Back end” - each core picks up a tile+bin

- Rasterization
- Early depth
- Pixel shading
- Depth & blend

Haven’t people tried binning before?

...and these are the things to be careful of:

Binning hardware can be a bottleneck

- High rejection rates leaves pixel shader hardware idle
- But we don’t have any dedicated units – just cores
- Cores do whatever jobs need doing & load balance

Running out of memory

- Causes fallback to driver on the host to multi-pass
- But we’re just as smart as any driver
- Multi-passing is just like doing more rendertargets

Complex intermediate structures

- Not a problem for an x86 processor

Challenges of an all-software pipeline

We can work smarter, not harder

- We can have multiple versions of the pipeline
- Each can be tuned for different app styles
- We can tune for new apps without releasing new hardware
- We can support new APIs on existing hardware

But that means a lot of code to write

- It all takes time to write, test & tune
- But once its written, it has little end-user cost (disk space)

…and in case it's not difficult enough…

- Running under a real grown-up OS
- Full pre-emptive multitasking and virtual memory
- Co-existing with other "native" apps
- All being hidden under a standard Vista/Win7 DX driver

Larrabee hardware summary

Pentium-derived x86 core

- Well-understood pipeline
- 64-bit x86 instruction set
- 4 threads per core

512-bit wide SIMD

- “Vector complete” float32, int32, float64 instruction set

Local caches

- L1 data and instruction caches
- L2 combined cache
- Coherent, use standard x86 memory ordering

High-speed ring

- Inter-processor comms and shared DRAM access

Larrabee programmer's summary

Priorities

- Think about how you can feed 100+ threads
- Think about how to optimise for limited memory bandwidth
- Think about how to do 16 things at once

You always have standard x86

- All your code will run, even if you don't SIMD it
- Great for the 90% of code that just has to work
- Profile it once it's running, find out which bits need love

Enormous math horsepower available

- 16 float32 multiply-adds per clock, plus scalar control
- Flow control and address generation are nearly free

Gather & scatter to talk to non-16 layouts

- But be mindful of cache bandwidth

Larrabee resources

GDC 2009 talks by Abrash & Forsyth

Dr. Dobb’s Journal article by Michael Abrash

SIGGRAPH 2008

- “Larrabee: A Many-Core x86 Architecture for Visual Computing” Seiler et al
- Assorted other SIGGRAPH and SIGGRAPH Asia talks

C++ Larrabee Prototype Library

- Very close to the intrinsics, works on current hardware

www.intel.com/software/graphics

Questions?

- Backup

4-wide SIMD – SOA or AOS?

Using 4-wide SSE, there are two choices

Needs reorg? Intro?

AOS or "pac XYZ_

- Each iteratior
- At most 75%

SOA or "sca XXXX

- Another regis ZZZZ
- Each iteration of code produces four results
- Code is roughly 3x as long
- 100% use of math units
- But you have to have 4 things to do
- And the data is usually not in an SOA-friendly format

16-wide SIMD - AOS

AOS is really two options:

Simple: register holds XYZ_______________

- Basically the same code as SSE
- Only at most 19% use of math units
- But can still be appropriate if you do have wide vectors
  - Matrix math, geometric algebra

4-wide: register holds XYZ_XYZ_XYZ_XYZ_

- Each iteration produces four results
- Code is the same length
- At most 75% use of math units
- But you have to have 4 things to do
- Data is often in a reasonable format

16-wide SIMD - SOA

16-wide register holds: XXXXXXXXXXXXXXXX

- Others hold YYYYYYYYYYYYYYYY, ZZZZZZZZZZZZZZZZ
- Each iteration produces 16 results
- Code is roughly 3x as long
- 100% use of math units

But you have to have 16 things to do!

- Larrabee adds **predication**
- Allows each lane to execute different code

Data is usually not in a friendly format!

- Larrabee adds **scatter/gather** support for reformatting

Allows 90% of our code to use this mode

- Very little use of AOS modes

Gather example

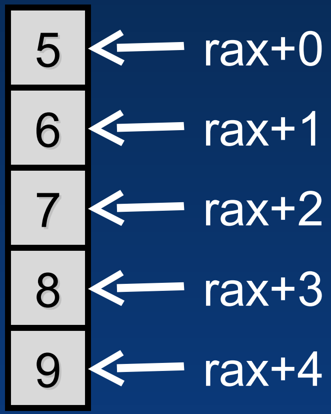
```
vgather v1{k2},[rax+v3]

v3 = 3 0 1 2 5 4 2 1 2 0 3 0 3 6 2 1

k2 = 1 1 1 0 0 1 1 0 1 1 0 0 0 0 1 1

v1 = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

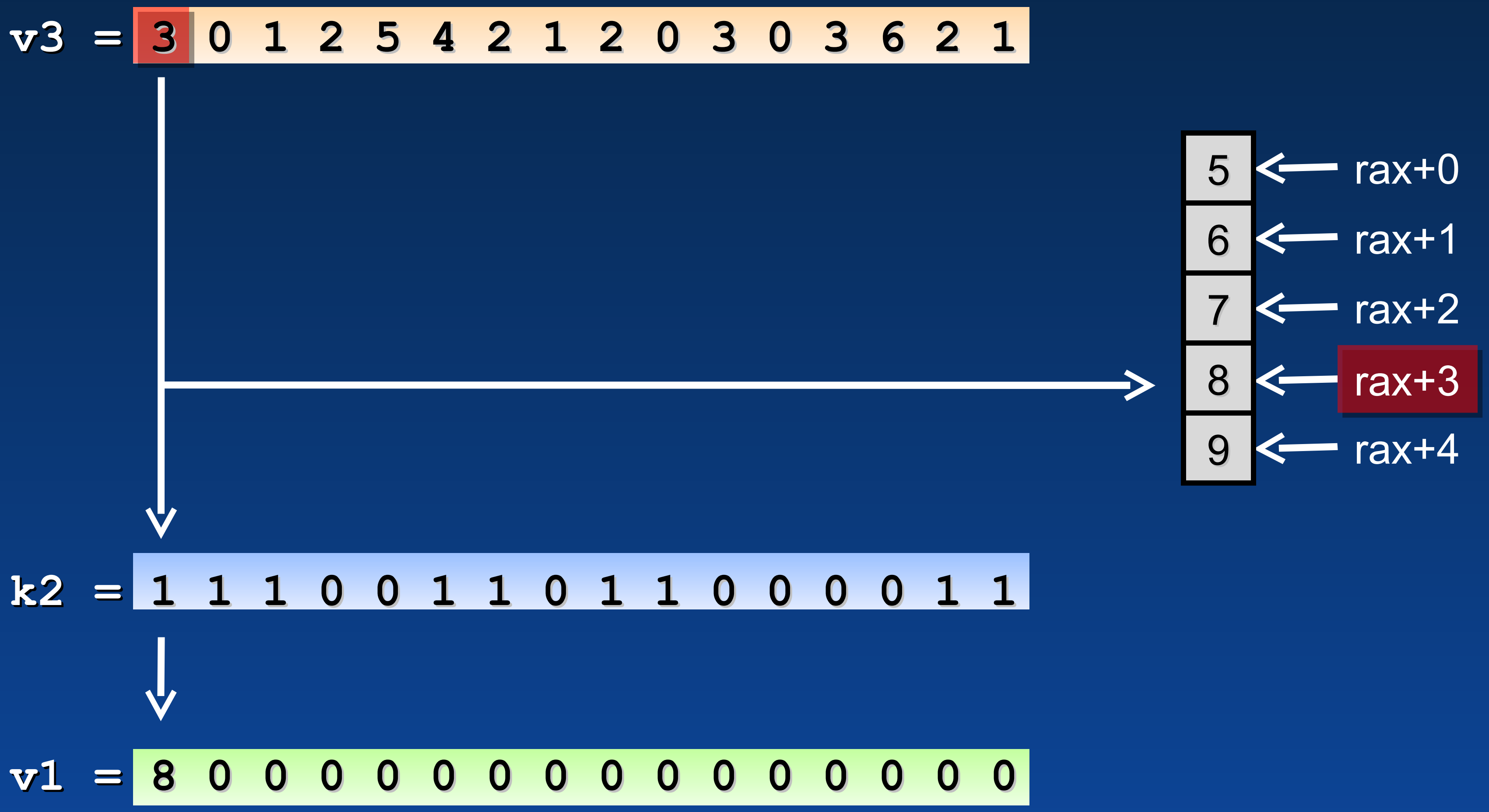
Values in memory



5	rax+0
6	rax+1
7	rax+2
8	rax+3
9	rax+4

Gather example

```
vgather v1{k2},[rax+v3]
```



v3 = 3 0 1 2 5 4 2 1 2 0 3 0 3 6 2 1

k2 = 1 1 1 0 0 1 1 0 1 1 0 0 0 0 1 1

v1 = 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

5 rax+0

6 rax+1

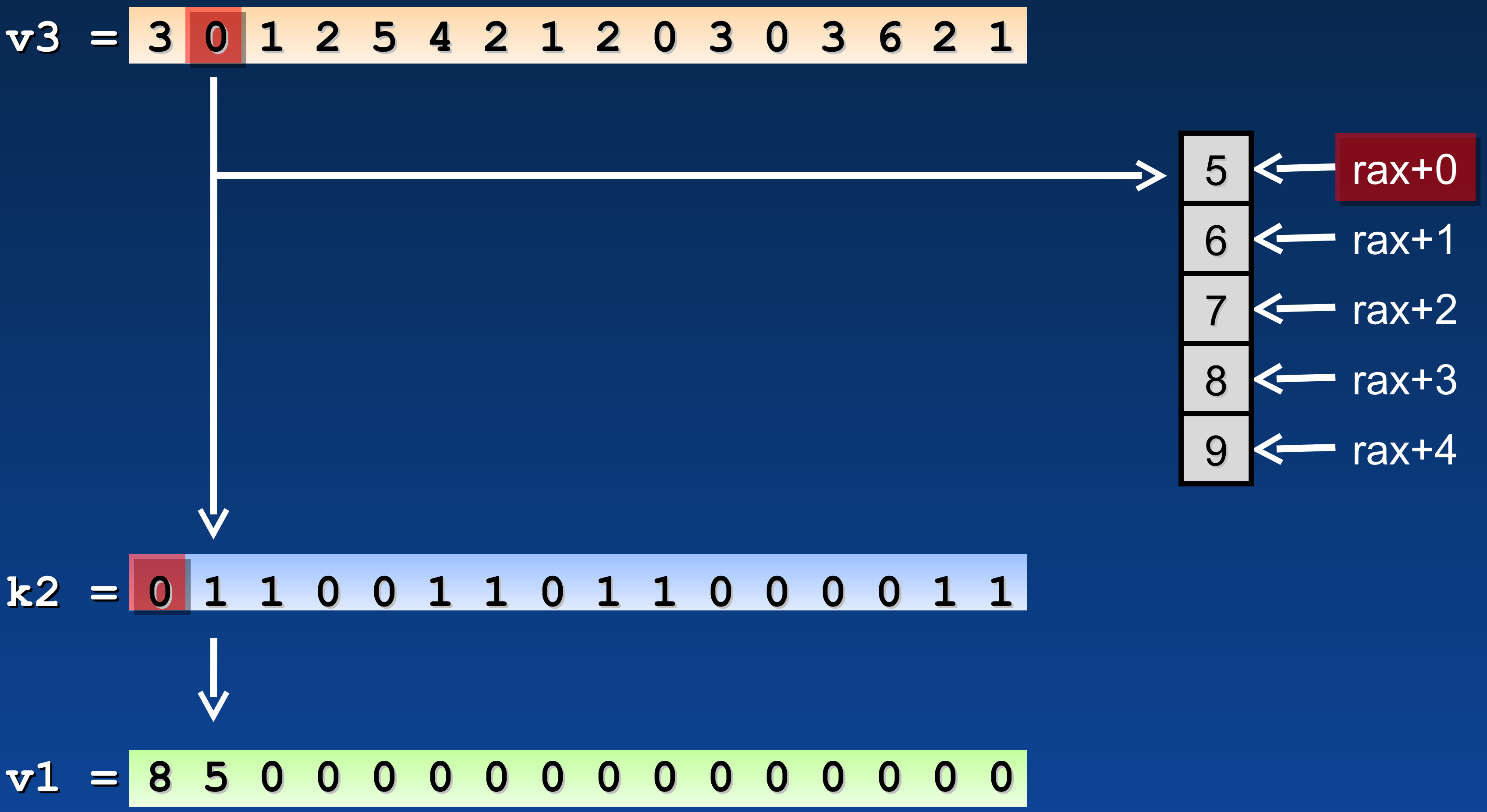
7 rax+2

8 rax+3

9 rax+4

Gather example

```
vgather v1{k2},[rax+v3]
```



v3 = 3 0 1 2 5 4 2 1 2 0 3 0 3 6 2 1

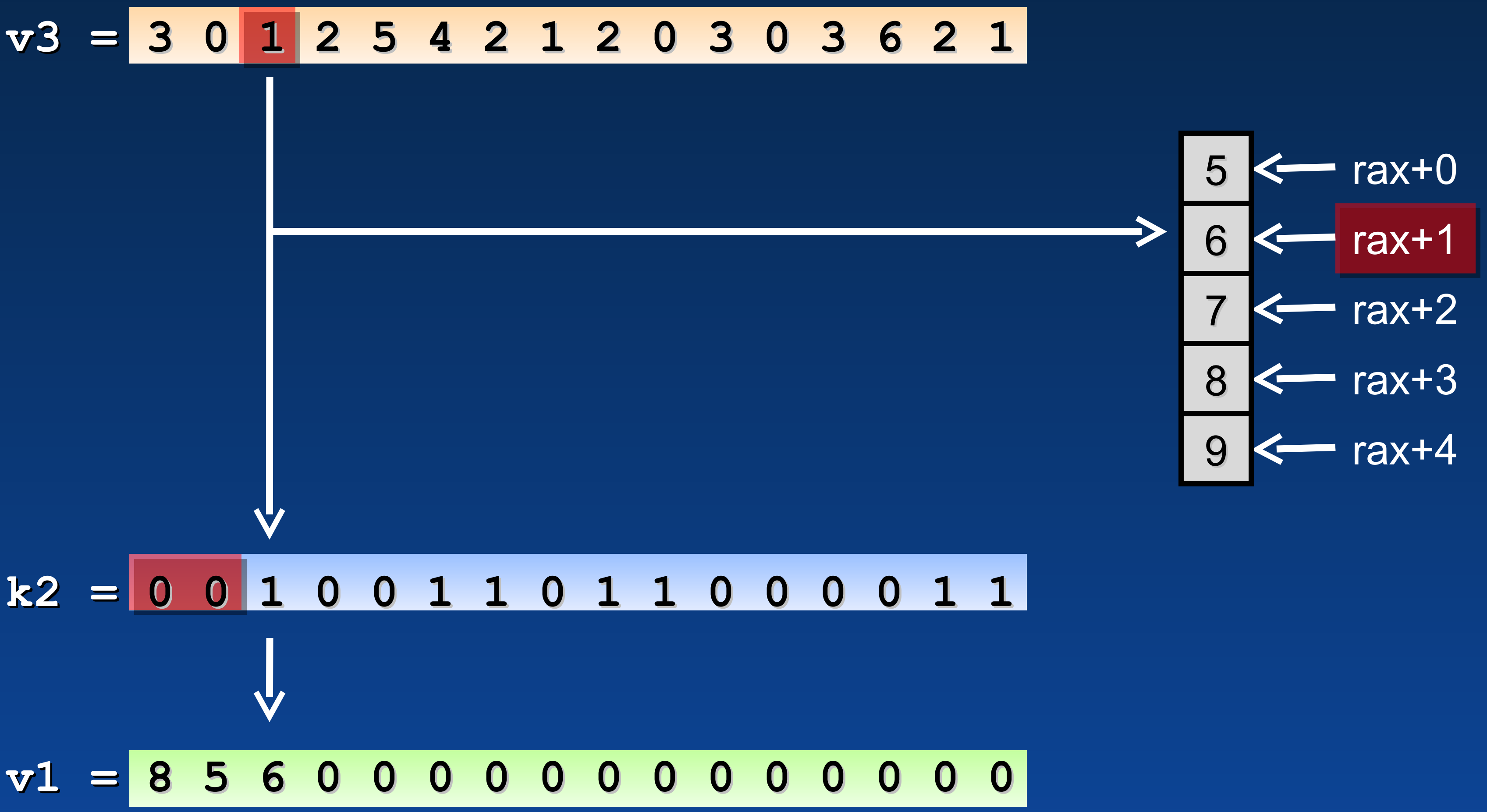
k2 = 0 1 1 0 0 1 1 0 1 1 0 0 0 0 1 1

v1 = 8 5 0 0 0 0 0 0 0 0 0 0 0 0 0 0

5	rax+0
6	rax+1
7	rax+2
8	rax+3
9	rax+4

Gather example

```
vgather v1{k2},[rax+v3]
```



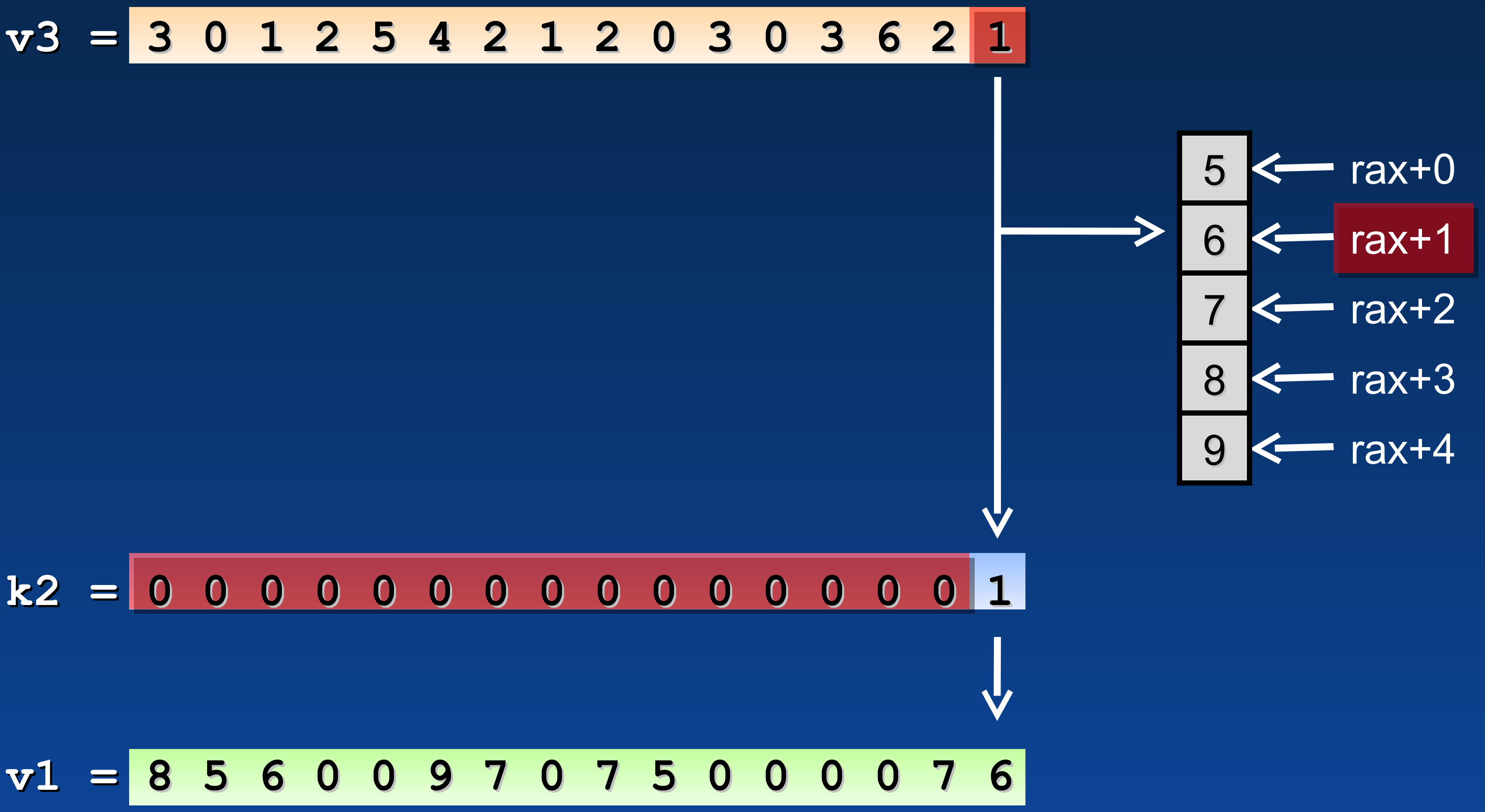
v3 = 3 0 1 2 5 4 2 1 2 0 3 0 3 6 2 1

k2 = 0 0 1 0 0 1 1 0 1 1 0 0 0 0 1 1

v1 = 8 5 6 0 0 0 0 0 0 0 0 0 0 0 0 0

Gather example

```
vgather v1{k2},[rax+v3]
```



v3 = 3 0 1 2 5 4 2 1 2 0 3 0 3 6 2 1

k2 = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

v1 = 8 5 6 0 0 9 7 0 7 5 0 0 0 0 7 6

5	rax+0
6	rax+1
7	rax+2
8	rax+3
9	rax+4

Using gather

```
y=1;
while (x>0) {
    y*=x;
    x--;
};
z = a[y];
```

...same loop as the predication example before...

...but we use the result to look up into an array. In the SIMD code, 16 different values are stored in “y”

Using gather

Indexed lookup done using gather

```
; y=1; while(x>0) { y*=x; x--; }; z = a[y];
kxnor kL, kL
vloadps vY, [ConstOne]{1to16}
loop:
vmulps vY{kL}, vY, vX
vsubps vX{kL}, vX, [ConstOne]{1to16}
vcmpps_gt kL{kL}, vX, [ConstOne]{1to16}
kortest kL, kL
jnz loop
kxnor kL, kL
vgather vZ{kL}, [rax+vY]
```

Hints and tips

- vmadd233ps
  - Does an arbit ck

Keep? Ditch?

- vcompress,
  - Allows you to ta
  - Repack into 1 efficiency
- Format conv tore
  - Keep memory 3, etc
  - Efficient use of memory bandwidth and cache space
- In most code, scalar ops are “free”
  - Hide in the shadow of vector ops
  - As do most vector stores

C++ Larrabee Prototype Library

Looks very like an intrinsics library

- But behind th
- Just a header
- Compiles on a GCC, etc
- Should work

Cut'n'paste from DaVinci pres?

Cut prototype library stuff entirely?

No claims o

- Fast enough t
- Some paths have SSE for a modest boost

Precision caution!

- It's just C, so you get whatever precision the compiler has
- May not be bit-perfect with Larrabee without care
- Multiply-add, square roots, x87 rounding mode, etc
- Same caveats as any other cross-platform development

C++ Larrabee Prototype Library

Allows experimentation with 16-wide SIMD

- Debugging is simple – just step into the function

Allows porting of algorithms and brains

- Helps people think “the other way up”
- Prototype different styles of execution

Runs on existing machines

- Allows LNI code into cross-platform libraries
- Useful for developing on laptops, etc

C++ Larrabee Prototype Library at www.intel.com/software/graphics

- Instruction count gives some feel for performance
- Please give us feedback for the final intrinsics library

C++ Larrabee Prototype Library…

```
m512 mandelbrot ( m512 x_in, m512 y_in ) {
  const float ConstOne = 1.0f;
  mmask mask = 0xFFFF;
  m512 x = x_in;
  m512 y = y_in;
  m512 iter = m512_setzero();
  do {
    m512 temp = m512_mul_ps ( x, y );
    temp = m512_add_ps ( temp, temp );
    x = m512_mask_madd213_ps ( x, mask, x, x_in );
    x = m512_mask_msub231_ps ( x, mask, y, y );
    y = m512_mask_add_ps ( y, mask, temp, y_in );
    iter = m512_mask_add_ps ( iter, mask, iter,
      m512_swizupconv_float32 ( &ConstOne, MM_BROADCAST_1X16) );
    m512 dist = m512_mul_ps ( x, x );
    dist = m512_madd231_ps ( dist, y, y );
    mask = m512_mask_cmple_ps ( mask, dist,
      m512_swizupconv_float32 ( &ConstOne, MM_BROADCAST_1X16) );
  } while ( mask !=0 );
  return iter;
}
```

...raw assembly

```
ConstOne: DD 1.0
mandelbrot:
  kxnor k2, k2
  vorpi v0, v2, v2
  vorpi v1, v3, v3
  vxorpi v4, v4, v4
  loop:
    vmulps v21{k2}, v0, v1
    vaddps v21{k2}, v21, v21
    vmadd213ps v0{k2}, v0, v2
    vmsub231ps v0{k2}, v1, v1
    vaddps v1{k2}, v21, v3
    vaddps v4{k2}, v4,
      [ConstOne]{1to16}
    vmulps  v25{k2}, v0, v0
    vmaddps v25{k2}, v1, v1
    vcmpps_le k2{k2}, v25,
      [ConstOne]{1to16}
  kortest k2, k2
  jnz loop
ret
```