How to Compute This Fast?

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

```c
for (I = 0; I < 1024; I++) {
  Z[I] = A*X[I] + Y[I];
}
```

- 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 (Single-Instruction, Multiple-Data)
  - 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!
  - ALUs were expensive, did not perform 64 operations in parallel!

---

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

- Operations
  - Load vector: ldf.v [X+r1]→v1
  - Multiply vector to scalar: mulf.vs v1,f0→f2
  - 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 & Implementation

- 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”)
  - 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” will bring 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

- 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
  ```c
  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]; 
    }
  }
  ```

- Auto Vectorized
  ```c
  .L6:
  movaps (%rdi, %rax), %xmm0
  addps (%rsi, %rax), %xmm1
  movaps %xmm1, (%rdx, %rax)
  addq $16, %rax
  incl %rdx
  cmpl %rdx, %r9d
  ja .L6
  ```

  + Scalar loop to handle last few iterations (if length % 4 != 0)
  + “mulps”: multiply packed ‘single’
**SAXPY Example: Actual**

- **Code**
  ```c
  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];
      }
  }
  ```

- **Auto Vectorized**
  ```c
  .L8:
          movaps xmm1, xmm0
          movaps xmm2, xmm2
          movaps (rsi,rsi), xmm2
          movaps 8(rsi,rsi), xmm1
          muls  xmm4, xmm1
          incl  rdi
          adds  xmm2, xmm1
          movaps xmm1, (rdx,rdx)
          addq $16, trdx
          cmpl $r8d, r8d
          ja .L7
          haddps %xmm1, %xmm1
          haddps %xmm1, %xmm1
          movaps %xmm1, %xmm0
          je .L3
  
  .L3:
          movss (trdi,trax), xmm1
          mulss %xmm0, xmm1
          addq $4, rdx
          cmpq %rcx, rdx
          jne .L3
  ```

**Scalar**

- **Scalar**
  ```c
  movss (trdi,trax), xmm1
  mulss %xmm0, xmm1
  adds (rsi,rsi), xmm1
  movss %xmm1, (rdx,rdx)
  addq $4, rdx
  cmpq %rcx, rdx
  jne .L3
  ```

**Bridging “Best Case” and “Actual”**

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

**Reduction Example**

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

- **Auto Vectorized**
  ```c
  .L7:
          movaps (rdi,rsi), xmm0
          incl rdx
          subps (rsi,rsi), xmm0
          addq $16, rdx
          adds %xmm0, %xmm1
          cmpl $r8d, r8d
          ja .L7
          haddps %xmm1, %xmm1
          haddps %xmm1, %xmm1
          movaps %xmm1, %xmm0
          je .L3
  
  .L4:
          movss (trdi,trax), xmm1
          subss (rsi,rsi), xmm1
          addq $4, rdx
          cmpq %rcx, rdx
          jne .L4
  ```

- **Scalar**
  ```c
  movss (trdi,trax), xmm1
  subss (rsi,rsi), xmm1
  addq $4, rdx
  cmpq %rcx, rdx
  jne .L4
  ```

  **"haddps": Packed Single-FP Horizontal Add**

- **Even with both, still has the “last few iterations” code**

CIS 501 (Martin): Vectors
Beyond Today’s Vectors

• Today’s vectors are 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
    • \(a[i] = b[y[i]]\) \(\rightarrow b[y[i]] = a[i]\)
    • 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)

• Recall “cmov” prediction to avoid branches

  • **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];
    ```

  • `cmov_eq.v v1,f0 -> r2` \(\rightarrow\) \(0.0\) is in \(f0\)
  ```
    ldf.v [X+r1] -> v1
    cmp.ne.v v1,f0 -> r2
    divf.sv {r2} v1,f1 -> v2
    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;
    count[bucket] = count[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 \(r1+v1\),
    ```
      stf v2o->[r1+v1o], stf v2o->[r1+v1v],
      stf v2o->[r1+v1v], stf v2o->[r1+v1v]
    ```

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

  • Scatter/gathers slower than regular vector load/store ops
    • Still provides a throughput advantage over non-vector version
Today's GPU's "SIMT" Model

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
  - 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