

ECE 574 – Cluster Computing

Lecture 10

Vince Weaver

`http://web.eece.maine.edu/~vweaver`

`vincent.weaver@maine.edu`

25 September 2026

Announcements

- Don't forget HW#3
- HW#4 will be posted (it might be delayed a bit, it's a new homework assignment)

Homework #2 Review – HPL Measurements

Procs	1	2	4	8	16	32	64
Time	115	61	34	21	15	15	15
GFLOPS	46	87	156	255	366	346	351
Speedup	—	1.9	3.4	5.5	7.7	7.7	7.7
Peff	—	0.94	0.85	0.68	0.48	0.48	0.48

- 3bi) Speedup: (t_1/t_p)

This year moving to 32 threads didn't help

What is different? Kernel? Compiler? OpenBLAS?

Firmware?

- 3bii) Parallel efficiency: (S_p/p or T_1/pT_p)

Homework #2 Review – Scaling

- 3biii) Strong? Yes, time decreases as you add cores.
Not ideal strong scaling though.
Note: “weak” is not the same as “poor strong scaling”
- 3biv) No weak, didn't test with sizes constant
How could we test this?
- 3bv Time is less as only dgemm, not malloc or randomizing
- 3bvi More because user adds up all threads/cores

Homework #2 Review – perf record

- 3a) dgemm kernel (double-precision generic matrix-matrix multiply. algorithm kernel (core) not Linux kernel)

If you got big time in kernel, you ran perf on time

59.96%	xhpl	xhpl	[.] dgemm_kernel
19.41%	xhpl	xhpl	[.] inner_thread
6.67%	xhpl	xhpl	[.] blas_thread_server
3.83%	xhpl	xhpl	[.] HPL_lmul
1.99%	xhpl	xhpl	[.] exec_blas_async_wait
1.44%	xhpl	xhpl	[.] HPL_rand
1.34%	xhpl	xhpl	[.] HPL_dlaswp00N
1.15%	xhpl	xhpl	[.] HPL_ladd
0.93%	xhpl	xhpl	[.] HPL_setran
0.52%	xhpl	[kernel.kallsyms]	[k] clear_page_erms

Homework #2 Review – perf annotate

```
0.24 -      vfmadd231pd  %ymm0,%ymm1,%ymm4
0.27 -      vfmadd231pd  %ymm0,%ymm2,%ymm8
0.29 -      vfmadd231pd  %ymm0,%ymm3,%ymm12
0.22 -      vbroadcastsd -0x58(%rdi),%ymm0
0.44 -      vfmadd231pd  %ymm0,%ymm1,%ymm5
0.20 -      vfmadd231pd  %ymm0,%ymm2,%ymm9
0.35 -      vfmadd231pd  %ymm0,%ymm3,%ymm13
0.09 -      vbroadcastsd -0x50(%rdi),%ymm0
0.35 -      vfmadd231pd  %ymm0,%ymm1,%ymm6
```

- in `dgemm_kernel()`
 `vbroadcastsd` – broadcast 64-bit fp value from memory
 and copy 4 times in 256-bit AVX register
 `vfmadd231pd` – fused multiply-add of packed doubles.
 231 refers to the order of the operands ($2*3+1$, store
 in 1)

Homework #2 Review – skid

- 3c) skid
 - Will :pp avoid the skid?
 - Why no one thing stand out in profile?
Hand optimized assembly, people have worked a long time on optimizing this getting low hanging stuff
 - Both instructions only listed as latency 1 in the agner fogg, though that doesn't count cache access

Homework #2 – Weak Scaling Info

If ideal strong scaling, then parallel efficiency would be closer to 1. Not enough results for weak scaling.

To get 1G/core, roughly $\frac{2}{3} * n^3 = 500B * p$

Cores		N=20k	Size=3.2G	Size=1G/core	time	Speedup	GFLOPs
1	119s	3.2G	11,000	9000	11.5	—	42.5
2	64s	1.6G	16,000	11500	14	0.82	72
4	37s	0.8G	22,360	14400	14	0.82	139
8	22s	0.4G	31,600	18200	18	0.63	222
16	18s	0.2G	44,700	22900	26	0.44	304
32	18s	0.1G	63,240	28800	47	0.24	336
64	18s	0.05G	89,000	36000	93	0.12	334

SIMD / Vector Instructions

- Flynn's Taxonomy
- SISD – single instruction, single data, your normal serial processor
- SIMD – single instruction, multiple data – one instruction can act on many values in parallel
- MISD – multiple instruction, single data – wavefront or pipeline? some debate about if this really exists
- MIMD – sort of like a cluster

SIMD Benefits

- Can be faster (2, 4, 8, 16, etc. things at once)

SIMD Drawbacks

- Harder to code (assembly or clever compiler)
- Puts more pressure on memory.
- More registers to save at context switch
- Might take more setup, enough more that doing things serially might be faster

Traditional Floating Point (SISD)

- ARM/RISC
 - Dedicated floating point registers (separate from general purpose)
 - Use special instructions like fadd, fsub, fmul
- x86 8087 floating point
 - First to follow IEEE-754
 - Very complex, stack based
 - 80-bit floating point
 - Not only arithmetic but trig, exponentials, etc
 - Deprecated for SSE

SIMD / Vector Instructions

Different architectures had their own implementations

- x86: MMX/SSE/SSE2/AVX/AVX2 (semi-related FMA)
- AMD: 3DNow! (deprecated)
- PowerPC: AltiVec
- ARM: Neon / A64 SIMD / “Scalable Vector Extensions” SVE
- RISC-V: Vector Extensions

x86 : MMX

- Multi Media Extension (or Matrix Math Extension). 1997
- Pentium (P5), AMD K6, etc
- Integer operations only
- Eight 64-bit registers, MM0...MM7
Can hold 2x32, 4x16, 8x8
- Registers Overlap with floating point registers. Meant you could save/restore for free, but a hassle switching between FP and MMX and you had to switch modes which could take a while
- Math was saturating

x86 : MMX instructions

- Boolean: pxor, por, pand, pandn
- Shift: psll, psrl, psra
- Math: padd, psub, pmull
- Comparisons: pcmp
- Data Pack/Unpack: pack*, punpck*

x86: SSE (Streaming SIMD)

- Intel Pentium III (1999) Originally KNI (Katmai New Instructions)
AMD added with Athlon XP
- Eight new 128-bit registers XMM0 - XMM7
- When x86_64 came out eight more were added, XMM8..XMM15
- Registers can be used as four 32-bit floats
- Added new integer instructions but for MMX registers, still can't use MMX on XMM yet

x86: SSE Instructions

- Move: MOVSS/MOVAPS/MOVUPS
(aligned/unaligned)
- Math (Scalar)
 - Adds lowest 32 bit number in the 128-bit register
 - ADDSS, SUBSS, MULSS, DIVSS
 - Reciprocal, Square-root, Reverse Square-root
 - RCPSS, SQRTSS, MAXSS, MINSS, RSQRTSS
- Math (Packed)
 - Adds all four 32-bit numbers in the 128-bit register
 - ADDPS, SUBPS, MULPS, DIVPS
 - RCPPS, SQRTPS, MAXPS, MINPS, RSQRTPS

x86: SSE Instructions continued

- Compare (scalar): CMPSS, COMISS, UCOMISS
- Compare (packed): CMPPS
- Pack/Unpack/Shuffle: SHUFPS, UNPCKHPS, UNPCKLPS
- Shuffle moves two of the four from src1 and two of four from src2 and combine them in result
- Data Conversion: CVTSI2SS, CVTSS2SI, CVTTSS2SI, CVTPI2PS, CVTPS2PI, CVTTPS2PI
- Bitwise (Packed): ANDPS, ORPS, XORPS, ANDNPS
- Why would you do this? Sometimes you might want to mask floating point values

x86: SSE Instructions (extra)

- MMX: PMULHUW, PSADBWB, PAVGB, PAVGW, PMAXUB, PMINUB, PMAXSW, PMINSW
- MMX: PEXTRW, PINSRW, PMOVBMSKB, PSHUFW
- Threw in some cache and memory too, like MOVNTQ, MOVNTPS, MASKMOVQ, PREFETCH0, PREFETCH1, PREFETCH2, PREFETCHNTA, SFENCE

x86: SSE2

- Pentium 4 (2000), AMD Opteron (2003)
- This is the baseline for x86_64-v1 support
- Added replacement instructions for MMX that work in XMM registers
- Now instead of 128-bit registers (just four floats), you can also do:
 - 2x64-bit int
 - 2x64-bit float
 - 4x32-bit int
 - 4x32-bit float
 - 8x16 bit int
 - 16x8-bit int

x86: SSE2 instructions

- All of sse but with “double” support (addps to addpd)
- All of mmx but with “quad” support (padd to paddq)
Also can use XMM registers now, not just MM
- Various cache flush, non-temporal moves, pause (for spinlocks)

x86: SSE3

- Intel Prescott New instructions (Pentium 4 2004)
- Support for horizontal math (math within same register)
addsub/hadd/hsub
- They talk about DSP instructions but they just mean horizontal math can be used to calculate FFTs
- LDDQU unaligned load

x86: SSSE3 (Supplemental SSE3)

- Intel Core (2006)
- Integer instructions
- PSIGN (packed sign), PABS (packed absolute value)
- PALIGNR (packed align right)
- PSHUFB (packed shuffle byte)
- PMULHRSW (Packed multiply)
- PHSUB/PHADD – packed horizontal subtract/add

x86: SSE4.1

- Intel 2006? AMD support complicated
- PSADBW – complicated
- PHMINPOSUW
- PMULL – packed multiply
- DPP – dot product
- BLEND – conditional copy
- PMIN/PMAX
- ROUND
- INSERT/PIN/EXTRACT/PEXT – insert/extract bits
- PMOV – packed move
- PCMP / PACKUSDW / MOVNTDQA

x86: SSE4.2

- Intel 2008? Nehalem?
 - CRC32
 - PCMP (packed compare)
 - POPCNT/LZCNT – pop count (how many 1s) / leading zero count
- These aren't technically SSE but added at same time

x86: XOP

- AMD's attempt at SSE5
- Discontinued because of AVX
- I actually have hardware that supports XOP

x86: AVX (Advanced Vector Extensions)

- 2008 (proposed) 2011 (hardware) (Sandybridge/Bulldozer)
- Registers now 256 bits, YMM0-YMM15
 - low bits are the XMM registers
 - Also now 32 registers, YMM0-YMM31
- Most instructions updated to support the wider/more registers

x86: **AVX** new instructions

- VBROADCAST
- VINSERT/VEXTRACT
- VMASKMOV – masked move
- VPERM – shuffle/permute
- VTEST
- VZEROALL/VZEROUPPER – used when switching between 128-bit and 256-bit for compatibility reasons

x86: AVX2

- Haswell (2013) but no low end until 2020
- Integer instructions expanded to 256 bit
- Gather support, vectors from non-contiguous memory

x86: AVX2 Instructions

- VBROADCAST
- VINSERT/EXTRACT
- VGATHERQ – gather
- VPERM
- VPBLEND
- VPSRLL/VPSRL/VPSRA – shifts

x86: FMA (Fused Multiply Add)

- Sometimes considered part of AVX2, but separate
- AMD had FMA4 first but discontinued
- FMA3 (intel Haswell) (amd piledriver)
- Allows three arguments (x86 usually two)
- VFMADD: $\text{result} = +a*b+c$
VFMADD231 as seen in HW#2 results
- VFNMADD: $\text{result} = -a*b+c$
- VFMSUB, VFNMSUB
- VFMADDSUB, VFMSUBADD

AVX-512

- Support Complicated. Xeon Phi (incompatible) Some intel but removed? Modern AMD
- Various parts, not all have to be implemented
- Skylake and newer x86 (similar but not same as on Phi), but not always on desktop and they disabled with update on hybrid systems
- Issues with throttling due to heat slowing down too much
- now 512 bit registers, ZMM0-ZMM31

AVX-512 Variants

- Foundation (F) – expand everything to 512 bits
- Conflict Detection (CD) – allow more lops to vectorize
- Exponential and Reciprocal (ER) – knights landing
- Prefetch Instructions (PF) – knights landing
- Vector Neural Network (4VNNIW) – knights mill
- Fused Multiply Accumulate (4FMAPS) – knights mill
- Vector Length Extension (VL)
- Double/Quadword Extension (DQ) – new 32/64-bit instructions
- Byte/Word Extension (BW) – cover 8/16-bit integers
- ... goes on and on

AVX-512 New Instructions

- Way too many to list
- All kinds of wacky things
- Linux Torvalds Quote 2020: “I hope AVX-512 dies a painful death, and that Intel starts fixing real problems instead of trying to create magic instructions to then create benchmarks that they can look good on”

AVX-10

- Intel 2023, Granite Rapids
- Replaces AVX-512
- First version AVX10.1 does not add anything over what you might find in AVX-512
- Still not ideal
- Some of the subsets actually worse than AVX2

x86: AMX (Advanced Matrix Extensions)

- For AI/ML workloads
- Intel Sapphire Rapids 2023
- Can handle 16-bit float and 8 bit integers
- tmm0..tmm7 registers, each hold 1k of data

ARM NEON

- Cortex A8, optional on Cortex A9
- 64 or 128bit, but some procs break 128-bit into two operations
- 8, 16, 32-bit ints, single-precision floating point
- Re-uses the floating point registers
- On ARMv7 flushes sub-normals to 0 so gcc won't use it unless `-funsafe-math-optimizations` enabled

ARM Scalable Vector Extension (SVE)

- ARMv8.2-A and newer
- Scale from 128 to 2048 bits transparently
- 32 scalable registers Z0-Z31, bottom 128 bits V0-V31
NEON registers
16 predicate registers P0-P15, fault register
- int/double/float/half
- There are intrinsics
- Can also try -fvectorize with compiler

ARM SVE2

- SVE2 adds more instructions beyond HPC workloads
 - UDOT – for machine learning
 - TBL and TBX – computer vision
 - CADD and CMLA – baseband networking
 - BDEP and BEXT – genomics
 - MATCH and NMATCH – server

RISC-V vector extension (RVV)

- Vector-length agnostic?
- Can set what vector length you want on the fly?
- complicated

SSE example (From Wikipedia)

Doing a 4 element single-precision vector add would take 4 separate floating point adds:

```
vec_res.x = v1.x + v2.x;  
vec_res.y = v1.y + v2.y;  
vec_res.z = v1.z + v2.z;  
vec_res.w = v1.w + v2.w;
```

With SSE you only need one add instruction:

```
; move aligned packed single precision  
movaps xmm0, [v1]           ;xmm0 = v1.w / v1.z / v1.y / v1.x  
addps  xmm0, [v2]           ;xmm0 = v1.w+v2.w / v1.z+v2.z /  
                             ;      v1.y+v2.y / v1.x+v2.x  
  
movaps [vec_res], xmm0
```

Intrinsics

TODO: make this match the previous example

```
__m256i in1;
```

```
/* vmovdqa (%rcx),%ymm1 */
```

```
__m256i filter_avx = _mm256_load_si256( (__m256i *)filter);
```

Enabling with Compiler

- To enable any vectorization on gcc need `-O3` and to get AVX need to specify something like `-march=haswell` or `-march=native`
- To see if any vectors worked `-fopt-info-vec`
- To see why not vectorizing: `-fopt-info-vec-missed`

Intrinsics

- One way is to program in assembly language with some obscure opcodes: an example PMADDWD 16-bit integer parallel 128-bit multiply and add
- On recent gcc and other compilers there are “intrinsics” to use in C, for example you can use `_mm_madd_epi16()` to do a PMADDWD instruction

Initial SIMD try

9 values from the three input pointers (16-bit)

A B C X D E F X G H I X X X X X

The sobel filter values (16-bit)

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

Multiply and add all in parallel

A1+B2 C3+0 D4+E5 F6+0 G7+H8 I9+00 0+0 0+0

Rearrange and then do a "horizontal add"

A1+B2+G7+H8 C3+I9 D4+E5 F6+0

Another Horizontal Add

0 0 A1+B2+G7+H8+C3+I9 D4+E5+F6

Another Horizontal Add

0 0 0 A1+B2+G7+H8+C3+I9+D4+E5+F6

Convert to 16-bit result, saturate, and be done

The 18 ops (9mul/9add) turned into 4 ops

Problems

- Math is very fast, handfull of instructions
- Problem is getting memory from 3 pointers with 3-byte offsets into registers (can we make jpeg load as 4-bytes?)
- This is a “scatter/gather” problem found often with SIMD (and GPU)
- There are instructions to try to gather the values together, but not really suited for this
- Once you do it manually performance is actually worse than regular code
- Challenge: if picture not multiple of 16-bytes

Improved SIMD – Can we do better?

With many problems: re-think outside the serial box

Load full 16 bytes of pixel info from the three pointers,
multiply by the 9 values in sobel filter, shifting right by 3

```
A * RGB RGB RGB RGB RGB RGB R
B *      RGB RGB RGB RGB RGB R
C *      RGB RGB RGB RGB RGB R
D * RGB RGB RGB RGB RGB RGB R
E *      RGB RGB RGB RGB RGB R
F *      RGB RGB RGB RGB RGB R
G * RGB RGB RGB RGB RGB RGB R
H *      RGB RGB RGB RGB RGB R
+ I *      RGB RGB RGB RGB RGB R
=====
```

RGB RGB RGB RGB R 13 values of result

Use compare instruction to saturate in parallel

Store out the 13 bytes at once

So (18*13) operations reduced to (~20) I think. Still haven't tried this yet