EECS 570
Lecture 3
Data-level Parallelism
Winter 2022
Prof. Yatin Manerkar
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.
Announcements

Discussion this Friday: Programming Assignment 1
Readings

For today:

- H Kim, R Vuduc, S Baghsorkhi, J Choi, Wen-mei Hwu, xPerformance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU), Ch. 1

For next Monday:

- Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architectures, Ch. 3.1-3.3, 4.1-4.3
Agenda

Finish message passing vs shared memory from L02

Data-level parallelism
Thread-Level Parallelism

```c
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
int id,amt;
if (accts[id].bal >= amt)
{
    accts[id].bal -= amt;
    spew_cash();
}
```

- **Thread-level parallelism (TLP)**
  - Collection of asynchronous tasks: not started and stopped together
  - Data shared loosely, dynamically

- **Example:** database/web server (each query is a thread)
  - `accts` is **shared**, can’t register allocate even if it were scalar
  - `id` and `amt` are private variables, register allocated to `r1, r2`
Synchronization

• Mutual exclusion: locks, ...
• Order: barriers, signal-wait, ...

• Implemented using read/write/RMW to shared location
  □ Language-level:
    ○ libraries (e.g., locks in pthread)
    ○ Programmers can write custom synchronizations
  □ Hardware ISA
    ○ E.g., test-and-set

• OS provides support for managing threads
  □ scheduling, fork, join, futex signal/wait

We’ll cover synchronization in more detail in a couple of weeks
Paired vs. Separate Processor/Memory?

- **Separate processor/memory**
  - Uniform memory access (UMA): equal latency to all memory
  - Simple software, doesn’t matter where you put data
  - Lower peak performance
  - Bus-based UMAs common: symmetric multi-processors (SMP)

- **Paired processor/memory**
  - Non-uniform memory access (NUMA): faster to local memory
  - More complex software: where you put data matters
  - Higher peak performance: assuming proper data placement
**Shared vs. Point-to-Point Networks**

- **Shared network**: e.g., bus (left)
  - Low latency
  - Low bandwidth: doesn’t scale beyond ~16 processors
  - Shared property simplifies cache coherence protocols (later)

- **Point-to-point network**: e.g., mesh or ring (right)
  - Longer latency: may need multiple “hops” to communicate
  - Higher bandwidth: scales to 1000s of processors
  - Cache coherence protocols are complex
Implementation #1: Snooping Bus MP

- Two basic implementations
- Bus-based systems
  - Typically small: 2–8 (maybe 16) processors
  - Typically processors split from memories (UMA)
    - Sometimes multiple processors on single chip (CMP)
    - Symmetric multiprocessors (SMPs)
    - Common, I use one everyday
Implementation #2: Scalable MP

- General point-to-point network-based systems
  - Typically processor/memory/router blocks (NUMA)
    - **Glueless MP**: no need for additional “glue” chips
  - Can be arbitrarily large: 1000’s of processors
    - **Massively parallel processors (MPPs)**
  - In reality only government (DoD) has MPPs...
    - Companies have much smaller systems: 32–64 processors
    - **Scalable multi-processors**
Cache Coherence

- Two $100 withdrawals from account #241 at two ATMs
  - Each transaction maps to thread on different processor
  - Track `accts[241].bal` (address is in `r3`)
No-Cache, No-Problem

• Scenario I: processors have no caches
  - No problem
Cache Incoherence

- Scenario II: processors have write-back caches
  - Potentially 3 copies of `accts[241].bal`: memory, p0$, p1$
  - Can get incoherent (out of sync)
Snooping Cache-Coherence Protocols

Bus provides serialization point

Each cache controller “snoops” all bus transactions
  - take action to ensure coherence
    - invalidate
    - update
    - supply value
  - depends on state of the block and the protocol
Scalable Cache Coherence

- **Scalable cache coherence**: two part solution

  - **Part I**: *bus bandwidth*
    - Replace non-scalable bandwidth substrate (bus)...
    - ...with scalable bandwidth one (point-to-point network, e.g., mesh)

  - **Part II**: *processor snooping bandwidth*
    - Interesting: most snoops result in no action
    - Replace non-scalable broadcast protocol (spam everyone)...
    - ...with scalable *directory protocol* (only spam processors that care)

- We will cover this in Unit 3
Shared Memory Summary

• Shared-memory multiprocessors
  + Simple software: easy data sharing, handles both DLP and TLP
  – Complex hardware: must provide illusion of global address space

• Two basic implementations
  □ Symmetric (UMA) multi-processors (SMPs)
    ○ Underlying communication network: bus (ordered)
    + Low-latency, simple protocols
    – Low-bandwidth, poor scalability
  □ Scalable (NUMA) multi-processors (MPPs)
    ○ Underlying communication network: point-to-point (often unordered)
    + Scalable bandwidth
    – Higher-latency, complex protocols
1. Very strict QoS puts a lot of pressure on 1-thread perf
2. With low QoS constraints, balance ILP and TLP
3. Limited parallelism calls for more powerful cores
4. For medium QoS, ratio of big-to-small cores should follow ratio of big-to-small requests.

5. But, as $f_{\text{parallel}}$ decreases, big cores are rapidly favored.
Amdahl's Law for Tail Latency

[Delimitrou & Kozyrakis]

6. 30-50% area for cache is ideal for workloads with locality & strict QoS

7. Less cache needed (~30%) with QoS less strict

8. Less parallelism → need more cache
Data-Level Parallelism
How to Compute This Fast?

• Performing the **same** operations on **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?

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

**Example:**
- **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

**Lecture 3 Slide 21**
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!
• 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: \texttt{ldf.v [X+r1]} \rightarrow v1
    \begin{align*}
    \texttt{ldf [X+r1+0]} & \rightarrow v1_0 \\
    \texttt{ldf [X+r1+1]} & \rightarrow v1_1 \\
    \texttt{ldf [X+r1+2]} & \rightarrow v1_2 \\
    \texttt{ldf [X+r1+3]} & \rightarrow v1_3
    \end{align*}
  □ Add two vectors: \texttt{addf.vv v1,v2} \rightarrow v3
    \begin{align*}
    \texttt{addf v1}_i , v2_i & \rightarrow v3_i \text{ (where } i \text{ is } 0,1,2,3) \\
    \end{align*}
  □ Add vector to scalar: \texttt{addf.vs v1,f2,v3}
    \begin{align*}
    \texttt{addf v1}_i , f2 & \rightarrow v3_i \text{ (where } i \text{ is } 0,1,2,3)
    \end{align*}

• 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, f2 -> v2`
- Add two vectors: `addf.vv v2, v3 -> v4`
- 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 (\texttt{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
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
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**

```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];
    }
}
```

- **Scalar**

```c
.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**

```c
.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

```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];
    }
}
```

• Scalar

```assembly
.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

```assembly
.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 $16, %rax
    jne .L3
```

◦ + Explicit alignment test
◦ + Explicit aliasing test
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,
           __restrict__ y,
           __restrict__ z, float a, int length)
```

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

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 (maskI) { 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
    ```
    stf v2_0->[r1+v1_0],   stf v2_1->[r1+v1_1],
    stf v2_2->[r1+v1_2],   stf v2_3->[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