Languages and Abstractions for High-Performance Scientific Computing
CS598 APK

Andreas Kloezkner

Fall 2018
Outline

Introduction
  Notes
  About This Class
  Why Bother with Parallel Computers?
  Lowest Accessible Abstraction: Assembly
  Architecture of an Execution Pipeline
  Architecture of a Memory System
  Shared-Memory Multiprocessors

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation
Outline

Introduction

Notes
About This Class
Why Bother with Parallel Computers?
Lowest Accessible Abstraction: Assembly
Architecture of an Execution Pipeline
Architecture of a Memory System
Shared-Memory Multiprocessors

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation
Outline

Introduction
   Notes
About This Class
   Why Bother with Parallel Computers?
   Lowest Accessible Abstraction: Assembly
   Architecture of an Execution Pipeline
   Architecture of a Memory System
   Shared-Memory Multiprocessors

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation
Why this class?

- Setting: Performance-Constrained Code
  When is a code performance-constrained?

  A desirable quality (fidelity/capability) is limited by computational cost on a given computer.

- If your code is performance-constrained, what is the best approach?

  Use a more efficient method/algorithm.

- If your code is performance-constrained, what is the second-best approach?

  Ensure the current algorithm uses your computer efficiently. Observe that this is a desperate measure.
Examples of Performance-Constrained Codes

- Simulation codes
  - Weather/climate models
  - Oil/gas exploration
  - Electronic structure
  - Electromagnetic design
  - Aerodynamic design
  - Molecular dynamics / biological systems
  - Cryptanalysis
- Machine Learning
- Data Mining

Discussion:
- In what way are these codes constrained?
- How do these scale in terms of the problem size?
What Problem are we Trying To Solve?

$$(C_{ij})_{i,j=1}^{m,n} = \sum_{k=1}^{\ell} A_{ik} B_{kj}$$

Reference BLAS DGEMM code:
https://github.com/Reference-LAPACK/lapack/blob/master/BLAS/

OpenBLAS DGEMM code:
https://github.com/xianyi/OpenBLAS/blob/develop/kernel/x86_64

Demo: intro/DGEMM Performance

Demo Instructions: Compare OpenBLAS against Fortran BLAS on large square matrix
Goals: What are we Allowed to Ask For?

- Goal: “make efficient use of the machine”
- In general: not an easy question to answer
- In theory: limited by *some* peak machine throughput
  - Memory Access
  - Compute
- In practice: many other limits (Instruction cache, TLB, memory hierarchy, NUMA, registers)

contains:

- Class outline
- Slides/demos/materials
- Assignments
- Virtual Machine Image
- Piazza
- Grading Policies
- Video
- HW1 (soon)
Welcome Survey

Please go to:


and click on 'Start Activity'.

If you are seeing this later, you can find the activity at Activity: welcome-survey.
Grading / Workload

Four components:

- Homework: 25%
- Paper Presentation: 25%
  - 30 minutes (two per class)
  - Presentation sessions scheduled throughout the semester
  - Paper list on web page
  - Sign-up survey: soon
- Paper Reactions: 10%
- Computational Project: 40%
Approaches to High Performance

- Libraries (seen)
- Black-box Optimizing Compilers
- Compilers with Directives
- Code Transform Systems
- “Active Libraries”

Q: Give examples of the latter two.

- Code Transform System: CHiLL
- Active Library: PyTorch
Libraries: A Case Study

\[(C_{ij})_{i,j=1}^{m,n} = \sum_{k=1}^{\ell} A_{ik} B_{kj}\]

Demo: intro/DGEMM Performance

Demo Instructions: Compare OpenBLAS on large square and small odd-shape matrices
Do Libraries Stand a Chance? (in general)

- Tremendously successful approach — Name some examples
  (e.g.) LAPACK, Eigen, UMFPACK, FFTW, Numpy, Deal.II

- Saw: Three simple integer parameters suffice to lose ’good’ performance

- Separation of Concerns
  Example: Finite differences — e.g. implement $\partial_x$, $\partial_y$, $\partial_z$ as separate (library) subroutines — What is the problem?
  Data locality $\rightarrow$ data should be traversed once, $\partial_x$, $\partial_y$, $\partial_z$ computed together
  Separation of concerns $\rightarrow$ each operator traverses the data separately.

- Flexibility and composition
Why is black-box optimizing compilation so difficult?

- Application developer knowledge lost
  - Simple example: “Rough” matrix sizes
  - Data-dependent control flow
  - Data-dependent access patterns
  - Activities of other, possibly concurrent parts of the program
  - Profile-guided optimization can recover some knowledge

- Obtain proofs of required properties
- Size of the search space

Consider
Directive-Based Compiler: Challenges

What is a directive-based compiler?

Demo Instructions: Show 12dformta_qbx from pyfmmlib/vec_wrappers.f90.

- Generally same as optimizing compiler
- Make use of extra promises made by the user
- What should the user promise?
- Ideally: feedback cycle between compiler and user
  - Often broken in both directions
  - User may not know what the compiler did
  - Compiler may not be able to express what it needs
- Directives: generally not mandatory
Lies, Lies Everywhere

- Semantics form a contract between programmer and language/environment
- Within those bounds, the implementation is free to do as it chooses
- True at every level:
  - Assembly
  - “High-level” language (C)

Give examples of lies at these levels:

- Assembly: Concurrent execution
- “High-level” language (C): (e.g.) strength reduction, eliminated ops

One approach: *Lie to yourself*

- “Domain-specific languages” ← A fresh language, I can do what I want!
- Consistent semantics are notoriously hard to develop
  - Especially as soon as you start allowing subsets of even (e.g.)
Class Outline

High-level Sections:

- Intro, Armchair-level Computer Architecture
- Machine Abstractions
- Performance: Expectation, Experiment, Observation
- Programming Languages for Performance
- Program Representation and Optimization Strategies
- Code Generation/JIT
Survey: Class Makeup

- Compiler class: 11 no, 3 yes
- HPC class: 10 yes, 4 no
- C: very proficient on average
- Python: proficient on average
- Assembly: some have experience
- GPU: Half the class has experience, some substantial
- CPU perf: Very proficient
- 10 PhD, 4 Masters, mostly CS (plus physics, CEE, MechSE)
Survey: Learning Goals

- How to use hardware efficiently to write fast code (1 response)
- I want to learn about commonly encountered problems in HPC and efficient ways to approach and solve them. (1 response)
- About writing high performance code for large scale problems. (1 response)
- More (and more) about high-performance computing beyond parallel programming. (1 response)
- This summer (while interning at Sandia national labs), I got familiar with GPU programming using Kokkos as the back end. I enjoyed this work immensely, and hope to continue learning about it, especially so that I can become better at writing GPU code myself. I am also interested in the relationship between a higher level abstraction (Kokkos), the compiler, and the actual compute device (GPU/CPU) relate together, and what tricks we have to help fix issues regarding this. For example, Kokkos uses a small amount of template metaprogramming to convert the source code into actual code. (1 response)
- Some GPU stuff, course description sounded interesting for my research in HPC/Parallel Computing. Would be interesting to look at different programming models or abstractions for HPC. (1 response)
- Getting better at doing high performance computing. (1 response)
- Become more familiar with abstractions (1 response)
- I want to be able to auto generate performance portable C++ code, specifically for small batched tensor contractions. (1 response)
- Languages and abstractions for high-performance scientific computing (1 response)
- Investigating problems in high performance computing and looking for solutions, especially large-scale and using GPUs. (1 response)
- Better ways to efficiently (in terms of human time) write high-performance code that may be useful to/readable by others (1 response)
- About high-level languages and frameworks for high performance computing, the different interfaces they expose, compilation and runtime techniques they use, and the tradeoffs of these for an application developer. (1 response)
Outline

Introduction

Notes

About This Class

Why Bother with Parallel Computers?

Lowest Accessible Abstraction: Assembly

Architecture of an Execution Pipeline

Architecture of a Memory System

Shared-Memory Multiprocessors

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation
Moore’s Law

**Issue:** More transistors = faster?

\[
\frac{\text{Work}}{s} = \text{Clock Frequency} \times \frac{\text{Work}}{\text{Clock}}
\]
## Dennard Scaling of MOSFETs

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimension</td>
<td>$1/\kappa$</td>
</tr>
<tr>
<td>Voltage</td>
<td>$1/\kappa$</td>
</tr>
<tr>
<td>Current</td>
<td>$1/\kappa$</td>
</tr>
<tr>
<td>Capacitance</td>
<td>$1/\kappa$</td>
</tr>
<tr>
<td>Delay Time</td>
<td>$1/\kappa$</td>
</tr>
<tr>
<td>Power dissipation/circuit</td>
<td>$1/\kappa^2$</td>
</tr>
<tr>
<td>Power density</td>
<td>1</td>
</tr>
</tbody>
</table>

[Dennard et al. ’74, via Bohr ’07]

- Frequency = Delay time$^{-1}$
MOSFETs ("CMOS" – "complementary" MOS): Schematic

[Dennard et al. ‘74]
MOSFETs: Scaling

'New' problem at small scale:
Sub-threshold leakage (due to low voltage, small structure)
Dennard scaling is over – and has been for a while.
# Peak Architectural Instructions per Clock: Intel

<table>
<thead>
<tr>
<th>CPU</th>
<th>IPC</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium 1</td>
<td>1.1</td>
<td>1993</td>
</tr>
<tr>
<td>Pentium MMX</td>
<td>1.2</td>
<td>1996</td>
</tr>
<tr>
<td>Pentium 3</td>
<td>1.9</td>
<td>1999</td>
</tr>
<tr>
<td>Pentium 4 (Willamette)</td>
<td>1.5</td>
<td>2003</td>
</tr>
<tr>
<td>Pentium 4 (Northwood)</td>
<td>1.6</td>
<td>2003</td>
</tr>
<tr>
<td>Pentium 4 (Prescott)</td>
<td>1.8</td>
<td>2003</td>
</tr>
<tr>
<td>Pentium 4 (Gallatin)</td>
<td>1.9</td>
<td>20</td>
</tr>
<tr>
<td>Pentium D</td>
<td>2</td>
<td>2005</td>
</tr>
<tr>
<td>Pentium M</td>
<td>2.5</td>
<td>2003</td>
</tr>
<tr>
<td>Core 2</td>
<td>3</td>
<td>2006</td>
</tr>
<tr>
<td>Sandy Bridge...</td>
<td>4ish</td>
<td>2011</td>
</tr>
</tbody>
</table>

[Charlie Brej](http://brej.org/blog/?p=15)

Discuss: How do we get out of this dilemma?
The Performance Dilemma

▶ IPC: Brick Wall
▶ Clock Frequency: Brick Wall

Ideas:

▶ Make one instruction do more copies of the same thing ("SIMD")
▶ Use copies of the same processor ("SPMD"/"MPMD")

Question: What is the conceptual difference between those ideas?

▶ SIMD executes multiple program instances in lockstep.
▶ SPMD has no synchronization assumptions.
The Performance Dilemma: Another Look

- **Really:** A crisis of the ’starts-at-the-top-ends-at-the-bottom’ programming model
- **Tough luck:** Most of our codes are written that way
- **Even tougher luck:** Everybody on the planet is *trained* to write codes this way

So:

- **Need:** Different tools/abstractions to write those codes
Outline

Introduction

Notes
About This Class
Why Bother with Parallel Computers?
Lowest Accessible Abstraction: Assembly
Architecture of an Execution Pipeline
Architecture of a Memory System
Shared-Memory Multiprocessors

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation
A Basic Processor: Closer to the Truth

- Address ALU
- Register File
- Flags
- Data ALU
- Address ALU
- Memory Interface
- Address Bus
- Data Bus
- Internal Bus
- Control Unit
- PC
- Data ALU

- loosely based on Intel 8086
- What’s a bus?
A Very Simple Program

```
int a = 5;
int b = 17;
int z = a * b;
```

Things to know:

- **Question**: Which is it?
  - `<opcode> <src>, <dest>`
  - `<opcode> <dest>, <src>`

- **Addressing modes** (Immediate, Register, Base plus Offset)

- **0xHexadecimal**
A Very Simple Program: Another Look

4: c7 45 f4 05 00 00 00 00 movl $0x5,-0xc(%rbp)
b: c7 45 f8 11 00 00 00 00 movl $0x11,-0x8(%rbp)
12: 8b 45 f4 mov -0xc(%rbp),%eax
15: 0f af 45 f8 imul -0x8(%rbp),%eax
19: 89 45 fc mov %eax,-0x4(%rbp)
1c: 8b 45 fc mov -0x4(%rbp),%eax
A Very Simple Program: Intel Form

4: c7 45 f4 05 00 00 00 mov DWORD PTR [rbp-0xc],0x5
b: c7 45 f8 11 00 00 00 mov DWORD PTR [rbp-0x8],0x11
12: 8b 45 f4 mov eax,DWORD PTR [rbp-0xc]
15: 0f af 45 f8 imul eax,DWORD PTR [rbp-0x8]
19: 89 45 fc mov DWORD PTR [rbp-0x4],eax
1c: 8b 45 fc mov eax,DWORD PTR [rbp-0x4]

- “Intel Form”: (you might see this on the net)
  <opcode> <sized dest>, <sized source>

- Previous: “AT&T Form”: (we’ll use this)

- Goal: Reading comprehension.

- Don’t understand an opcode?
Assembly Loops

```c
int main()
{
    int y = 0, i;
    for (i = 0; y < 10; ++i)
        y += i;
    return y;
}
```

Things to know:

- **Condition Codes (Flags)**: Zero, Sign, Carry, etc.
- **Call Stack**: Stack frame, stack pointer, base pointer
- **ABI**: Calling conventions

Demo Instructions: C → Assembly mapping from [https://github.com/ynh/cpp-to-assembly](https://github.com/ynh/cpp-to-assembly)
Demo: intro/Assembly Reading Comprehension

Demo: Source-to-assembly mapping

Code to try:

```c
int main()
{
    int y = 0, i;
    for (i = 0; y < 10; ++i)
        y += i;
    return y;
}
```
Outline

Introduction
Notes
About This Class
Why Bother with Parallel Computers?
Lowest Accessible Abstraction: Assembly

Architecture of an Execution Pipeline
Architecture of a Memory System
Shared-Memory Multiprocessors

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation
Modern Processors?

All of this can be built in about 4000 transistors. (e.g. MOS 6502 in Apple II, Commodore 64, Atari 2600)

So what exactly are Intel/ARM/AMD/Nvidia doing with the other billions of transistors?
Execution in a Simple Processor

- **[IF]** Instruction fetch
- **[ID]** Instruction Decode
- **[EX]** Execution
- **[MEM]** Memory Read/Write
- **[WB]** Result Writeback

[Wikipedia ©]
Solution: Pipelining
MIPS Pipeline: 110,000 transistors

[Image of MIPS Pipeline Diagram]

[Wikipedia CC]
Q: Types of Pipeline Hazards? (aka: what can go wrong?)

- Data
- Structural
- Control

[Wikipedia ©]
Demo

Demo: intro/Pipeline Performance Mystery

- a, a: elapsed time 3.83603 s
- a, b: elapsed time 2.58667 s
- a, a unrolled: elapsed time 3.83673 s
- aa, bb unrolled: elapsed time 1.92509 s
- a, b unrolled: elapsed time 1.92084 s
A Glimpse of a More Modern Processor

[Sandy Bridge]

- Instruction Fetch Unit
- Branch Predictors
  - 144 Entry L1 I-Cache (8 way)
  - 32KB L1 I-Cache (8 way)
  - 16B Predecode, Fetch Buffer
  - 18+ Entry Instruction Queue
    - µcode Engine
    - Complex Decode
    - Simple Decode
    - Simple Decode
    - Simple Decode
  - 28 µop Decoder Queue
    - 168 Entry Reorder Buffer (ROB)
    - 144 Entry FP Physical Register File
    - 160 Entry Physical Register File
  - 54 Entry Unified Scheduler
    - Port 1
      - ALU LEA Shift
      - SIMD MUL Shift
    - Port 2
      - ALU LEA MUL
      - SIMD ALU MUL
    - Port 3
      - ALU Shift Branch
      - SIMD ALU Shuffle
    - Port 4
      - 64-bit AGU
      - 64-bit AGU
    - Port 5
      - 64 Entry Load Buffer
      - 36 Entry Store Buffer
  - 512 Entry L2 TLB (4 way)
  - 100 Entry L1 DTLB (fully)
  - 32KB L1 D-Cache (8 way)
  - 256KB L2 Cache (8 way)

[David Kanter / Realworldtech.com]
A Glimpse of a More Modern Processor: Frontend

Sandy Bridge

Instruction Fetch Unit

Branch Predictors

144 Entry L1 ITLB (4 way)

32KB L1 I-Cache (8 way)

16B Predecode, Fetch Buffer

6 instructions

18+ Entry Instruction Queue

μcode Engine

Complex Decode

Simple Decode

Simple Decode

Simple Decode

1.5K μop Cache (8 way)

28 μop Decoder Queue

4 μops

4 μops

1 μop

1 μop

1 μop

4 μops

32 Bytes

[David Kanter / Realworldtech.com]
A Glimpse of a More Modern Processor: Backend

- **New concept:** Instruction-level parallelism ("ILP", "superscalar")
- **Where does the IPC number from earlier come from?**

[David Kanter / Realworldtech.com]
Demo: intro/More Pipeline Mysteries
Q: Potential issues?

- $n \times$ the cache demand!
- Power?
- Some people just turn it off and manage their own ILP.
SMT/“Hyperthreading”

Q: Potential issues?

▶ $n \times$ the cache demand!
▶ Power?
▶ Some people just turn it off and manage their own ILP.
Outline

Introduction

Notes
About This Class
Why Bother with Parallel Computers?
Lowest Accessible Abstraction: Assembly
Architecture of an Execution Pipeline
Architecture of a Memory System
Shared-Memory Multiprocessors

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation
## More bad news from Dennard

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimension</td>
<td>$1/\kappa$</td>
</tr>
<tr>
<td>Line Resistance</td>
<td>$\kappa$</td>
</tr>
<tr>
<td>Voltage drop</td>
<td>$\kappa$</td>
</tr>
<tr>
<td>Response time</td>
<td>1</td>
</tr>
<tr>
<td>Current density</td>
<td>$\kappa$</td>
</tr>
</tbody>
</table>

[Dennard et al. ‘74, via Bohr ‘07]

- The above scaling law is for on-chip interconnects.
- Current $\sim$ Power vs. response time

Getting information from

- processor to memory
- one computer to the next

is

- slow (in latency)
- power-hungry
Performance characteristics of memory:

- Bandwidth
- Latency

*Flops are cheap*

*Bandwidth is money*

*Latency is physics*

- *M. Hoemmen*

Minor addition (but important for us)?

- Bandwidth is money *and code structure*
Latency is Physics: Distance
Latency is Physics: Electrical Model
Latency is Physics: DRAM
Latency is Physics: Performance Impact?

What is the performance impact of high memory latency?

Processor stalled, waiting for data.

Idea:

- Put a look-up table of recently-used data onto the chip.
- Cache
Memory Hierarchy

- Registers
- L1 Cache: 1 kB, 1 cycle
- L2 Cache: 10 kB, 10 cycles
- L3 Cache: 100 kB, 10 cycles
- DRAM: 1 MB, 100 cycles
- Virtual Memory (hard drive): 1 GB, 1000 cycles
- Virtual Memory (hard drive): 1 TB, 1 M cycles
A Basic Cache

Demands on cache implementation:

- Fast, small, cheap, low power
- Fine-grained
- High “hit”-rate (few “misses”)

![Diagram of Main Memory and Cache Memory with Index and Tag Data一览](image)

Design Goals: at odds with each other. Why?

Address matching logic expensive

[Wikipedia ©️]
Engineering Decisions:

- More data per unit of access matching logic → Larger “Cache Lines”
- Simpler/less access matching logic → Less than full “Associativity”
- Eviction strategy
- Size
Associativity

Direct Mapped:

<table>
<thead>
<tr>
<th>Memory</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

2-way set associative:

<table>
<thead>
<tr>
<th>Memory</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Miss rate versus cache size on the Integer portion of SPEC CPU2000 [Cantin, Hill 2003]
Demo: Learning about Caches

Demo: intro/Cache Organization on Your Machine
Experiments: 1. Strides: Setup

```c
int go(unsigned count, unsigned stride)
{
    const unsigned array_size = 64 * 1024 * 1024;
    int *ary = (int *) malloc(sizeof(int) * array_size);

    for (unsigned it = 0; it < count; ++it)
    {
        for (unsigned i = 0; i < array_size; i += stride)
            ary[i] *= 17;
    }

    int result = 0;
    for (unsigned i = 0; i < array_size; ++i)
        result += ary[i];

    free(ary);
    return result;
}
```

What do you expect? [Ostrovsky ‘10]
Experiments: 1. Strides: Results
Experiments: 2. Bandwidth: Setup

```c
int go(unsigned array_size, unsigned steps)
{
    int *ary = (int *) malloc(sizeof(int) * array_size);
    unsigned asm1 = array_size - 1;

    for (unsigned i = 0; i < 100*steps;)
    {
        #define ONE ary[(i++*16) & asm1] ++;
        #define FIVE ONE ONE ONE ONE ONE
        #define TEN FIVE FIVE
        #define FIFTY TEN TEN TEN TEN TEN
        #define HUNDRED FIFTY FIFTY
        HUNDRED
    }

    int result = 0;
    for (unsigned i = 0; i < array_size; ++i)
        result += ary[i];

    free(ary);
    return result;
}
```

What do you expect? [Ostrovsky ‘10]
Experiments: 2. Bandwidth: Results

![Graph showing the relationship between array size and effective bandwidth. The x-axis represents array size in bytes, ranging from $2^{12}$ to $2^{28}$. The y-axis represents effective bandwidth in GB/s, on a logarithmic scale ranging from $10^{-1}$ to $10^2$. The graph consists of two lines: a blue line and a green dashed line. The blue line shows a decrease in effective bandwidth as the array size increases, while the green dashed line remains relatively flat.](image-url)
Experiments: 3. A Mystery: Setup

```c
int go(unsigned array_size, unsigned stride, unsigned steps)
{
    char *ary = (char *) malloc(sizeof(int) * array_size);

    unsigned p = 0;
    for (unsigned i = 0; i < steps; ++i)
    {
        ary[p] ++;
        p += stride;
        if (p >= array_size)
            p = 0;
    }

    int result = 0;
    for (unsigned i = 0; i < array_size; ++i)
        result += ary[i];

    free(ary);
    return result;
}
```

What do you expect? [Ostrovsky ‘10]
Experiments: 3. A Mystery: Results

Color represents achieved bandwidth:
- Red: high
- Blue: low
Thinking about the Memory Hierarchy

- What is a working set?
- What is data locality of an algorithm?
- What does this have to with caches?
Case Study: Streaming Workloads

Q: Estimate expected throughput for saxpy on an architecture with caches. What are the right units?

\[ z_i = \alpha x_i + y_i \quad (i = 1, \ldots, n) \]

- **Units:** GBytes/s
- **Net memory accessed:** \( n \times 4 \times 3 \) bytes
- **Actual memory accessed:** \( n \times 4 \times 4 \) bytes
  (To read \( z \) read into the cache before modification)

Demo: [https://github.com/lcw/stream_ispc](https://github.com/lcw/stream_ispc)
Special Store Instructions

At least two aspects to keep apart:

- Temporal Locality: Are we likely to refer to this data again soon? (non-temporal store)
- Spatial Locality: Will (e.g.) the entire cache line be overwritten? (streaming store)

What hardware behavior might result from these aspects?

- Non-temporal: Write past cache entirely (/invalidate), or evict soon
- Spatial: Do not fetch cache line before overwriting

Comment on what a compiler can promise on these aspects.

- Might these 'flags' apply to loads/prefetches?

(see also: [McCalpin ‘18])
Case study: Matrix-Matrix Mult. (’MMM’): Code Structure

- How would you structure a high-performance MMM?
- What are sources of concurrency?
- What should you consider your working set?

**Sources of concurrency:**
- row, column loop,
- summation loop (?)

**Working set:** artificially created blocks

**Provide enough concurrency:** SIMD, ILP, Core

$A \cdot B$
Case study: Matrix-Matrix Mult. (’MMM’) via Latency

Come up with a simple cost model for MMM in a two-level hierarchy based on latency:

\[
\text{Avg latency per access} = (1 - \text{Miss ratio}) \cdot \text{Cache Latency} + \text{Miss ratio} \cdot \text{Mem Latency}
\]

Assume: Working sets fit in cache, No conflict misses

Calculation:
- Total accesses: \(4N_B^3\) (\(N_B\): block size)
- Misses: \(3N_B^2\)
- Miss rate: \(\frac{3}{4N_B \cdot \text{cache line size}}\)

[Yotov et al. ’07]
Case study: Matrix-Matrix Mult. (’MMM’) via Bandwidth

Come up with a cost model for MMM in a two-level hierarchy based on bandwidth:

- **FMA throughput**: $16 \times 2$ SP FMAs per clock (e.g.)
- **Cycle count**: $2N^3/(2 \cdot 32) = N^3/32$
- **Required cache bandwidth**: 
  
  \[
  \text{words accessed}/(\text{cycles}) = 4N^3/(N^3/32) = 128 \text{ floats/cycle (GB/s?)}
  \]
- **Total mem. data motion**: 
  
  \[
  \# \text{ blocks} \cdot 4 \cdot (\text{block size}) = (N/N_B)^3 \cdot 4N_B^2 = 4N^3/N_B
  \]
- **Required mem. bandwidth**: 
  
  \[
  \text{(Mem.motion)}/(\text{cycles}) = 4N^3/N_B/(N^3/32) = 128/N_B \text{ floats/cycle (GB/s?)}
  \]
- **What size cache do we need to get to feasible memory bandwidth?**

[Yotov et al. ’07]
Case study: Matrix-Matrix Mult. (’MMM’): Discussion

Discussion: What are the main simplifications in each model?

Bandwidth:
- Miss assumptions
- Multiple cache levels
- Latency effects

Latency:
- Miss assumptions
- Concurrency/parallelism of memory accesses
- (HW) prefetching
- Machine Limits

[Yotov et al. ’07]

General Q: How can we analyze cache cost of algorithms in general?
Hong/Kung: Red/Blue Pebble Game

Simple means of I/O cost analysis: “Red/blue pebble game”

▶ A way to quantify I/O cost on a DAG (why a DAG?)
▶ “Red Hot” pebbles: data that can be computed on
▶ “Blue Cool” pebbles: data that is stored, but not available for computation without I/O

Note: Can allow “Red/Purple/Blue/Black”: more levels

Q: What are the cost metrics in this model?

▶ I/O Cost: Turn a red into a blue pebble and vice versa
▶ Number of red pebbles (corresponding to size of ’near’ storage)

[Hong/Kung ‘81]
Cache-Oblivious Algorithms

Annoying chore: Have to pick multiple machine-adapted block sizes in cache-adapted algorithms, one for each level in the memory hierarchy, starting with registers.

Idea:

- Step 1: Express algorithm recursively in divide & conquer manner
- Step 2: Pick a strategy to decrease block size

Give examples of block size strategies, e.g. for MMM:

- All dimensions
- Largest dimension

Result:

- Asymptotically optimal on Hong/Kung metric
What are potential issues on actual hardware?

- In pure form:
  - Function call overhead
  - Register allocation

- With good base case:
  - I-cache overflow
  - Instruction scheduling

[Yotov et al. ’07]
Recall: Big-O Notation

Classical Analysis of Algorithms (e.g.):

\[ \text{Cost}(n) = O(n^3). \]

Precise meaning? Anything missing from that statement?

**Missing:** ‘as \( n \to \infty \)’

There exists a \( C \) and an \( N_0 \) independent of \( n \) so that for all \( n \geq N_0 \),

\[ \text{Cost}(n) \leq C \cdot n^3. \]
Comment: “Asymptotically Optimal”

Comments on asymptotic statements about cost in relation to high performance?

- No statement about finite $n$
- No statement about the constant

Net effect: Having an understanding of asymptotic cost is necessary, but not sufficient for high performance.

HPC is in the business of minimizing $C$ in:

$$\text{Cost}(n) \leq C \cdot n^3 \quad \text{(for all } n)$$
Alignment describes the process of matching the base address of:

- Single word: double, float
- SIMD vector
- Larger structure

To machine granularities:

- Natural word size
- Vector size
- Cache line

Q: What is the performance impact of misalignment?
Performance Impact of Misalignment

Matched structure

Matched structure
SIMD: Basic Idea

What’s the basic idea behind SIMD?

What architectural need does it satisfy?

- Insufficient instruction decode/dispatch bandwidth
- Tack more operations onto one decoded instruction

Typically characterized by width of data path:

- SSE: 128 bit (4 floats, 2 doubles)
- AVX-2: 256 bit (8 floats, 4 doubles)
- AVX-512: 512 bit (16 floats, 8 doubles)
SIMD: Architectural Issues

Realization of inter-lane comm. in SIMD? Find instructions.

▶ Misaligned stores.loads? (no)
▶ Broadcast, Unpack+Interleave, Shuffle, Permute
▶ Reductions (“horizontal”)

Name tricky/slow aspects in terms of expressing SIMD:

▶ Divergent control flow
  ▶ Masking
  ▶ Reconvergence
▶ Indirect addressing: gather/scatter

x86 SIMD suffixes: What does the “ps” suffix mean? “sd”?

▶ ps: Packed single precision
▶ sd: Scalar double precision
Why are transposes important? Where do they occur?

- Whenever SIMD encounters a mismatched data layout
- For example: MMM of two row-major matrices

Example implementation aspects:
- HPTT: [Springer et al. ‘17]
- github: springer13/hptt 8x8 transpose microkernel
- Q: Why 8x8?
Outline

Introduction

Notes
About This Class
Why Bother with Parallel Computers?
Lowest Accessible Abstraction: Assembly
Architecture of an Execution Pipeline
Architecture of a Memory System
Shared-Memory Multiprocessors

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation
Multiple Cores vs Bandwidth

Assume (roughly right for Intel):

- memory latency of 100 ns
- peak DRAM bandwidth of 50 GB/s (per socket)

How many cache lines should be/are in flight at one time?

\[
100\text{ns} \cdot 50\text{GB/s} = 5000\text{bytes}
\]

- About 80 cache lines

Oops: Intel hardware can only handle about 10 pending requests per core at one time

\[
10 \cdot \frac{64}{100\text{ns}} \approx 6.4\text{GB/s}
\]

[McCalpin ‘18]
Toplogy and NUMA

[SuperMicro Inc. ‘15]

Demo: Show lstopo on porter, from hwloc.
Placement and Pinning

Who decides on what core my code runs? How?

- The OS scheduler: “Oh, hey, look! A free core!”
- You, explicitly, by pinning:
  - OMP_PLACES=cores
  - pthread_setaffinity_np()

Who decides on what NUMA node memory is allocated?

- malloc uses 'first touch'
- You can decide explicitly (through libnuma)

Demo: intro/NUMA and Bandwidths

What is the main expense in NUMA?

Latency (but it impacts bandwidth by way of queuing)
Cache Coherence

What is *cache coherence*?

- As soon as you make a copy of (cache) something, you risk inconsistency with the original
- A set of guarantees on how (and in what order) changes to memory become visible to other cores

How is cache coherence implemented?

- Snooping
- Protocols, operating on cache line states (e.g. “MESI”)

What are the performance impacts?

- Demo: intro/Threads vs Cache
- Demo: intro/Lock Contention
'Conventional' vs Atomic Memory Update

- **Read** → **Increment** → **Write**
  - **Interruptible!**

- **Read** → **Increment** → **Write**
  - **Protected**
Outline

Introduction

Machine Abstractions
  C
  OpenCL/CUDA
  Convergence, Differences in Machine Mapping
  Lower-Level Abstractions: SPIR-V, PTX

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation

Polyhedral Representation and Transformation
Outline

Introduction

Machine Abstractions
- C
- OpenCL/CUDA
- Convergence, Differences in Machine Mapping
- Lower-Level Abstractions: SPIR-V, PTX

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation

Polyhedral Representation and Transformation
Atomic Operations: Compare-and-Swap

```c
#include <stdatomic.h>
_Bool atomic_compare_exchange_strong(
    volatile A* obj,
    C* expected, C desired);
```

What does `volatile` mean?

Memory may change at any time, do not keep in register.

What does this do?

- Store `(*obj == *expected) ? desired : *obj` into `*obj`.
- Return `true` iff memory contents was as expected.

How might you use this to implement atomic FP multiplication?

`Read previous, perform operation, try CAS, maybe retry`
Memory Ordering

Why is Memory Ordering a Problem?

- Out-of-order CPUs reorder memory operations
- Compilers reorder memory operations

What are the different memory orders and what do they mean?

- Atomicity itself is unaffected
- Makes sure that 'and then' is meaningful

Types:

- Sequentially consistent: no reordering
- Acquire: later loads may not reorder across
- Release: earlier writes may not reorder across
- Relaxed: reordering OK
Example: A Semaphore With Atomics

```c
#include <stdatomic.h> // mo->memory_order, a->atomic
typedef struct { atomic_int v; } naive_sem_t;
void sem_down(naive_sem_t *s) {
    while (1) {
        while (a_load_explicit(&(s->v), mo_acquire) < 1)
            spinloop_body();
        int tmp=a_fetch_add_explicit(&(s->v), -1, mo_acq_rel);
        if (tmp >= 1)
            break; // we got the lock
        else // undo our attempt
            a_fetch_add_explicit(&(s->v), 1, mo_relaxed);
    }
}
void sem_up(naive_s_t *s) {
    a_fetch_add_explicit(&(s->v), 1, mo_release);
}
[Cordes ’16] — Hardware implementation: how?
```
C: What is ’order’?

C11 Committee Draft, December ’10, Sec. 5.1.2.3, “Program execution”:

(3) *Sequenced before* is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread, which induces a partial order among those evaluations. Given any two evaluations A and B, if A is sequenced before B, then the execution of A shall precede the execution of B. (Conversely, if A is sequenced before B, then B is sequenced after A.) If A is not sequenced before or after B, then A and B are unsequenced. Evaluations A and B are *indeterminately sequenced* when A is sequenced either before or after B, but it is unspecified which. The presence of a sequence point between the evaluation of expressions A and B implies that every value computation and side effect associated with A is sequenced before every value computation and side effect associated with B. (A summary of the *sequence points* is given in annex C.)

Q: Where is this definition used (in the standard document)?

In defining the semantics of atomic operations.
C: What is ’order’? (Encore)

C11 Draft, 5.1.2.4 “Multi-threaded executions and data races”:

- All modifications to a particular atomic object M occur in some particular total order, called the modification order of M.
- An evaluation A carries a dependency to an evaluation B if . . .
- An evaluation A is dependency-ordered before an evaluation B if . . .
- An evaluation A inter-thread happens before an evaluation B if . . .
- An evaluation A happens before an evaluation B if . . .

Why is this so subtle?

- Many common optimizations depend on the ability to reorder operations.
- Two options:
  1. Lose the ability to do those optimizations
  2. Specify precisely how much of the order should be externally observable
C: How Much Lying is OK?

C11 Committee Draft, December ‘10, Sec. 5.1.2.3, “Program execution”:

1. The semantic descriptions in this International Standard describe the behavior of an abstract machine in which issues of optimization are irrelevant.

2. Accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations are all side effects, which are changes in the state of the execution environment. […]
(4) In the abstract machine, all expressions are evaluated as specified by the semantics. An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced (including any caused by calling a function or accessing a volatile object).

(6) The least requirements on a conforming implementation are:

- Accesses to volatile objects are evaluated strictly according to the rules of the abstract machine.
- At program termination, all data written into files shall be identical to the result that execution of the program according to the abstract semantics would have produced.
- The input and output dynamics of interactive devices shall take place as specified in 7.21.3. The intent of these requirements is that unbuffered or line-buffered output appear as soon as possible, to ensure that prompting messages actually appear prior to a program waiting for input. This is the observable behavior of the program.
Arrays

Why are arrays the dominant data structure in high-performance code?

- Performance is mostly achieved with regular, structured code (e.g. SIMD, rectangular loops)
- Arrays are a natural fit for that type of code
- Abstractions of linear algebra map directly onto arrays

Any comments on C’s arrays?

- 1D arrays: fine, no surprises
- nD arrays: basically useless: sizes baked into types
  - Interestingly: Fortran is (incrementally) smarter
## Arrays vs Abstraction

Arrays-of-Structures or Structures-of-Arrays? What’s the difference? Give an example.

- **Example:** Array of XYZ coordinates:
  - XYZXYZXYZ...
  - XXX....YYY....ZZZ...
- Which of these will be suitable for SIMD? (e.g. computing a norm?)
- Structures-of-Arrays if at all possible – to expose regularity

Language aspects of the distinction? Salient example?

- **C struct** forces you into arrays-of-structures
  - AoS: more “conceptually sound”
  - SoA: better for performance
- Complex numbers
SIMD

Name language mechanisms for SIMD:

- Inline Assembly
- Intrinsics
- Vector Types
  
  ```c
  typedef int v4si __attribute__((vector_size (16)));
  ```
- `#pragma simd`
- Merging of scalar program instances (in hw/sw)

Contrast outer-loop vs inner-loop vectorization.

- Inner-loop: Inner-most loop vectorized
- Outer loop: Vectorize a whole kernel. Requires:
  - Changed memory layout
  - Must be able to express all control flow

Side q: Would you consider GPUs outer- or inner-loop-vectorizing?

[Franchetti/Püschel ‘08]
Alignment: How?

The old way:

```c
int __attribute__((aligned(8))) a_int;
```

Difference between these two?

```c
int __attribute__((aligned(8))) * ptr_t_1;
int *__attribute__((aligned(8))) ptr_t_2;
```

The 'new' way (C/C++11):

```c
struct alignas(64) somestruct_t { /* ... */ };
struct alignas(alignof(other_t))
    somestruct_t { /* ... */ };
struct
    alignas(
        std::hardware_destructive_interference_size)
    somestruct_t { /* ... */ };
```

What is constructive interference?
What is the concrete impact of the constructs on the previous slide?

- Compiler needs to *know* whether data is aligned
  - Generate the correct instructions (which encode alignment promises)
  - Stack-allocate memory of the correct alignment
- Heap-allocated memory needs to actually satisfy the alignment promise!
  - `posix_memalign`
  - Hack it by overallocating
  - In `numpy`: overallocate in bytes, get base address, offset, obtain view
Pointers and Aliasing

Demo: machabstr/Pointer Aliasing
Register Pressure

What if the register working set gets larger than the registers can hold? What is the performance impact?

▶ “Register Spill”: save/reload code being generated
▶ CPU: L1 is relatively fast
▶ Other architectures: can be quite dramatic

Demo: machabstr/Register Pressure
Object-oriented programming: The weapon of choice for encapsulation and separation of concerns!
Performance perspective on OOP?

- Fine-grain OOP leads to an AoS disaster
- Operator overloading
- Copy-out/Rvalue references
- Run-time polymorphism (virtual methods) lead to fine-grain flow control

**Summary:** No good, very bad. *Must* have sufficient granularity to offset cost.
Some rules of thumb:

- Use indices rather than pointers
- Extract common subexpressions
- Make functions static
- Use `const`
- Avoid store-to-load dependencies

What are the concrete impacts of doing these things?
Outline

Introduction

Machine Abstractions
  C
  OpenCL/CUDA
    Convergence, Differences in Machine Mapping
    Lower-Level Abstractions: SPIR-V, PTX

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation

Polyhedral Representation and Transformation
65 nm, 4 SP ops at a time, 1 MiB L2.
"CPU-style" Cores

[Fatahalian ‘08]
Slimming down

Idea #1:
Remove components that help a single instruction stream run fast

[Fatahalian ‘08]
More Space: Double the Number of Cores

[Fatahalian ‘08]
Even more

[Refahalian ‘08]
Idea #2: SIMD

Amortize cost/complexity of managing an instruction stream across many ALUs

[Fatahalian ‘08]
Idea #2: SIMD

Amortize cost/complexity of managing an instruction stream across many ALUs

[Fatahalian ‘08]
Idea #2: SIMD

Amortize cost/complexity of managing an instruction stream across many ALUs

[Fatahalian ‘08]
Idea #2: SIMD

Amortize cost/complexity of managing an instruction stream across many ALUs

[Fatahalian '08]
GPU Abstraction: Core Beliefs

- View core count as an implementation detail
- View SIMD lane count as an implementation detail
- Program as if there are infinitely many of them
- Hardware division is expensive
  Make $n$D grids part of the model to avoid it
- Design the model to expose *extremely* fine-grain concurrency
  (e.g. between loop iterations!)
Wrangling the Grid

- get_local_id(axis) / size(axis)
- get_group_id(axis) / num_groups(axis)
- get_global_id(axis) / size(axis)

axis=0,1,2,...
‘SIMT’ and Branches

But what about branches?

\[
\begin{align*}
\text{ALU 1} & \quad \text{ALU 2} & \quad \ldots & \quad \ldots & \quad \text{ALU 8} \\
T & \quad T & \quad F & \quad T & \quad F & \quad F & \quad F & \quad F
\end{align*}
\]

Time (clocks)

\[
\begin{align*}
\text{if (} x > 0 \text{)} & \{
\text{\quad } y = \text{pow}(x, \exp); \\
\text{\quad } y *= \text{Ks}; \\
\text{\quad } \text{refl} = y + \text{Ka};
\} & \quad \text{else } \{
\text{\quad } x = 0; \\
\text{\quad } \text{refl} = \text{Ka};
\}
\]

\text{<unconditional shader code>}

\text{<resume unconditional shader code>}

[Fatahalian ‘08]
GPUs: Core Ideas

Three core ideas:

- Remove things that help with latency in single-thread
- Massive core and SIMD parallelism
- Cover latency with concurrency
  - SMT
  - ILP
How do these aspects show up in the model?

- View core count as an implementation detail
- View SIMD lane count as an implementation detail
- Program as if there are infinitely many of them
- Hardware division is expensive
  Make $n$D grids part of the model to avoid it
- Design the model to expose *extremely* fine-grain concurrency
  (e.g. between loop iterations!)
<table>
<thead>
<tr>
<th>Hardware</th>
<th>CL adjective</th>
<th>OpenCL</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMD lane</td>
<td>private</td>
<td>Work Item</td>
<td>Thread</td>
</tr>
<tr>
<td>SIMD Vector</td>
<td>—</td>
<td>Subgroup</td>
<td>Warp</td>
</tr>
<tr>
<td>Core</td>
<td>local</td>
<td>Workgroup</td>
<td>Thread Block</td>
</tr>
<tr>
<td>Processor</td>
<td>global</td>
<td>NDRRange</td>
<td>Grid</td>
</tr>
</tbody>
</table>
What forms of communication exist at each scope?

- Subgroup: Shuffles (!)
- Workgroup: Scratchpad, local atomics
- Grid: Global atomics

Can we just do locking like we might do on a CPU?

- Independent forward progress of all threads is not guaranteed: no.
- But: Device partitioning can help!
GPU Programming Model: Commentary

iales

“Vector” / “Warp” / “Wavefront”
  ▶ Important Hardware granularity
  ▶ Poorly/very implicitly represented

▶ What is the impact of reconvergence?
Outline

Introduction

Machine Abstractions
  C
  OpenCL/CUDA
  Convergence, Differences in Machine Mapping
  Lower-Level Abstractions: SPIR-V, PTX

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation

Polyhedral Representation and Transformation
What can we expect from future processor architectures?

- Commodity chips
- “Infinitely” many cores
- “Infinite” vector width
- Must hide memory latency (→ ILP, SMT)
- Compute bandwidth $\gg$ Memory bandwidth
- Bandwidth only achievable by homogeneity
- Can’t keep the whole thing powered all the time anyway. Consequence?

  Lots of weird stuff springs up. Examples: “Raytracing Cores”, “Tensor Cores”
Outline

Introduction

Machine Abstractions

C
OpenCL/CUDA
Convergence, Differences in Machine Mapping
Lower-Level Abstractions: SPIR-V, PTX

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation

Polyhedral Representation and Transformation
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation
  Forming Expectations of Performance
  Simple Performance Models: Rooflines
  Timing Experiments and Potential Issues
  Profiling and Observable Quantities
  Practical Tools: perf and toplevel (work-alike of Intel VTune)

Performance-Oriented Languages and Abstractions

Program Representation and Transformation

Polyhedral Representation and Transformation
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation
  Forming Expectations of Performance
  Simple Performance Models: Rooflines
  Timing Experiments and Potential Issues
  Profiling and Observable Quantities
  Practical Tools: perf and toplevel (work-alike of Intel VTune)

Performance-Oriented Languages and Abstractions

Program Representation and Transformation

Polyhedral Representation and Transformation
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation
  Forming Expectations of Performance
  Simple Performance Models: Rooflines
  Timing Experiments and Potential Issues
  Profiling and Observable Quantities
  Practical Tools: perf and toplevel (work-alike of Intel VTTune)

Performance-Oriented Languages and Abstractions

Program Representation and Transformation

Polyhedral Representation and Transformation
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation
  Forming Expectations of Performance
  Simple Performance Models: Rooflines
  Timing Experiments and Potential Issues
  Profiling and Observable Quantities
  Practical Tools: perf and toplev (work-alike of Intel VTune)

Performance-Oriented Languages and Abstractions

Program Representation and Transformation

Polyhedral Representation and Transformation
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation
  Forming Expectations of Performance
  Simple Performance Models: Rooflines
  Timing Experiments and Potential Issues
  Profiling and Observable Quantities
  Practical Tools: perf and toplev (work-alike of Intel VTune)

Performance-Oriented Languages and Abstractions

Program Representation and Transformation

Polyhedral Representation and Transformation
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation
   Forming Expectations of Performance
   Simple Performance Models: Rooflines
   Timing Experiments and Potential Issues
   Profiling and Observable Quantities
   Practical Tools: perf and toplev (work-alike of Intel VTune)

Performance-Oriented Languages and Abstractions

Program Representation and Transformation

Polyhedral Representation and Transformation
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions
- The Importance of Semantics: Defined and Undefined Behavior
- The Importance of Batches: kernels and traffic cops
- Functional and not
- Lazy and eager
- Embedded Languages
- Array and scalar: APL/numpy
- Tensorflow, Torch, and Tensor Programs
- Parallel Patterns
- NESL
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

The Importance of Semantics: Defined and Undefined Behavior

The Importance of Batches: kernels and traffic cops

Functional and not

Lazy and eager

Embedded Languages

Array and scalar: APL/numpy

Tensorflow, Torch, and Tensor Programs

Parallel Patterns

NESL
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions
  The Importance of Semantics: Defined and Undefined Behavior
  The Importance of Batches: kernels and traffic cops
  Functional and not
  Lazy and eager
  Embedded Languages
  Array and scalar: APL/numpy
  Tensorflow, Torch, and Tensor Programs
  Parallel Patterns
  NESL
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions
  The Importance of Semantics: Defined and Undefined Behavior
  The Importance of Batches: kernels and traffic cops
  Functional and not
  Lazy and eager
  Embedded Languages
  Array and scalar: APL/numpy
  Tensorflow, Torch, and Tensor Programs
  Parallel Patterns
  NESL
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions
- The Importance of Semantics: Defined and Undefined Behavior
- The Importance of Batches: kernels and traffic cops
- Functional and not
- Lazy and eager
- Embedded Languages
- Array and scalar: APL/numpy
- Tensorflow, Torch, and Tensor Programs
- Parallel Patterns
- NESL
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions
  The Importance of Semantics: Defined and Undefined Behavior
  The Importance of Batches: kernels and traffic cops
  Functional and not
  Lazy and eager

Embedded Languages
  Array and scalar: APL/numpy
  Tensorflow, Torch, and Tensor Programs

Parallel Patterns

NESL
Abusing the Type System
Expression templates
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions
   The Importance of Semantics: Defined and Undefined Behavior
   The Importance of Batches: kernels and traffic cops
   Functional and not
   Lazy and eager
   Embedded Languages
   Array and scalar: APL/numpy
   Tensorflow, Torch, and Tensor Programs
   Parallel Patterns
   NESL
Numpy as a data structure

Q: What is the cost of array.T in numpy?

\( O(1) \) in data volume: Only need to return a view with changed strides.
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions
  The Importance of Semantics: Defined and Undefined Behavior
  The Importance of Batches: kernels and traffic cops
  Functional and not
  Lazy and eager
  Embedded Languages
  Array and scalar: APL/numpy

Tensorflow, Torch, and Tensor Programs
Parallel Patterns
NESL
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions
  The Importance of Semantics: Defined and Undefined Behavior
  The Importance of Batches: kernels and traffic cops
  Functional and not
  Lazy and eager
  Embedded Languages
  Array and scalar: APL/numpy
  Tensorflow, Torch, and Tensor Programs

Parallel Patterns

NESL
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions
  The Importance of Semantics: Defined and Undefined Behavior
  The Importance of Batches: kernels and traffic cops
  Functional and not
  Lazy and eager
  Embedded Languages
  Array and scalar: APL/numpy
  Tensorflow, Torch, and Tensor Programs
  Parallel Patterns
  NESL
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation
  - Expression Trees
  - Metaprogramming
  - Data Flow Analysis
  - A Zoology of Compiler Passes and Ways to Control Them

Polyhedral Representation and Transformation
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation
  Expression Trees
  Metaprogramming
  Data Flow Analysis
  A Zoology of Compiler Passes and Ways to Control Them

Polyhedral Representation and Transformation
Term Rewriting
Macros: Textual and Syntactic, Hygiene
Common Subexpression Elimination
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation
  Expression Trees
  Metaprogramming
  Data Flow Analysis
  A Zoology of Compiler Passes and Ways to Control Them

Polyhedral Representation and Transformation
Textual
Macro Systems
Hygiene
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation
  - Expression Trees
  - Metaprogramming
  - Data Flow Analysis
  - A Zoology of Compiler Passes and Ways to Control Them

Polyhedral Representation and Transformation
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation
   Expression Trees
   Metaprogramming
   Data Flow Analysis
   A Zoology of Compiler Passes and Ways to Control Them

Polyhedral Representation and Transformation
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation

Polyhedral Representation and Transformation
  Presburger arithmetic, sets, and relations
  “Available Tools: Integer Linear Programming and Fourier-Motzkin”
  Loop Domains and Relations
  Polyhedral Schedule Models
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation

Polyhedral Representation and Transformation

- Presburger arithmetic, sets, and relations
- “Available Tools: Integer Linear Programming and Fourier-Motzkin”
- Loop Domains and Relations
- Polyhedral Schedule Models
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation

Polyhedral Representation and Transformation
  Presburger arithmetic, sets, and relations
  “Available Tools: Integer Linear Programming and Fourier-Motzkin”
  Loop Domains and Relations
  Polyhedral Schedule Models
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation

Polyhedral Representation and Transformation
  
  Presburger arithmetic, sets, and relations
  “Available Tools: Integer Linear Programming and Fourier-Motzkin”

Loop Domains and Relations

Polyhedral Schedule Models
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation

Polyhedral Representation and Transformation

Presburger arithmetic, sets, and relations

“Available Tools: Integer Linear Programming and Fourier-Motzkin”

Loop Domains and Relations

Polyhedral Schedule Models
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation

Polyhedral Representation and Transformation

Code Generation and Just-in-Time Compilation
  C as IR, LLVM IR
  Caching Strategies
  Constant Propagation and Run-time Typing
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation

Polyhedral Representation and Transformation

Code Generation and Just-in-Time Compilation

C as IR, LLVM IR

Caching Strategies

Constant Propagation and Run-time Typing
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation

Polyhedral Representation and Transformation

Code Generation and Just-in-Time Compilation

C as IR, LLVM IR

Caching Strategies

Constant Propagation and Run-time Typing
Outline

Introduction

Machine Abstractions

Performance: Expectation, Experiment, Observation

Performance-Oriented Languages and Abstractions

Program Representation and Transformation

Polyhedral Representation and Transformation

Code Generation and Just-in-Time Compilation
   C as IR, LLVM IR
   Caching Strategies
   Constant Propagation and Run-time Typing