EECS 570

Lecture 6

Data-level Parallelism

Winter 2025

Prof. Satish Narayanasamy

http://www.eecs.umich.edu/courses/eecs570/

Slides developed in part by Profs. Adve, Falsafi, Martin, Roth, Nowatzyk, and Wenisch of EPFL, CMU, UPenn, U-M, UIUC.

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];
}
```

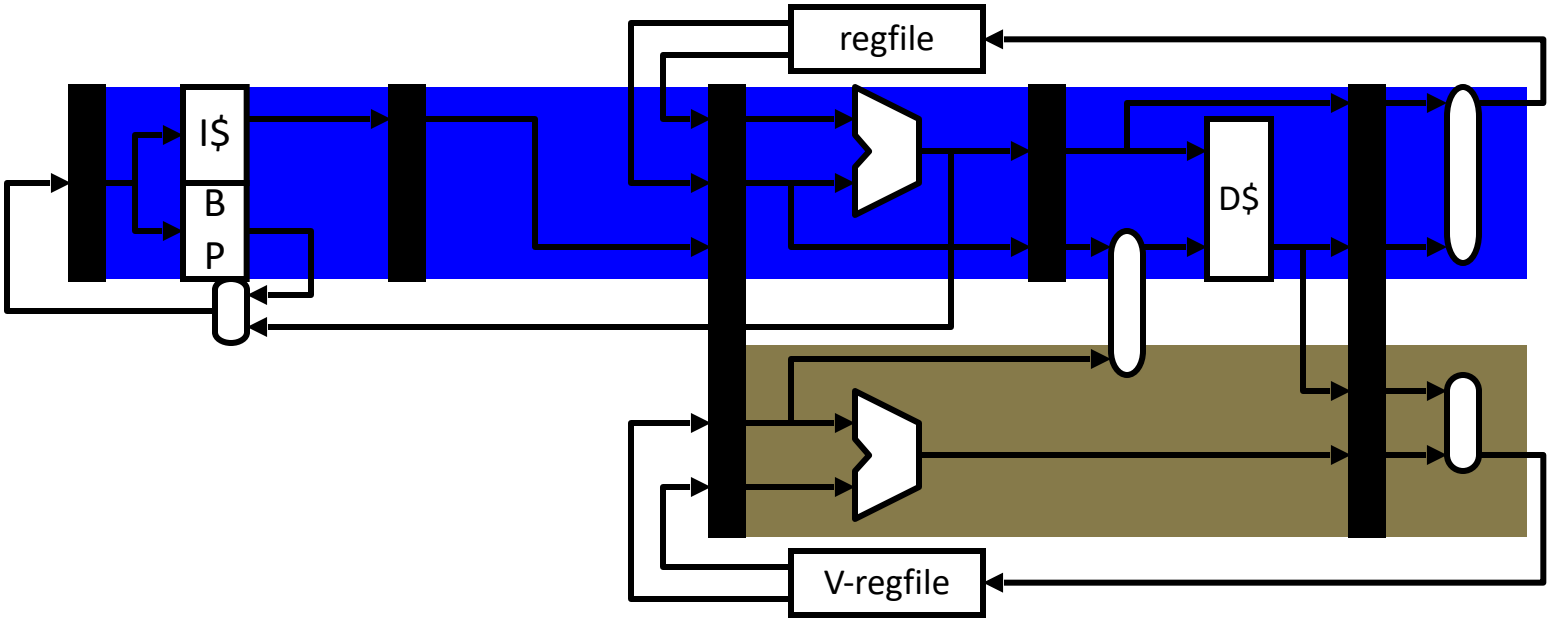
```
L1: ldf [X+r1]->f1   // I is in r1
    mulf f0,f1->f2   // A is in f0
    ldf [Y+r1]->f3
    addf f2,f3->f4
    stf f4->[Z+r1]
    addi r1,4->r1
    blti r1,4096,L1
```

- 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?

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 addition, subtraction, multiply, etc.
    - Vector+Constant addition, subtraction, multiply, etc.
    - In Cray-1, each instruction specifies 64 operations!

Vector Architectures



- One way to exploit data level parallelism: **vectors**
  - Extend processor with **vector "data type"**
  - Vector: array of 32-bit FP numbers
    - **Maximum vector length (MVL)**: typically 8–64
  - **Vector register file**: 8–16 vector registers (**`v0–v15`**)

Today's 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`

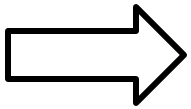
    `addf v1`$_i$`,v2`$_i$`->v3`$_i$ `(where i is 0,1,2,3)`
  - Add vector to scalar: `addf.vs v1,f2,v3`

    `addf v1`$_i$`,f2->v3`$_i$ `(where i is 0,1,2,3)`
- Today's vectors: short (128 bits), but fully parallel

Example Use of Vectors – 4-wide

```
ldf [X+r1]->f1
mulf f0,f1->f2
ldf [Y+r1]->f3
addf f2,f3->f4
stf f4->[Z+r1]
addi r1,4->r1
blti r1,4096,L1
```

7x1024 instructions



```
ldf.v [X+r1]->v1
mulf.vs v1,f0->v2
ldf.v [Y+r1]->v3
addf.vv v2,v3->v4
stf.v v4,[Z+r1]
addi r1,16->r1
blti r1,4096,L1
```

7x256 instructions
(4x fewer instructions)

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 1-cycle throughput
    - Execution width (implementation) vs vector width (ISA)

Vector Datapath & Implementation

- Vector insn. are just like normal insn… only "wider"
  - Single instruction fetch
  - Wide register read & write (not multiple ports)
  - Wide execute: replicate FP 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)
  - E.g. 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)

- Intel's AVX-512
  - Intel's "Haswell" and Xeon Phi brought 512-bit vectors to x86
  - Latest is AVX10

Key Features of AVX10 vs. Previous Generations

Feature	AVX10	AVX-512	AVX2
Vector Width	Supports **128, 256, and 512-bit** instructions on all cores (P-cores & E-cores)	Supports **128, 256, and 512-bit** but **not available on all cores**	Supports **128 & 256-bit only**
Unified Instruction Set	One instruction set across **P-cores & E-cores**	**P-cores only** in hybrid architectures	Works across all cores
Backward Compatibility	Fully backward compatible with **AVX-512 and AVX2**	Not always backward compatible across core types	Backward compatible with AVX
Simplified CPUID Interface	Easier **feature detection** for software developers	Complex CPUID feature set	Simpler feature detection
Lower Overhead	Optimized execution for mixed-width operations	Higher overhead when switching between 256-bit & 512-bit	Lower than AVX-512 but less efficient in hybrid cores
Efficient Vector Execution	Supports **masked execution & compression** for efficiency	Fully supports masking but has **higher register state pressure**	No mask registers
Energy Efficiency	Works on **E-cores**, making AVX **more power-efficient**	AVX-512 was **power-hungry**, causing downclocking	More efficient but **lower performance** than AVX-512
AI & HPC Optimization	Improved **BF16, INT8, and FP16** support for **AI workloads**	Strong AI support, but only on P-cores	Limited AI acceleration

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 Xeon Phi)
  - Scatter/gather loads: indirect store (or load) from a vector of pointers
  - Vector mask: predication (conditional execution) of specific elements

Vector Predication

```
for (i = 0; i < N; i++) {
    if (condition[i])
        result[i] = a[i] + b[i];  // Only some elements need computation
}
```

```
vaddps zmm1 {k1}, zmm2, zmm3  // Add only where k1 mask is 1
```

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"

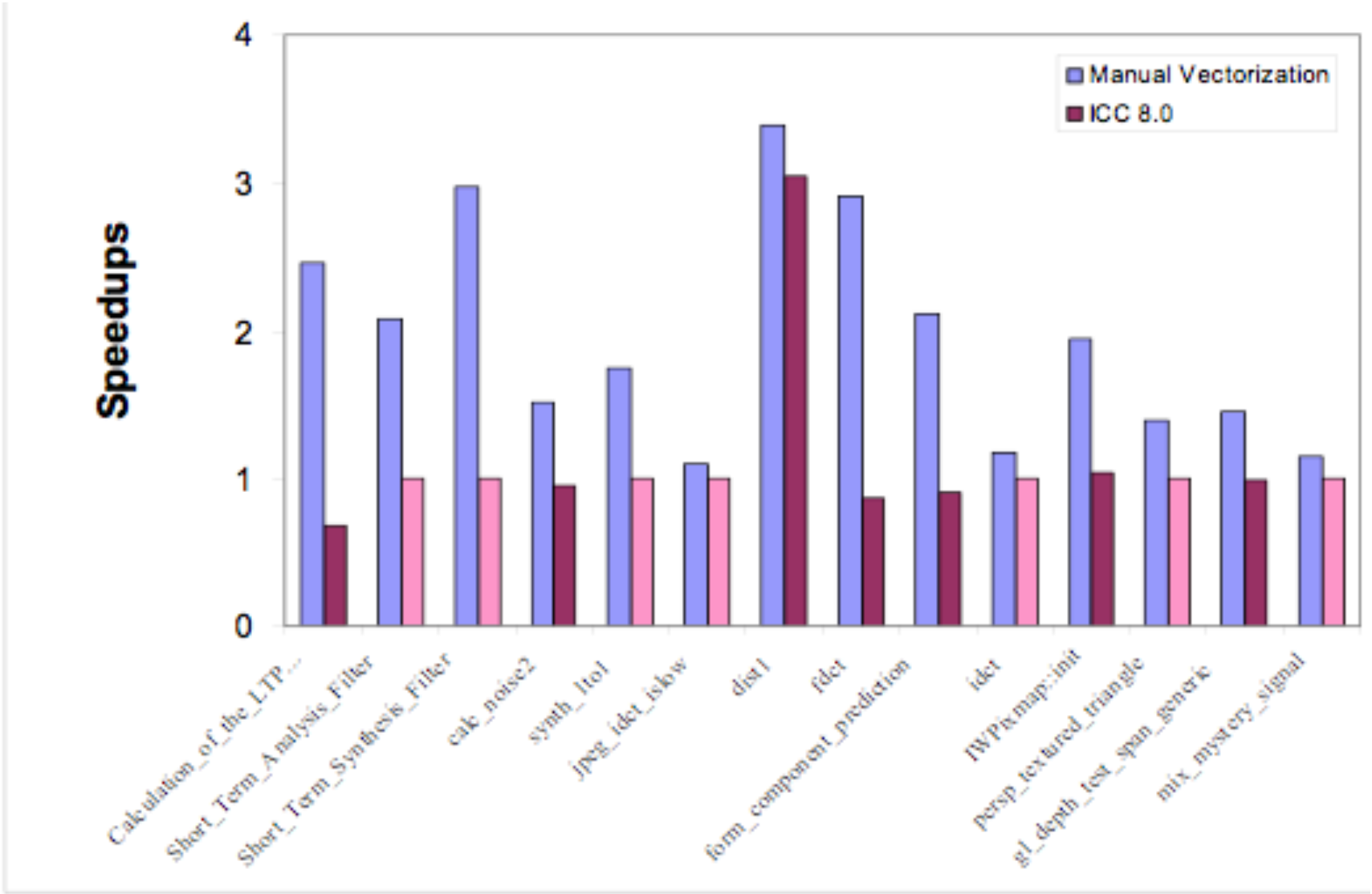
- Align arrays

```
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



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

New Developments in "CPU" Vectors

Emerging Features

- Past vectors were limited
  - Wide compute
  - Wide load/store of consecutive addresses
  - Allows for "SOA" (structures of arrays) style parallelism
- Looking forward (and backward)...
  - **Vector masks**
    - Conditional execution on a per-element basis
    - Allows vectorization of conditionals
  - **Scatter/gather**
    - a[i] = b[y[i]] b[y[i]] = a[i]
    - Helps with sparse matrices, "AOS" (array of structures) parallelism
- Together, enables a different style vectorization
  - Translate arbitrary (parallel) loop bodies into vectorized code (later)

Vector Masks (Predication)

- **Vector Masks**: 1 bit per vector element
  - Implicit predicate in all vector operations

```
for (I=0; I<N; I++) if (mask_I) { vop… }
```

  - Usually stored in a "scalar" register (up to 64-bits)
  - Used to vectorize loops with conditionals in them

`cmp_eq.v, cmp_lt.v,` etc.: sets vector predicates

```
for (I=0; I<32; I++)
   if (X[I] != 0.0) Z[I] = A/X[I];

ldf.v [X+r1] -> v1
cmp_ne.v v1,f0 -> r2        // 0.0 is in f0
divf.sv {r2} v1,f1 -> v2    // A is in f1
stf.v {r2} v2 -> [Z+r1]
```

Scatter Stores & Gather Loads

- How to vectorize:

```
for(int i = 1, i<N, i++) {
    int bucket = val[i] / scalefactor;
    found[bucket] = 1;
```

  - Easy to vectorize the divide, but what about the load/store?

- Solution: hardware support for vector "scatter stores"
    - `stf.v v2->[r1+v1]`
  - Each address calculated from $r1+v1_i$

$stf\ v2_0\text{->}[r1+v1_0],\quad stf\ v2_1\text{->}[r1+v1_1],$

$stf\ v2_2\text{->}[r1+v1_2],\quad stf\ v2_3\text{->}[r1+v1_3]$

- Vector "gather loads" defined analogously
  - `ldf.v [r1+v1]->v2`

- Scatter/gathers slower than regular vector load/store ops
  - Still provides throughput advantage over non-vector version