

CIS 501 Computer Architecture

Unit 10: Vectors & Data-Level Parallelism

Slides developed by Milo Martin & Amir Roth at the University of Pennsylvania with sources that included University of Wisconsin slides by Mark Hill, Guri Sohi, Jim Smith, and David Wood.

Data-Level Parallelism

- **Data-level parallelism (DLP)**
 - Single operation repeated on multiple data elements
 - SIMD (**S**ingle-**I**nstruction, **M**ultiple-**D**ata)
 - Less general than ILP: parallel insns are all same operation
 - Exploit with **vectors**
- Old idea: Cray-1 supercomputer from late 1970s
 - Eight 64-entry x 64-bit floating point “vector registers”
 - 4096 bits (0.5KB) in each register! 4KB for vector register file
 - Special vector instructions to perform vector operations
 - Load vector, store vector (wide memory operation)
 - Vector+Vector or Vector+Vector
 - addition, subtraction, multiply, etc.
 - In Cray-1, each instruction specifies 64 operations!
 - ALUs were expensive, so one operation per cycle (not parallel)

How to Compute This Fast?

- Performing the **same** operations on **many** data items
 - Example: SAXPY

```
for (I = 0; I < 1024; I++) {
  Z[I] = A*X[I] + Y[I];
}
L1: ldf [X+r1]->f1 // I is in r1
    mulf f0,f1->f2 // A is in f0
    ldf [Y+r1]->f3
    addf f2,f3->f4
    stf f4->[Z+r1]
    addi r1,4->r1
    blti r1,4096,L1
```

- Instruction-level parallelism (ILP) - fine grained
 - Loop unrolling with static scheduling –or– dynamic scheduling
 - Wide-issue superscalar (non-)scaling limits benefits
- Thread-level parallelism (TLP) - coarse grained
 - Multicore
- Can we do some “medium grained” parallelism?

Today’s “CPU” Vectors / SIMD

Example Vector ISA Extensions (SIMD)

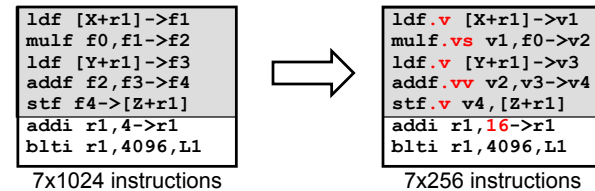
- Extend ISA with floating point (FP) vector storage ...
 - Vector register:** fixed-size array of 32- or 64- bit FP elements
 - Vector length:** For example: 4, 8, 16, 64, ...
- ... and example operations for vector length of 4
 - Load vector: `ldf.v [X+r1]->v1`

```
ldf [X+r1+0]->v1_0
ldf [X+r1+1]->v1_1
ldf [X+r1+2]->v1_2
ldf [X+r1+3]->v1_3
```
 - Add two vectors: `addf.vv v1,v2->v3`

```
addf v1_i,v2_i->v3_i (where i is 0,1,2,3)
```
 - Add vector to scalar: `addf.vs v1,f2,v3`

```
addf v1_i,f2->v3_i (where i is 0,1,2,3)
```
- Today's vectors: short (128 or 256 bits), but fully parallel

Example Use of Vectors – 4-wide



- Operations
 - Load vector: `ldf.v [X+r1]->v1`
 - Multiply vector to scalar: `mulf.vs v1,f2->v3`
 - Add two vectors: `addf.vv v1,v2->v3`
 - Store vector: `stf.v v1->[X+r1]`
- Performance?
 - Best case: 4x speedup
 - But, vector instructions don't always have single-cycle throughput
 - Execution width (implementation) vs vector width (ISA)

Vector Datapath & Implementatoin

- Vector insn. are just like normal insn... only "wider"
 - Single instruction fetch (no extra N^2 checks)
 - Wide register read & write (not multiple ports)
 - Wide execute: replicate floating point unit (same as superscalar)
 - Wide bypass (avoid N^2 bypass problem)
 - Wide cache read & write (single cache tag check)
- Execution width (implementation) vs vector width (ISA)
 - Example: Pentium 4 and "Core 1" executes vector ops at half width
 - "Core 2" executes them at full width
- Because they are just instructions...
 - ...superscalar execution of vector instructions
 - Multiple n-wide vector instructions per cycle

Intel's SSE2/SSE3/SSE4...

- Intel SSE2 (Streaming SIMD Extensions 2) - 2001**
 - 16 128bit floating point registers (`xmm0-xmm15`)
 - Each can be treated as 2x64b FP or 4x32b FP ("packed FP")
 - Or 2x64b or 4x32b or 8x16b or 16x8b ints ("packed integer")
 - Or 1x64b or 1x32b FP (just normal scalar floating point)
 - Original SSE: only 8 registers, no packed integer support
- Other vector extensions
 - AMD 3DNow!: 64b (2x32b)
 - PowerPC AltiVEC/VMX: 128b (2x64b or 4x32b)
- Looking forward for x86
 - Intel's "Sandy Bridge" brings 256-bit vectors to x86
 - Intel's "Knights Ferry" multicore will bring 512-bit vectors to x86

Other Vector Instructions

- These target specific domains: e.g., image processing, crypto
 - Vector reduction (sum all elements of a vector)
 - Geometry processing: 4x4 translation/rotation matrices
 - Saturating (non-overflowing) subword add/sub: image processing
 - Byte asymmetric operations: blending and composition in graphics
 - Byte shuffle/permute: crypto
 - Population (bit) count: crypto
 - Max/min/argmax/argmin: video codec
 - Absolute differences: video codec
 - Multiply-accumulate: digital-signal processing
 - Special instructions for AES encryption
- More advanced (but in Intel's Larrabee/Knights Ferry)
 - Scatter/gather loads: indirect store (or load) from a vector of pointers
 - Vector mask: predication (conditional execution) of specific elements

Using Vectors in Your Code

Using Vectors in Your Code

- **Write in assembly**
 - Ugh
- **Use "intrinsic" functions and data types**
 - For example: `_mm_mul_ps()` and `"__m128"` datatype
- **Use vector data types**
 - `typedef double v2df __attribute__((vector_size(16)));`
- **Use a library someone else wrote**
 - Let them do the hard work
 - Matrix and linear algebra packages
- **Let the compiler do it (automatic vectorization, with feedback)**
 - GCC's `"-ftree-vectorize"` option, `-ftree-vectorizer-verbose=n`
 - Limited impact for C/C++ code (old, hard problem)

SAXPY Example: Best Case

- **Code**

```
void saxpy(float* x, float* y,
          float* z, float a,
          int length) {
    for (int i = 0; i < length; i++) {
        z[i] = a*x[i] + y[i];
    }
}
```
- **Scalar**

```
.L3:
    movss (%rdi,%rax), %xmm1
    mulss %xmm0, %xmm1
    addss (%rsi,%rax), %xmm1
    movss %xmm1, (%rdx,%rax)
    addq $4, %rax
    cmpq %rcx, %rax
    jne .L3
```
- **Auto Vectorized**

```
.L6:
    movaps (%rdi,%rax), %xmm1
    mulps %xmm2, %xmm1
    addps (%rsi,%rax), %xmm1
    movaps %xmm1, (%rdx,%rax)
    addq $16, %rax
    incl %r8d
    cmpl %r8d, %r9d
    ja .L6
```

 - + Scalar loop to handle last few iterations (if `length % 4 != 0`)
 - "mulps": multiply packed 'single'

SAXPY Example: Actual

- Code

```
void saxpy(float* x, float* y,
          float* z, float a,
          int length) {
    for (int i = 0; i < length; i++) {
        z[i] = a*x[i] + y[i];
    }
}
```

- Scalar

```
.L3:
movss (%rdi,%rax), %xmm1
mulss %xmm0, %xmm1
addss (%rsi,%rax), %xmm1
movss %xmm1, (%rdx,%rax)
addq $4, %rax
cmpq %rcx, %rax
jne .L3
```

- Auto Vectorized

```
.L8:
movaps %xmm3, %xmm1
movaps %xmm3, %xmm2
movlps (%rdi,%rax), %xmm1
movlps (%rsi,%rax), %xmm2
movhps 8(%rdi,%rax), %xmm1
movhps 8(%rsi,%rax), %xmm2
mulps %xmm4, %xmm1
incl %r8d
addps %xmm2, %xmm1
movaps %xmm1, (%rdx,%rax)
addq $16, %rax
cmpl %r9d, %r8d
jb .L8
```

- + Explicit alignment test
- + Explicit aliasing test

Bridging "Best Case" and "Actual"

- Align arrays

```
typedef float afloat __attribute__((__aligned__(16)));
void saxpy(afloat* x,
          afloat* y,
          afloat* z,
          float a, int length) {
    for (int i = 0; i < length; i++) {
        z[i] = a*x[i] + y[i];
    }
}
```

- Avoid aliasing check

```
typedef float afloat __attribute__((__aligned__(16)));
void saxpy(afloat* __restrict x,
          afloat* __restrict y,
          afloat* __restrict z, float a, int length)
```

- Even with both, still has the "last few iterations" code

Reduction Example

- Code

```
float diff = 0.0;
for (int i = 0; i < N; i++) {
    diff += (a[i] - b[i]);
}
return diff;
```

- Scalar

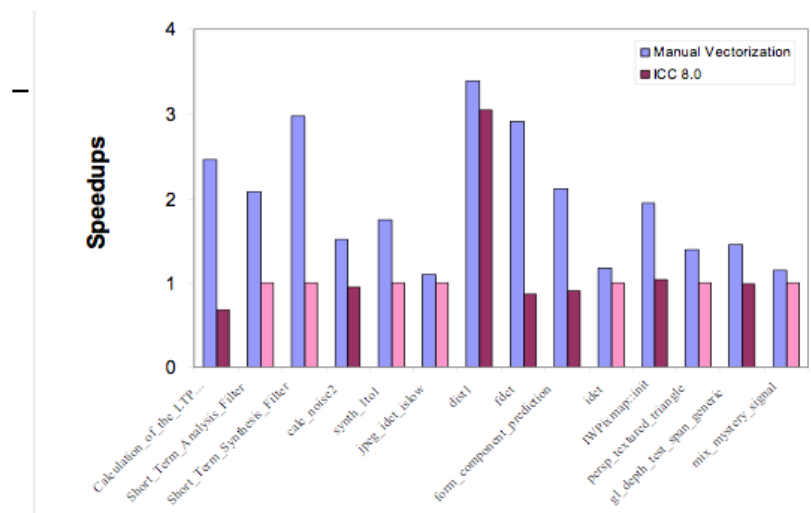
```
.L4:
movss (%rdi,%rax), %xmm1
subss (%rsi,%rax), %xmm1
addq $4, %rax
addss %xmm1, %xmm0
cmpq %rdx, %rax
jne .L4
```

- Auto Vectorized

```
.L7:
movaps (%rdi,%rax), %xmm0
incl %ecx
subps (%rsi,%rax), %xmm0
addq $16, %rax
addps %xmm0, %xmm1
cmpl %ecx, %r8d
ja .L7

haddps %xmm1, %xmm1
haddps %xmm1, %xmm1
movaps %xmm1, %xmm0
je .L3
```

- "haddps": Packed Single-FP Horizontal Add

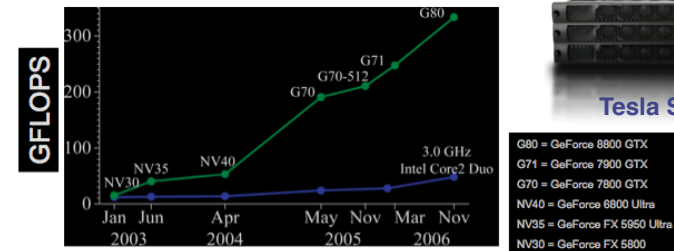


G. Ren, P. Wu, and D. Padua: An Empirical Study on the Vectorization of Multimedia Applications for Multimedia Extensions. IPDPS 2005 SSE2 on Pentium 4

Today's GPU's "SIMT" Model

Graphics Processing Units (GPU)

- Killer app for parallelism: graphics (3D games)
- A quiet revolution and potential build-up
 - Calculation: 367 GFLOPS vs. 32 GFLOPS
 - Memory Bandwidth: 86.4 GB/s vs. 8.4 GB/s
 - Until recently, programmed through graphics API



- GPU in every desktop, laptop, mobile device
- massive volume and potential impact

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009 ECE 498AL, University of Illinois, Urbana-Champaign

GPUs and SIMD/Vector Data Parallelism

- Graphics processing units (GPUs)
 - How do they have such high peak FLOPS?
 - Exploit massive data parallelism
- "SIMT" execution model
 - Single instruction multiple threads
 - Similar to both "vectors" and "SIMD"
 - A key difference: better support for conditional control flow
- Program it with CUDA or OpenCL
 - Extensions to C
 - Perform a "shader task" (a snippet of scalar computation) over many elements
 - Internally, GPU uses scatter/gather and vector mask operations

Data Parallelism Recap

- Data Level Parallelism
 - "medium-grained" parallelism between ILP and TLP
 - Still one flow of execution (unlike TLP)
 - Compiler/programmer explicitly expresses it (unlike ILP)
- Hardware support: new "wide" instructions (SIMD)
 - Wide registers, perform multiple operations in parallel
- Trends
 - Wider: 64-bit (MMX, 1996), 128-bit (SSE2, 2000), 256-bit (AVX, 2011), 512-bit (Larrabee/Knights Corner)
 - More advanced and specialized instructions
- GPUs
 - Embrace data parallelism via "SIMT" execution model
 - Becoming more programmable all the time
- Today's chips exploit parallelism at all levels: ILP, DLP, TLP