How to Compute This Fast?

- Performing the \textbf{same} operations on \textbf{many} data items
  - Example: SAXPY

```plaintext
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

- \textbf{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 \textbf{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 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: \( \text{ldf.v} [X+r1] \rightarrow v1 \)
    - \( \text{ldf} [X+r1+0] \rightarrow v1_0 \)
    - \( \text{ldf} [X+r1+1] \rightarrow v1_1 \)
    - \( \text{ldf} [X+r1+2] \rightarrow v1_2 \)
    - \( \text{ldf} [X+r1+3] \rightarrow v1_3 \)
  - Add two vectors: \( \text{addf.vv} v1,v2 \rightarrow v3 \)
    - \( \text{addf} v1_i,v2_i \rightarrow v3_i \) (where \( i \) is 0,1,2,3)
  - Add vector to scalar: \( \text{addf.vs} v1,f2 \rightarrow v3 \)
    - \( \text{addf} v1_i,f2 \rightarrow v3_i \) (where \( i \) is 0,1,2,3)
- Today’s vectors: short (256 bits), but fully parallel

Example Use of Vectors – 4-wide

- Operations
  - Load vector: \( \text{ldf.v} [X+r1] \rightarrow v1 \)
  - Multiply vector to scalar: \( \text{mulf.vs} v1,f1 \rightarrow v2 \)
  - Add two vectors: \( \text{addf.vv} v1,v2 \rightarrow v3 \)
  - Store vector: \( \text{stf.v} v1 \rightarrow [X+r1] \)
- 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 (\( \text{xmm0–xmm15} \))
  - 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” (2011) 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

- 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)

Recap: Vectors for Exploiting DLP

- Vectors are an efficient way of capturing parallelism
  - Data-level parallelism
  - Avoid the $N^2$ problems of superscalar
  - Avoid the difficult fetch problem of superscalar
  - Area efficient, power efficient

- The catch?
  - Need code that is "vector-izable"
  - Need to modify program (unlike dynamic-scheduled superscalar)
  - Requires some help from the programmer

- Looking forward: Intel Larrabee’s vectors
  - More flexible (vector “masks”, scatter, gather) and wider
  - Should be easier to exploit, more bang for the buck

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 490AL, 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 Summary

- 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