GETTING THE MOST OUT OF RCPP

HIGH-PERFORMANCE C++ IN PRACTICE

Matt P. Dziubinski
R/Finance 2016

matt@math.aau.dk // @matt_dz
Department of Mathematical Sciences, Aalborg University
CREATES (Center for Research in Econometric Analysis of Time Series)
• Performance
  • Why do we care?
  • What is it?
  • How to
    • measure it - reason about it - improve it?
Why?

“Grove giveth and Gates taketh away.” – anon.

James Larus
larus@microsoft.com
Microsoft Research
<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem</td>
<td></td>
</tr>
<tr>
<td>Algorithm</td>
<td></td>
</tr>
<tr>
<td>Program</td>
<td></td>
</tr>
<tr>
<td>ISA (Instruction Set Arch)</td>
<td></td>
</tr>
<tr>
<td>Microarchitecture</td>
<td></td>
</tr>
<tr>
<td>Circuits</td>
<td></td>
</tr>
<tr>
<td>Electrons</td>
<td></td>
</tr>
</tbody>
</table>

Yale N. Patt, *Microprocessor Performance, Phase 2: Can We Harness the Transformation Hierarchy*

https://youtube.com/watch?v=0fLlDkC625Q
Until Recently (Phase I)

- Maintain the artificial walls between the layers
- Keeps the abstraction layers secure
  - Makes for a better comfort zone
- (Mostly) Improves the Microarchitecture
  - Pipelining, Caches
  - Branch Prediction, Speculative Execution
  - Out-of-order Execution, Trace Cache
- Today, we have too many transistors
40 Years of Microprocessor Trend Data

Transistors (thousands)
Single-Thread Performance (SpecINT x 10^3)
Frequency (MHz)
Typical Power (Watts)
Number of Logical Cores

https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/
the difference in the time between processor memory requests (for a single processor or core) and the latency of a DRAM access


Computer Architecture is Back: Parallel Computing Landscape
https://www.youtube.com/watch?v=On-k-E5HpcQ
Figure 1.1: DRAM Capacity & Latency Over Time [27, 111, 207, 232]


† We refer to the dominant DRAM chips during the period of time [27, 111].
Lee, Yunsup, ”Decoupled Vector-Fetch Architecture with a Scalarizing Compiler,” EECS Department, University of California, Berkeley. 2016. http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-82.html
What?
Origins of Matrix Methods

Jiuzhang Suanshu – is a Chinese manuscript dating from approximately 200 BC and containing 246 problems intended to illustrate methods of solution for everyday problems in areas such as engineering, surveying, and trade. (To the left, we see the opening of Chapter 1.) Chapter 8 of this ancient document details the first known example of matrix methods with a method known as fangcheng. The method of solution, described in detail, is what would become known centuries later as Gaussian elimination. The coefficients of the system are written as a table on a “counting board.” The text also compares two different fangcheng methods by counting the number of counting board operations needed in each method. [46, 7]

ON THE COMPUTATIONAL COMPLEXITY OF ALGORITHMS

BY

J. HARTMANIS AND R. E. STEARNS

I. Introduction. In his celebrated paper [1], A. M. Turing investigated the computability of sequences (functions) by mechanical procedures and showed that the set of sequences can be partitioned into computable and noncomputable sequences. One finds, however, that some computable sequences are very easy to compute whereas other computable sequences seem to have an inherent complexity that makes them difficult to compute. In this paper, we investigate a scheme of classifying sequences according to how hard they are to compute. This scheme puts a rich structure on the computable sequences and a variety of theorems are established. Furthermore, this scheme can be generalized to classify numbers, functions, or recognition problems according to their computational complexity.
The computational complexity of a sequence is to be measured by how fast a multitape Turing machine can print out the terms of the sequence. This particular abstract model of a computing device is chosen because much of the work in this area is stimulated by the rapidly growing importance of computation through the use of digital computers, and all digital computers in a slightly idealized form belong to the class of multitape Turing machines. More specifically, if $T(n)$ is a computable, monotone increasing function of positive integers into positive integers and if $\alpha$ is a (binary) sequence, then we say that $\alpha$ is in complexity class $S_T$ or that $\alpha$ is $T$-computable if and only if there is a multitape Turing machine $T$ such that $T$ computes the $n$th term of $\alpha$ within $T(n)$ operations.

**Complexity: Algorithms & Data Structures**

O(N)


*Complexity: At most last - first applications of the corresponding predicate.*

O(N \cdot \log(N))


*Complexity: \(\Theta(N \log(N))\) (where \(N = \text{last - first}\) comparisons.*


*Complexity: At most \(\log_2(\text{last - first}) + \Theta(1)\) comparisons.*

log(N)

http://en.cppreference.com/w/cpp/container/set/find

```
a.find(k) iterator;  // returns an iterator pointing to an element with the key
const_iterator iterator for equivalent to k, or a.end() if
constant a. such an element is not found
```
Scientific method

The very same approach that scientists use to understand the natural world is effective for studying the running time of programs:

- Observe some feature of the natural world, generally with precise measurements.
- Hypothesize a model that is consistent with the observations.
- Predict events using the hypothesis.
- Verify the predictions by making further observations.
- Validate by repeating until the hypothesis and observations agree.

**Definition.** We write $\sim f(N)$ to represent any function that, when divided by $f(N)$, approaches 1 as $N$ grows, and we write $g(N) \sim f(N)$ to indicate that $g(N)/f(N)$ approaches 1 as $N$ grows.

<table>
<thead>
<tr>
<th>function</th>
<th>tilde approximation</th>
<th>order of growth</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N^3/6 - N^2/2 + N/3$</td>
<td>$\sim N^3/6$</td>
<td>$N^3$</td>
</tr>
<tr>
<td>$N^2/2 - N/2$</td>
<td>$\sim N^2/2$</td>
<td>$N^2$</td>
</tr>
<tr>
<td>$\lg N + 1$</td>
<td>$\sim \lg N$</td>
<td>$\lg N$</td>
</tr>
<tr>
<td>3</td>
<td>$\sim 3$</td>
<td>1</td>
</tr>
</tbody>
</table>

**Typical tilde approximations**

Proposition C. (Doubling ratio)  If $T(N) \sim aN^b \lg N$ then $T(2N)/T(N) \sim 2^b$.

Proof: Immediate from the following calculation:

$$
T(2N)/T(N) = \frac{a(2N)^b \lg (2N)}{aN^b \lg N}
= 2^b \left(1 + \frac{\lg 2}{\lg N}\right)
\sim 2^b
$$

Rcpp Example - Pass & Sum Experiment - C++ Code I

// [[Rcpp::plugins(cpp11)]]
#include <algorithm>  // std::accumulate
#include <cstddef>  // std::size_t
#include <iterator>  // std::begin, std::end
#include <vector>  // std::vector
//@include <Rcpp.h>  // note: not w/ <RcppArmadillo.h>
//@include <RcppArmadillo.h>
//@include <RcppEigen.h>
//@include <RcppEigen.h>

//@include [[Rcpp::export]]

double sum_Rcpp_NumericVector_for(const Rcpp::NumericVector input) {
  double sum = 0.0;
  for (R_xlen_t i = 0; i != input.size(); ++i)
    sum += input[i];
  return sum;
}
// [[Rcpp::export]]
double sum_Rcpp_NumericVector_sugar(const Rcpp::NumericVector input) {
    return sum(input);
}

// [[Rcpp::export]]
double sum_std_vector_for(const std::vector<double> & input) {
    double sum = 0.0;
    for (std::size_t i = 0; i != input.size(); ++i)
        sum += input[i];
    return sum;
}

// [[Rcpp::export]]
double sum_std_vector_accumulate(const std::vector<double> & input) {
    return std::accumulate(begin(input), end(input), 0.0);
}
// [[Rcpp::export]]
double sum_Eigen_Vector(const Eigen::VectorXd & input) {
    return input.sum();
}

// [[Rcpp::export]]
double sum_Eigen_Map(const Eigen::Map<Eigen::VectorXd> input) {
    return input.sum();
}

// [[Rcpp::export]]
double sum_Eigen_Map_for(const Eigen::Map<Eigen::VectorXd> input) {
    double sum = 0.0;
    for (Eigen::DenseIndex i = 0; i != input.size(); ++i)
        sum += input(i);
    return sum;
}
// [[Rcpp::export]]
double sum_Arma_ColVec(const arma::colvec & input) {
    return sum(input);
}

// [[Rcpp::export]]
double sum_Arma_RowVec(const arma::rowvec & input) {
    return sum(input);
}

Note:

types with reference semantics - pass-by-value (\texttt{const})
types with value semantics - pass-by-reference (-to-\texttt{const})
size = 10000000L
v = rep_len(1.0, size)

library("rbenchmark")

benchmark(sum_Rcpp_NumericVector_for(v), sum_Rcpp_NumericVector_sugar(v),
          sum_std_vector_for(v), sum_std_vector_accumulate(v),
          sum_Eigen_Vector(v), sum_Eigen_Map(v), sum_Eigen_Map_for(v),
          sum_Arma_ColVec(v), sum_Arma_RowVec(v),
          order = "relative",
          columns = c("test", "replications", "elapsed", "relative",
                       "user.self", "sys.self"))
### Rcpp Example - PASS & SUM Experiment - Results

<table>
<thead>
<tr>
<th>test</th>
<th>replications</th>
<th>elapsed</th>
<th>relative</th>
<th>user.self</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum_Eigen_Map(v)</td>
<td>100</td>
<td>0.64</td>
<td>1.000</td>
<td>0.64</td>
</tr>
<tr>
<td>sum_Rcpp_NumericVector_sugar(v)</td>
<td>100</td>
<td>0.67</td>
<td>1.047</td>
<td>0.67</td>
</tr>
<tr>
<td>sum_Arma_RowVec(v)</td>
<td>100</td>
<td>0.67</td>
<td>1.047</td>
<td>0.67</td>
</tr>
<tr>
<td>sum_Arma_ColVec(v)</td>
<td>100</td>
<td>0.68</td>
<td>1.062</td>
<td>0.67</td>
</tr>
<tr>
<td>sum_Eigen_Map_for(v)</td>
<td>100</td>
<td>1.41</td>
<td>2.203</td>
<td>1.41</td>
</tr>
<tr>
<td>sum_std_vector_accumulate(v)</td>
<td>100</td>
<td>4.80</td>
<td>7.500</td>
<td>2.55</td>
</tr>
<tr>
<td>sum_std_vector_for(v)</td>
<td>100</td>
<td>4.82</td>
<td>7.531</td>
<td>2.49</td>
</tr>
<tr>
<td>sum_Eigen_Vector(v)</td>
<td>100</td>
<td>4.88</td>
<td>7.625</td>
<td>2.92</td>
</tr>
<tr>
<td>sum_Rcpp_NumericVector_for(v)</td>
<td>100</td>
<td>6.68</td>
<td>10.438</td>
<td>6.67</td>
</tr>
</tbody>
</table>

- sum_Rcpp_NumericVector_sugar faster than sum_Rcpp_NumericVector_for
- sum_std_vector_for, sum_std_vector_accumulate, and sum_Eigen_Vector slow
- sum_Eigen_Map_for slower than sum_Eigen_Map
**Rcpp Example - Pass & Sum Experiment - NumericVector**

`sum_Rcpp_NumericVector_sugar` faster than `sum_Rcpp_NumericVector_for`.

Why?

```cpp
// [[Rcpp::export]]
double sum_Rcpp_NumericVector_for(const Rcpp::NumericVector input) {
  double sum = 0.0;
  for (R_xlen_t i = 0; i != input.size(); ++i)
    sum += input[i];
  return sum;
}
```

Alternative `sum_Rcpp_NumericVector_for`:

```cpp
R_xlen_t n = object.size();
for (R_xlen_t i = 0; i < n; i++)
```

Alternative `sum_Rcpp_NumericVector_for`:

```cpp
for (R_xlen_t i = 0, n = input.size(); i != n; ++i)
```
sum\_Rcpp\_NumericVector\_sugar now on par w/
sum\_Rcpp\_NumericVector\_for

Next:

Why is std::vector slow - and sum\_Eigen\_Vector even slower than sum\_Eigen\_Map\_for?

Hypothesis: (Implicit) conversions — passing means copying — linear complexity.
Hypothesis: (Implicit) conversions — passing means copying — linear complexity.

How to check? Recall doubling ratio experiments!

C++:

```cpp
#include <vector>

// [[Rcpp::export]]
double pass_std_vector(const std::vector<double> & v) {
    return v.back();
}
```

R:

```r
pass_vector = pass_std_vector

v = seq(from = 0.0, to = 1.0, length.out = 10000000)
system.time(pass_vector(v))

v = seq(from = 0.0, to = 1.0, length.out = 20000000)
system.time(pass_vector(v))

v = seq(from = 0.0, to = 1.0, length.out = 40000000)
system.time(pass_vector(v))
```

Timing results: 0.04, 0.08, 0.17
C++:

```cpp
#include <Rcpp.h>

// [[Rcpp::export]]
double pass_Rcpp_vector(const Rcpp::NumericVector v) {
    return v[v.size() - 1];
}
```

R:

```r
pass_vector = pass_Rcpp_vector

v = seq(from = 0.0, to = 1.0, length.out = 10000000)
system.time(pass_vector(v))

v = seq(from = 0.0, to = 1.0, length.out = 20000000)
system.time(pass_vector(v))

v = seq(from = 0.0, to = 1.0, length.out = 40000000)
system.time(pass_vector(v))
```

Timing results: 0, 0, 0
Rcpp Example - Timing Experiments - std::vector

pass_vector = pass_std_vector

time_argpass = function(length) {
  show(length)
  v = seq(from = 0.0, to = 1.0, length.out = length)
  r = system.time(pass_vector(v))['elapsed']
  show(r)
  r
}

lengths = seq(from = 1000000, to = 40000000, length.out = 10)
times = sapply(lengths, time_argpass)

plot(log(lengths), log(times), type="l")

lm(log(times) ~ log(lengths))

Coefficients:
(Intercept) log(lengths)
-19.347 1.005
• implicit conversions - costly copies - worth avoiding
• analogous case: `std::string` vs. `boost::string_ref`
• C++17: `std::string_view`

http://en.cppreference.com/w/cpp/experimental/basic_string_view
http://en.cppreference.com/w/cpp/cpp/string/basic_string_view
"My tongue in cheek phrase to emphasize the importance of tradeoffs to the discipline of computer architecture. Clearly, computer architecture is more art than science. Science, we like to think, involves a coherent body of knowledge, even though we have yet to figure out all the connections. Art, on the other hand, is the result of individual expressions of the various artists. Since each computer architecture is the result of the individual(s) who specified it, there is no such completely coherent structure. So, I opined if computer architecture is a science at all, it is a science of tradeoffs. In class, we keep coming up with design choices that involve tradeoffs. In my view, "tradeoffs” is at the heart of computer architecture.” — Yale N. Patt
Software Performance Optimization - Analogous!

The multiplicity of tradeoffs:

- Multidimensional
- Multiple levels
- Costs and benefits
Trade-offs - Multidimensional - Numerical Optimization

\[
\begin{align*}
\text{minimize}_{x \in \mathbb{R}^d} \quad & f(x) = \sum_{j=1}^{N} f_j(x) \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time per iteration</th>
<th>Error after T iterations</th>
<th>Error after N items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Newton</td>
<td>(O(d^2N+d^3))</td>
<td>(C_I^{2T})</td>
<td>(C_I^2)</td>
</tr>
<tr>
<td>Gradient</td>
<td>(O(dN))</td>
<td>(C_G^T)</td>
<td>(C_G)</td>
</tr>
<tr>
<td>SGD</td>
<td>(O(d)) (or constant)</td>
<td>(\frac{C_S}{T})</td>
<td>(\frac{C_S}{N})</td>
</tr>
</tbody>
</table>

Ben Recht, Feng Niu, Christopher Ré, Stephen Wright. "Lock-Free Approaches to Parallelizing Stochastic Gradient Descent" OPT 2011: 4th International Workshop on Optimization for Machine Learning
Gradient computation - accuracy vs. function evaluations

\[ f : \mathbb{R}^d \rightarrow \mathbb{R}^N \]

- **Finite differencing:**
  - forward-difference: \( O(\sqrt{\epsilon_M}) \) error, \( d \ O(\text{Cost}(f)) \) evaluations
  - central-difference: \( O(\epsilon_M^{2/3}) \) error, \( 2d \ O(\text{Cost}(f)) \) evaluations
    
    with the *machine epsilon* \( \epsilon_M := \inf\{\epsilon > 0 : 1.0 + \epsilon \neq 1.0\} \)

- **Algorithmic differentiation (AD):** precision - as in hand-coded analytical gradient
  - rough forward-mode cost \( d \ O(\text{Cost}(f)) \)
  - rough reverse-mode cost \( N \ O(\text{Cost}(f)) \)
<table>
<thead>
<tr>
<th>LS Algorithm</th>
<th>Flop Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal equations</td>
<td>$mn^2 + n^3/3$</td>
</tr>
<tr>
<td>Householder QR</td>
<td>$n^3/3$</td>
</tr>
<tr>
<td>Modified Gram-Schmidt</td>
<td>$2mn^2$</td>
</tr>
<tr>
<td>Givens QR</td>
<td>$3mn^2 - n^3$</td>
</tr>
<tr>
<td>Householder Bidiagonalization</td>
<td>$4mn^2 - 2n^3$</td>
</tr>
<tr>
<td>R-Bidiagonalization</td>
<td>$2mn^2 + 2n^3$</td>
</tr>
<tr>
<td>SVD</td>
<td>$4mn^2 + 8n^3$</td>
</tr>
<tr>
<td>R-SVD</td>
<td>$2mn^2 + 11n^3$</td>
</tr>
</tbody>
</table>

Figure 5.5.1. Flops associated with various least squares methods

Golub & van Van Loan (2013) ”Matrix Computations”

Trade-offs: FLOPs (FLoating-point OPerations) vs. Applicability / Numerical Stability / Speed / Accuracy

Example: Catalogue of dense decompositions

http://eigen.tuxfamily.org/dox/group__TopicLinearAlgebraDecompositions.html
That point is that people work very hard to attain every microsecond of speed that a computer demonstrates, and there are two major problems facing an implementor when he embarks on producing a Lisp system: the first problem is the myriad of decisions to be made, the interactions of various parts of the Lisp system when they are brought together, the unfortunate choice in one aspect of the system turing around and influencing, badly, the performance of another; the second problem is that writing a Lisp system is a monumental undertaking, and this undertaking is executed within the context of living a life as well. And, although an implementor might start out with large goals and spectacular intentions, the time it takes to do the thorough job required to produce an excellent Lisp system will bring many obstacles and intrusions, impediments and obstructions, and in the end, Time will have won out, in that every microsecond the implementor grabs from the hands of Time are bought with hours or days or weeks or months of effort expended by the implementor.

Costs and Benefits: Implications

- Important to know what to focus on
- Optimize the optimization: so that it doesn’t always take hours or days or weeks or months...
How?
Asymptotic growth & "random access machines"?


Asymptotic - growing problem size - but for large data need to take into account the costs of actually bringing it in - communication complexity vs. computation complexity, including overlapping computation-communication latencies.
"Array Layouts for Comparison-Based Searching"
Paul-Virak Khuong, Pat Morin
http://cglab.ca/~morin/misc/arraylayout-v2/

• "With this understanding, we are able to choose layouts and design search algorithms that perform searches in $1/2$ to $2/3$ (depending on the array length) the time of the C++ `std::lower_bound()` implementation of binary search
"Array Layouts for Comparison-Based Searching”
Paul-Virak Khuong, Pat Morin
http://cglab.ca/~morin/misc/arraylayout-v2/

- "With this understanding, we are able to choose layouts and design search algorithms that perform searches in \( \frac{1}{2} \) to \( \frac{2}{3} \) (depending on the array length) the time of the C++ std::lower_bound() implementation of binary search
- (which itself performs searches in \( \frac{1}{3} \) the time of searching in the std::set implementation of red-black trees).
"Array Layouts for Comparison-Based Searching"
Paul-Virak Khuong, Pat Morin
http://cglab.ca/~morin/misc/arraylayout-v2/

• "With this understanding, we are able to choose layouts and design search algorithms that perform searches in $1/2$ to $2/3$ (depending on the array length) the time of the C++ std::lower_bound() implementation of binary search
• (which itself performs searches in $1/3$ the time of searching in the std::set implementation of red-black trees).
• It was only through careful and controlled experimentation with different implementations of each of the search algorithms that we are able to understand how the interactions between processor features such as pipelining, prefetching, speculative execution, and conditional moves affect the running times of the search algorithms.”
Reasoning about Performance: The Scientific Method

Requires - enabled by - the knowledge of microarchitectural details.

- Develop alternative hypotheses. Hypothesis 1: The program has synchronization bottlenecks. Hypothesis 2: The program is taking too many cache misses. Having multiple hypotheses gives parallelism to the following steps and helps keep us from getting too attached to one hypothesis.

- Develop one or more experiments that can exclude or corroborate an alternative hypothesis. Experiment 1: Add code in every critical section that stalls for time $T$ and counts how often it is executed. Can you develop experiments for Hypothesis 2?

- Predict experimental results before running the experiment. If Hypothesis 1 is true, a $P$-processor program that executes $S$ stalls should slow by much more than $T \times S/P$. If not, Hypothesis 1 is excluded.

- Run experiments. If the program runs only $T \times S/P$ slower, then Hypothesis 1 is excluded. We can continue with experiments for our alternatives or return to the first step and develop new hypotheses. If the program does run much more slowly, then Hypothesis 1 is corroborated. We should develop refined hypotheses (e.g., the synchronization bottleneck is for data structure A or B) and return to the first step.


Example: https://gist.github.com/MattPD/06e293fb935eaf67ee9c301e70db6975
Figure 2-1. CPU Core Pipeline Functionality of the Skylake Microarchitecture

Intel® 64 and IA-32 Architectures Optimization Reference Manual
pipeline-level parallelism (PLP)
instruction-level parallelism (ILP)
memory-level parallelism (MLP)
data-level parallelism (DLP)
thread-level parallelism (TLP)
#include <cstddef>
#include <cstdint>
#include <cstdlib>
#include <iostream>
#include <vector>
#include <boost/timer/timer.hpp>
typedef double T;

T sum_1(const std::vector<T> & input) {
    T sum = 0.0;
    for (std::size_t i = 0, n = input.size(); i != n; ++i)
        sum += input[i];
    return sum;
}

T sum_2(const std::vector<T> & input) {
    T sum1 = 0.0, sum2 = 0.0;
    for (std::size_t i = 0, n = input.size(); i != n; i += 2)
        { sum1 += input[i]; sum2 += input[i + 1]; }
    return sum1 + sum2;
}
```cpp
int main(int argc, char * argv[]) {
  std::size_t n = (argc >= 2) ? std::atoll(argv[1]) : 10000000;
  std::size_t f = (argc >= 3) ? std::atoll(argv[2]) : 1;
  std::cout << "n = " << n << '\n'; // iterations count
  std::cout << "f = " << f << '\n'; // unroll factor
  std::vector<T> a(n, T(1));
  boost::timer::auto_cpu_timer timer;
  T sum = (f == 1) ? sum_1(a)
    : (f == 2) ? sum_2(a)
    : 0;
  std::cout << sum << '\n';
}
```
make vector_sums CXXFLAGS="-std=c++14 -O2 -march=native"
LDLIBS=-lboost_timer

$ ./vector_sums 1000000000 2
n = 1000000000
f = 2
1e+09
0.466293s wall, 0.460000s user + 0.000000s system = 0.460000s CPU (98.7%)

$ ./vector_sums 1000000000 1
n = 1000000000
f = 1
1e+09
0.841269s wall, 0.840000s user + 0.010000s system = 0.850000s CPU (101.0%)
- https://perf.wiki.kernel.org/
- http://www.brendangregg.com/perf.html
### Performance counter stats for './vector_sums 1000000000 1':

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>task-clock (msec)</td>
<td>1675.812457</td>
<td># 0.850 CPUs utilized</td>
</tr>
<tr>
<td>context-switches</td>
<td>34</td>
<td># 0.020 K/sec</td>
</tr>
<tr>
<td>cpu-migrations</td>
<td>5</td>
<td># 0.003 K/sec</td>
</tr>
<tr>
<td>page-faults</td>
<td>8,953</td>
<td># 0.005 M/sec</td>
</tr>
<tr>
<td>cycles</td>
<td>5,760,418,457</td>
<td># 3.437 GHz</td>
</tr>
<tr>
<td>stalled-cycles-frontend</td>
<td>3,456,046,515</td>
<td># 60.00% frontend cycles idle</td>
</tr>
<tr>
<td>instructions</td>
<td>8,225,763,566</td>
<td># 1.43 insns per cycle</td>
</tr>
<tr>
<td>branches</td>
<td>2,050,710,005</td>
<td># 1223.711 M/sec</td>
</tr>
<tr>
<td>branch-misses</td>
<td>104,331</td>
<td># 0.01% of all branches</td>
</tr>
</tbody>
</table>

1.970909249 seconds time elapsed
Performance counter stats for './vector_sums 1000000000 2':

- 1283.910371 task-clock (msec) # 0.835 CPUs utilized
- 38 context-switches # 0.030 K/sec
- 3 cpu-migrations # 0.002 K/sec
- 9,466 page-faults # 0.007 M/sec
- 4,458,594,733 cycles # 3.473 GHz
- 2,149,690,303 stalled-cycles-frontend # 48.21% frontend cycles idle
- 6,734,925,029 instructions # 1.51 insns per cycle # 0.32 stalled cycles per
- 1,552,029,608 branches # 1208.830 M/sec # 0.01% of all branches
- 119,358 branch-misses

1.537971058 seconds time elapsed
GCC EXPLORER: sum_1

```cpp
#include <cstddef>
#include <cstdint>
#include <vector>

typedef double T;

T sum_1(const std::vector<T> & input)
{
    T sum = 0.0;
    for (std::size_t i = 0, n = input.size(); i != n; ++i)
        sum += input[i];
    return sum;
}
```

http://gcc.godbolt.org/

```
sum_1(std::vector<double, std::allocator<double> > const&):
    mov   rcx, QWORD PTR [rdi]
    mov   rdx, QWORD PTR [rdi+8]
    sub   rdx, rcx
    sar   rdx, 3
    je    .L4
    xor   eax, eax
    vxorpd xmm0, xmm0, xmm0
    .L3:
    vadddsd xmm0, xmm0, QWORD PTR [rcx+rax*8]
    add   rax, 1
    cmp   rax, rdx
    jne   .L3
    ret
    .L4:
    vxorpd xmm0, xmm0, xmm0
    ret
```
GCC EXPLORER: sum_2

http://gcc.godbolt.org/

```c
#include <cstring>
#include <vector>

typedef double T;

T sum_2(const std::vector<T> & input) {
    T sum1 = 0.0, sum2 = 0.0;
    for (std::size_t i = 0, n = input.size(); i != n; i += 2) {
        sum1 += input[i];
        sum2 += input[i + 1];
    }
    return sum1 + sum2;
}
```

```assembly
sum_2(std::vector<double, std::allocator<double> > & input):
    mov    rdx, QWORD PTR [rdi]
    mov    rcx, QWORD PTR [rdi+8]
    sар    rcx, 3
    je .L4
    vxorpd xmm1, xmm1, xmm1
    xor eax, eax
    vmovapd xmm0, xmm1
    .L3:
    vaddsd xmm0, xmm0, QWORD PTR [rdx+rax*8]
    vaddsd xmm1, xmm1, QWORD PTR [rdx+8+rax*8]
    add    rax, 2
    cmp    rax, rcx
    jne .L3
    vaddsd xmm0, xmm0, xmm1
    ret
    .L4:
    vxorpd xmm0, xmm0, xmm0
    ret
```
```cpp
#include <iacaMarks.h>

T sum_2(const std::vector<T> & input) {
    T sum1 = 0.0, sum2 = 0.0;
    for (std::size_t i = 0, n = input.size(); i != n; i += 2) {
        IACA_START
        sum1 += input[i];
        sum2 += input[i + 1];
        IACA_END
    }
    return sum1 + sum2;
}

$ g++ -std=c++14 -O2 -march=native vector_sums_2i.cpp -o vector_sums_2i
$ iaca -64 -arch IVB -graph ./vector_sums_2i

- https://stackoverflow.com/questions/26021337/what-is-iaca-and-how-do-i-use-it
- http://kylehegeman.com/blog/2013/12/28/introduction-to-iaca/
```
IACA RESULTS - SUM_1

$ iaca -64 -arch IVB -graph ./vector_sums_1i
Intel(R) Architecture Code Analyzer Version - 2.1
Analyzed File - ./vector_sums_1i
Binary Format - 64Bit
Architecture - IVB
Analysis Type - Throughput

Throughput Analysis Report

---
Block Throughput: 3.00 Cycles
Throughput Bottleneck: InterIteration

Port Binding In Cycles Per Iteration:

<table>
<thead>
<tr>
<th>Port</th>
<th>0 - DV</th>
<th>1</th>
<th>2 - D</th>
<th>3 - D</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles</td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

N - port number or number of cycles resource conflict caused delay, DV - Divider pipe (on port 0)
D - Data fetch pipe (on ports 2 and 3), CP - on a critical path
F - Macro Fusion with the previous instruction occurred
* - instruction micro-ops not bound to a port
^ - Macro Fusion happened
# - ESP Tracking sync uop was issued
@ - SSE instruction followed an AVX256 instruction, dozens of cycles penalty is expected
! - instruction not supported, was not accounted in Analysis

<table>
<thead>
<tr>
<th>Num Of</th>
<th>Ports pressure in cycles</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Uops</td>
<td>0 - DV</td>
<td>1</td>
<td>2 - D</td>
</tr>
<tr>
<td>--------</td>
<td>--------</td>
<td>---</td>
<td>-------</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>0F</td>
<td></td>
<td></td>
<td>1.0</td>
</tr>
</tbody>
</table>

Total Num Of Uops: 5

mov rdx, qword ptr [rdi]

|                    | CP | vaddsd xmm0, xmm0, qword ptr [rdx+rax*8]
|                    |    |
| add rax, 0x1       |
| cmp rax, rcx       |
| jnz 0xfffffffffffffff87e7 |
IACA RESULTS - SUM_2

$ iaca -64 -arch IVB -graph ./vector_sums_2i
Intel(R) Architecture Code Analyzer Version - 2.1
Analyzed File - ./vector_sums_2i
Binary Format - 64Bit
Architecture - IVB
Analysis Type - Throughput

Throughput Analysis Report

Block Throughput: 6.00 Cycles  Throughput Bottleneck: InterIteration

Port Binding In Cycles Per Iteration:

<table>
<thead>
<tr>
<th>Port</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles</td>
<td>1.5</td>
<td>0.0</td>
<td>3.0</td>
<td>1.5</td>
<td>1.5</td>
<td>1.5</td>
</tr>
</tbody>
</table>

N - port number or number of cycles resource conflict caused delay, DV - Divider pipe (on port 0)
D - Data fetch pipe (on ports 2 and 3), CP - on a critical path
F - Macro Fusion with the previous instruction occurred
* - instruction micro-ops not bound to a port
^ - Micro Fusion happened
# - ESP Tracking sync uop was issued
@ - SSE instruction followed an AVX256 instruction, dozens of cycles penalty is expected
! - instruction not supported, was not accounted in Analysis

<table>
<thead>
<tr>
<th>Num Of</th>
<th>Ports pressure in cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uops</td>
<td>0 - DV</td>
</tr>
</tbody>
</table>

| | 0.5 | 0.5 | 0.5 | 0.5 | mov rcx, qword ptr [rdi] |
| | 1.0 | 0.5 | 0.5 | 0.5 | add rcx, 0x2 |
| | 1.0 | 0.5 | 0.5 | 0.5 | vaddsd xmm1, xmm1, qword ptr [rcx+rdx+1] |
| 1 | 0.5 | 0.5 | add rdx, 0x10 |
| 1 | 1.0 | cmp rax, rsi |
| 0F | | jnz 0xfffffffffffffffffde |
| 1 | 1.0 | CP | vaddsd xmm0, xmm0, xmm1 |

Total Num Of Uops: 9
IACA Data Dependency Graph - sum_1

1. vaddsd xmm0, xmm0, qword ptr [rdx+rax*8]

0. mov rdx, qword ptr [rdi]

2. add rax, 0x1

3. cmp rax, rcx

4. jnz 0xfffffffffffffffe7
IACA Data Dependency Graph - sum_2

4. add rdx, 0x10

0. mov rcx, qword ptr [rdi]

2. add rax, 0x2

1. vaddsd xmm0, xmm0, qword ptr [rcx+rax*8]

5. cmp rax, rsi

7. vaddsd xmm0, xmm0, xmm1

6. jnz 0xfffffffffffffffde

3. vaddsd xmm1, xmm1, qword ptr [rcx+rdx*1]
Empty Issue Slots: Horizontal Waste & Vertical Waste

## Wasted Slots: Causes

<table>
<thead>
<tr>
<th>Source of Wasted Issue Slots</th>
<th>Possible Latency-Hiding or Latency-Reducing Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction tbl miss, data tbl miss</td>
<td>decrease the TLB miss rates (e.g., increase the TLB sizes); hardware instruction prefetching; hardware or software data prefetching; faster servicing of TLB misses</td>
</tr>
<tr>
<td>I cache miss</td>
<td>larger, more associative, or faster instruction cache hierarchy; hardware instruction prefetching</td>
</tr>
<tr>
<td>D cache miss</td>
<td>larger, more associative, or faster data cache hierarchy; hardware or software prefetching; improved instruction scheduling; more sophisticated dynamic execution</td>
</tr>
<tr>
<td>branch misprediction</td>
<td>improved branch prediction scheme; lower branch misprediction penalty</td>
</tr>
<tr>
<td>control hazard</td>
<td>speculative execution; more aggressive if-conversion</td>
</tr>
<tr>
<td>load delays (first-level cache hits)</td>
<td>shorter load latency; improved instruction scheduling; dynamic scheduling</td>
</tr>
<tr>
<td>short integer delay</td>
<td>improved instruction scheduling</td>
</tr>
<tr>
<td>long integer, short fp, long fp delays</td>
<td>(multiply is the only long integer operation, divide is the only long floating point operation) shorter latencies; improved instruction scheduling</td>
</tr>
<tr>
<td>memory conflict</td>
<td>(accesses to the same memory location in a single cycle) improved instruction scheduling</td>
</tr>
</tbody>
</table>

Table 3: All possible causes of wasted issue slots, and latency-hiding or latency-reducing techniques that can reduce the number of cycles wasted by each cause.

LIKWID

- https://github.com/RRZE-HPC/likwid
- https://github.com/RRZE-HPC/likwid/wiki
$ likwid-perfctr -C S0:0 -g FLOPS_DP -f ./vector_sums 1000000000 1

CPU name:  Intel(R) Core(TM) i7-3720QM CPU @ 2.60GHz  
CPU type:  Intel Core IvyBridge processor  
CPU clock:  2.59 GHz

n = 1000000000  
f = 1  
1e+09  
1.090122s wall, 0.880000s user + 0.000000s system = 0.880000s CPU (80.7%)

Group 1: FLOPS_DP

<table>
<thead>
<tr>
<th>Event</th>
<th>Counter</th>
<th>Core 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTR_RETIRED_ANY</td>
<td>FIXC0</td>
<td>8002493499</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_CORE</td>
<td>FIXC1</td>
<td>4285189526</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_REF</td>
<td>FIXC2</td>
<td>3258346806</td>
</tr>
<tr>
<td>FP_COMP_OPS_EXE_SSE_FP_PACKED_DOUBLE</td>
<td>PMC0</td>
<td>0</td>
</tr>
<tr>
<td>FP_COMP_OPS_EXE_SSE_FP_SCALAR_DOUBLE</td>
<td>PMC1</td>
<td>1000155741</td>
</tr>
<tr>
<td>SIMD_FP_256_PACKED_DOUBLE</td>
<td>PMC2</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Metric</th>
<th>Core 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime (RDTSC) [s]</td>
<td>2.0456</td>
</tr>
<tr>
<td>Runtime unhalted [s]</td>
<td>1.6536</td>
</tr>
<tr>
<td>Clock [MHz]</td>
<td>3408.2011</td>
</tr>
<tr>
<td>CPI</td>
<td>0.5355</td>
</tr>
<tr>
<td>MFLOP/s</td>
<td>488.9303</td>
</tr>
<tr>
<td>AVX MFLOP/s</td>
<td>488.9303</td>
</tr>
<tr>
<td>Packed MUOPS/s</td>
<td>0</td>
</tr>
<tr>
<td>Scalar MUOPS/s</td>
<td>488.9303</td>
</tr>
</tbody>
</table>
Likwid Results - sum_2: 595 Scalar MUOPS/s

```
$ likwid-perfctr -C S0:0 -g FLOPS_DP -f ./vector_sums 1000000000 2

CPU name:          Intel(R) Core(TM) i7-3720QM CPU @ 2.60GHz
CPU type:          Intel Core IvyBridge processor
CPU clock:         2.59 GHz

n = 1000000000
f = 2
1e+09
0.620421s wall, 0.470000s user + 0.000000s system = 0.470000s CPU (75.8%)

Group 1: FLOPS_DP
```

<table>
<thead>
<tr>
<th>Event</th>
<th>Counter</th>
<th>Core 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTR_RETIRED_ANY</td>
<td>FIXC0</td>
<td>6502566958</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_CORE</td>
<td>FIXC1</td>
<td>2948446599</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_REF</td>
<td>FIXC2</td>
<td>2223894218</td>
</tr>
<tr>
<td>FP_COMP_OPS_EXE_SSE_FP_PACKED_DOUBLE</td>
<td>PMC0</td>
<td>0</td>
</tr>
<tr>
<td>FP_COMP_OPS_EXE_SSE_FP_SCALAR_DOUBLE</td>
<td>PMC1</td>
<td>1000328727</td>
</tr>
<tr>
<td>SIMD_FP_256_PACKED_DOUBLE</td>
<td>PMC2</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Metric</th>
<th>Core 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime (RDTSC) [s]</td>
<td>1.6809</td>
</tr>
<tr>
<td>Runtime unhalted [s]</td>
<td>1.1377</td>
</tr>
<tr>
<td>Clock [MHz]</td>
<td>3435.8987</td>
</tr>
<tr>
<td>CPI</td>
<td>0.4534</td>
</tr>
<tr>
<td>MFLOP/s</td>
<td>595.1079</td>
</tr>
<tr>
<td>AVX MFLOP/s</td>
<td>0</td>
</tr>
<tr>
<td>Packed MUOPS/s</td>
<td>0</td>
</tr>
<tr>
<td>Scalar MUOPS/s</td>
<td>595.1079</td>
</tr>
</tbody>
</table>
**LIKwid Results: sum_vectorized: 676 AVX MFLOP/s**

```bash
$ likwid-perfctr -C S0:0 -g FLOPS_DP -f ./vector_sums_vf 1000000000 1
```

<table>
<thead>
<tr>
<th>Event</th>
<th>Counter</th>
<th>Core 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTR_RETIRED_ANY</td>
<td>FIXC0</td>
<td>3002491149</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_CORE</td>
<td>FIXC1</td>
<td>2709364345</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_REF</td>
<td>FIXC2</td>
<td>2043804906</td>
</tr>
<tr>
<td>FP_COMP_ops_exe_SSE_FP_PACKED_DOUBLE</td>
<td>PMC0</td>
<td>0</td>
</tr>
<tr>
<td>FP_COMP_ops_exe_SSE_FP_SCALAR_DOUBLE</td>
<td>PMC1</td>
<td>91</td>
</tr>
<tr>
<td>SIMD_FP_256_PACKED_DOUBLE</td>
<td>PMC2</td>
<td>260258099</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Metric</th>
<th>Core 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime (RDTSC) [s]</td>
<td>1.5390</td>
</tr>
<tr>
<td>Runtime unhalted [s]</td>
<td>1.0454</td>
</tr>
<tr>
<td>Clock [MHz]</td>
<td>3435.5297</td>
</tr>
<tr>
<td>CPI</td>
<td>0.9024</td>
</tr>
<tr>
<td>MFLOP/s</td>
<td>676.4420</td>
</tr>
<tr>
<td>AVX MFLOP/s</td>
<td>676.4420</td>
</tr>
<tr>
<td>Packed MUOPS/s</td>
<td>169.1105</td>
</tr>
<tr>
<td>Scalar MUOPS/s</td>
<td>0.0001</td>
</tr>
</tbody>
</table>
CPI = (Infinite-cache CPI) + finite-cache effect (FCE)

Infinite-cache CPI = execute busy (EBusy) + execute idle (EIdle)

FCE = (cycles per miss) × (misses per instruction) = (miss penalty) × (miss rate)
Branch Misprediction Penalty

Fig. 6. Interval behavior for a branch misprediction.

```cpp
#include <cmath>
#include <cassert>
#include <cstdlib>
#include <vector>
#include <boost/timer/timer.hpp>

double sum1(const std::vector<double> & x, const std::vector<bool> & which)
{
    double sum = 0.0;
    for (std::size_t i = 0, n = which.size(); i < n; ++i)
    {
        sum += (which[i]) ? std::cos(x[i]) : std::sin(x[i]);
    }
    return sum;
}
```
double sum2(const std::vector<double> & x, const std::vector<bool> & which)
{
    double sum = 0.0;
    for (std::size_t i = 0, n = which.size(); i < n; ++i)
    {
        sum += (which[i]) ? std::sin(x[i]) : std::cos(x[i]);
    }
    return sum;
}

std::vector<bool> inclusion_random(std::size_t n)
{
    std::vector<bool> which;
    which.reserve(n);
    std::random_device rd;
    static std::mt19937 g(rd());
    static std::uniform_int_distribution<int> u(1, 4);
    for (std::size_t i = 0; i < n; ++i)
which.push_back(u(g) >= 3);
    return which;
}

int main(int argc, char * argv[])
{
    const std::size_t n = (argc > 1) ? std::atoll(argv[1]) : 1000;
    std::cout << "n = " << n << '
';

    // branch takenness / predictability type
    // 0: never; 1: always; 2: random
    std::size_t type = (argc > 2) ? std::atoll(argv[2]) : 0;
    std::cout << "type = " << type << '
';

    std::vector<bool> which;
    if (type == 0) which.resize(n, false);
    else if (type == 1) which.resize(n, true);
    else if (type == 2) which = inclusion_random(n);
std::vector<double> x(n, 1.1);

boost::timer::auto_cpu_timer timer;
std::cout << sum1(x, which) + sum2(x, which) << '\n';
}
$ make BP CXXFLAGS="-std=c++14 -O3 -march=native" LDLIBS=-lboost_timer-mt

$ ./BP 10000000 0
n = 10000000
type = 0
1.3448e+007
 1.190391s wall, 1.187500s user + 0.000000s system = 1.187500s CPU (99.8%)

$ ./BP 10000000 1
n = 10000000
type = 1
1.3448e+007
 1.172734s wall, 1.156250s user + 0.000000s system = 1.156250s CPU (98.6%)

$ ./BP 10000000 2
n = 10000000
type = 2
1.3448e+007
 1.296455s wall, 1.296875s user + 0.000000s system = 1.296875s CPU (100.0%)
$ likwid-perfctr -C S0:1 -g BRANCH -f ./BP 10000000 0

CPU name: Intel(R) Core(TM) i7-3720QM CPU @ 2.60GHz
CPU type: Intel Core IvyBridge processor
CPU clock: 2.59 GHz

n = 10000000
Type = 0
1.3448e+07
0.445464s wall, 0.440000s user + 0.000000s system = 0.440000s CPU (98.8%)

Group 1: BRANCH

<table>
<thead>
<tr>
<th>Event</th>
<th>Counter</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTR_RETIRED_ANY</td>
<td>FIXC0</td>
<td>2495177597</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_CORE</td>
<td>FIXC1</td>
<td>1167613066</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_REF</td>
<td>FIXC2</td>
<td>1167632206</td>
</tr>
<tr>
<td>BR_INST_RETIRED_ALL_BRANCHES</td>
<td>PMC0</td>
<td>372952380</td>
</tr>
<tr>
<td>BR_MISP_RETIRED_ALL_BRANCHES</td>
<td>PMC1</td>
<td>14796</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Metric</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime (RDTSC) [s]</td>
<td>0.4586</td>
</tr>
<tr>
<td>Runtime unhalted [s]</td>
<td>0.4505</td>
</tr>
<tr>
<td>Clock [MHz]</td>
<td>2591.5373</td>
</tr>
<tr>
<td>CPI</td>
<td>0.4679</td>
</tr>
<tr>
<td>Branch rate</td>
<td>0.1495</td>
</tr>
<tr>
<td>Branch misprediction rate</td>
<td>5.929838e-06</td>
</tr>
<tr>
<td>Branch misprediction ratio</td>
<td>3.967263e-05</td>
</tr>
<tr>
<td>Instructions per branch</td>
<td>6.6903</td>
</tr>
</tbody>
</table>
Likwid: Branch (Mis)Prediction Example

$ likwid-perfctr -C S0:1 -g BRANCH -f ./BP 10000000 1

-----
CPU name: Intel(R) Core(TM) i7-3720QM CPU @ 2.60GHz
CPU type: Intel Core IvyBridge processor
CPU clock: 2.59 GHz
-----
n = 10000000
type = 1
1.3448e+07
0.445354s wall, 0.440000s user + 0.000000s system = 0.440000s CPU (98.8%)

Group 1: BRANCH

<table>
<thead>
<tr>
<th>Event</th>
<th>Counter</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTR_RETIRED_ANY</td>
<td>FIXC0</td>
<td>2495177490</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_CORE</td>
<td>FIXC1</td>
<td>1167125701</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_REF</td>
<td>FIXC2</td>
<td>1167146162</td>
</tr>
<tr>
<td>BR_INST_RETIRED_ALL_BRANCHES</td>
<td>PMC0</td>
<td>372952366</td>
</tr>
<tr>
<td>BR_MISP_RETIRED_ALL_BRANCHES</td>
<td>PMC1</td>
<td>14720</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Metric</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime (RDTSC) [s]</td>
<td>0.4584</td>
</tr>
<tr>
<td>Runtime unhalted [s]</td>
<td>0.4504</td>
</tr>
<tr>
<td>Clock [MHz]</td>
<td>2591.5345</td>
</tr>
<tr>
<td>CPI</td>
<td>0.4678</td>
</tr>
<tr>
<td>Branch rate</td>
<td>0.1495</td>
</tr>
<tr>
<td>Branch misprediction rate</td>
<td>5.899380e-06</td>
</tr>
<tr>
<td>Branch misprediction ratio</td>
<td>3.946885e-05</td>
</tr>
<tr>
<td>Instructions per branch</td>
<td>6.6903</td>
</tr>
</tbody>
</table>
**Likwid: Branch (Mis)Prediction Example**

```
$ likwid-perfctr -C S0:1 -g BRANCH -f ./BP 10000000 2
```

---

**CPU name:** Intel(R) Core(TM) i7-3720QM CPU @ 2.60GHz  
**CPU type:** Intel Core IvyBridge processor  
**CPU clock:** 2.59 GHz

---

\[ n = 10000000 \]
\[ \text{type} = 2 \]
\[ 1.3448e+07 \]

0.509917s wall, 0.510000s user + 0.000000s system = 0.510000s CPU (100.0%)

---

**Group 1: BRANCH**

<table>
<thead>
<tr>
<th>Event</th>
<th>Counter</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTR_RETIRED_ANY</td>
<td>FIXC0</td>
<td>3191479747</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_CORE</td>
<td>FIXC1</td>
<td>2264945099</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_REF</td>
<td>FIXC2</td>
<td>2264967068</td>
</tr>
<tr>
<td>BR_INST_RETIRED_ALL_BRANCHES</td>
<td>PMC0</td>
<td>468135649</td>
</tr>
<tr>
<td>BR_MISP_RETIRED_ALL_BRANCHES</td>
<td>PMC1</td>
<td>15326586</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Metric</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime (RDTSC) [s]</td>
<td>0.8822</td>
</tr>
<tr>
<td>Runtime unhalted [s]</td>
<td>0.8740</td>
</tr>
<tr>
<td>Clock [MHz]</td>
<td>2591.5589</td>
</tr>
<tr>
<td>CPI</td>
<td>0.7097</td>
</tr>
<tr>
<td>Branch rate</td>
<td>0.1467</td>
</tr>
<tr>
<td>Branch misprediction rate</td>
<td>0.0048</td>
</tr>
<tr>
<td>Branch misprediction ratio</td>
<td>0.0327</td>
</tr>
<tr>
<td>Instructions per branch</td>
<td>6.8174</td>
</tr>
</tbody>
</table>
Perf: Branch (Mis)Prediction Example

$ perf stat -e branches,branch-misses -r 10 ./BP 10000000 0
Performance counter stats for './BP 10000000 0' (10 runs):
374,121,213 branches
    ( +- 0.02% )
  23,260 branch-misses  # 0.01% of all branches ( +- 0.35% )
0.460392835 seconds time elapsed
    ( +- 0.50% )

$ perf stat -e branches,branch-misses -r 10 ./BP 10000000 1
Performance counter stats for './BP 10000000 1' (10 runs):
374,040,282 branches
    ( +- 0.01% )
  23,124 branch-misses  # 0.01% of all branches ( +- 0.45% )
0.457583418 seconds time elapsed
    ( +- 0.04% )

$ perf stat -e branches,branch-misses -r 10 ./BP 10000000 2
Performance counter stats for './BP 10000000 2' (10 runs):
469,331,762 branches
    ( +- 0.01% )
  15,326,501 branch-misses  # 3.27% of all branches ( +- 0.01% )
0.884858777 seconds time elapsed
    ( +- 0.30% )
The Sniper Multi-Core Simulator

http://snipersim.org/
SNIPER: BRANCH (MIS)PREDICTION EXAMPLE

CPI stack: never taken
CPI stack: always taken
CPI stack: randomly taken
Sniper: Branch (Mis)Prediction Example

CPI graph: never taken
CPI graph: always taken
CPI graph: randomly taken
CPI graph (detailed): always taken
CPI graph (detailed): randomly taken
http://www.gem5.org/
Filling with numbers - std::vector vs. std::list

Machine code & assembly (std::vector)

```
400d78: 48 c7 02 2a 00 00 00       mov    QWORD PTR [rdx],0x2a
400d7f: 48 83 c2 08             add    rdx,0x8
400d83: 48 39 ca               cmp    rdx,rcx
400d86: 75 f0                 jne    400d78 <main+0x18>
```

Micro-ops execution breakdown (std::vector)

```
4031000: system.cpu T0: @main+24.0: MOV M I: limm t1, 0x2a: IntAlu: D=0x0000000000000002
4031500: system.cpu T0: @main+24.1: MOV M I: st t1, DS:[rdx]: MemWrite: D=0x0000000000000002 A=0x6dad58
4032500: system.cpu T0: @main+31.0: ADD R I: limm t1, 0x8: IntAlu: D=0x0000000000000008
4033000: system.cpu T0: @main+31.1: ADD R I: add rdx, rdx, t1: IntAlu: D=0x0000000000000000
4033500: system.cpu T0: @main+35.0: CMP R R: sub t0, rdx, rcx: IntAlu: D=0x0000000000000001
4034000: system.cpu T0: @main+38.0: JNZ I: rdip t1, %ctrl153: IntAlu: D=0x00000000000040d8
4034500: system.cpu T0: @main+38.1: JNZ I: limm t2, 0xffffffffffffffff: IntAlu: D=0xffffffffffffffff
4035000: system.cpu T0: @main+38.2: JNZ I: wrmp t1, t2: IntAlu:
```

Assembly is Too High Level:
http://xlogicx.net/?p=369
Micro-ops pipeline stages (std::vector)
Pipeline diagram - one iteration (std::vector)

Pipeline diagram - three iterations (std::vector)
Machine code & assembly (std::list)

heap allocation in the loop @ 400d85
what could possibly go wrong?
STD::LIST - ONE ITERATION (...DONE!)
Memory Access Patterns: Temporal & Spatial Locality

Figure 2 Memory usage during separate compilations

horizontal axis - time; vertical axis - address

Loop Fusion

0.429504s (unfused) down to 0.287501s (fused)

g++ -Ofast -march=native (5.2.0)

```c
void unfused(double * a, double * b, double * c, double * d, size_t N)
{
    for (size_t i = 0; i != N; ++i)
        a[i] = b[i] * c[i];

    for (size_t i = 0; i != N; ++i)
        d[i] = a[i] * c[i];
}

void fused(double * a, double * b, double * c, double * d, size_t N)
{
    for (size_t i = 0; i != N; ++i)
    {
        a[i] = b[i] * c[i];
        d[i] = a[i] * c[i];
    }
}
```
MLP (memory-level parallelism) & STC (stall-time criticality)

Figure 3: Execution timeline for: (a) an application with few packets, which do not overlap with each other, hence the packet latency is completely exposed to the processor as stall cycles, making each packet critical to the processor. (b) an application with packets that overlap with each other, hence some of the packet latency is hidden from the processor, accruing fewer stall cycles per packet. The packets are thus less critical.

Overlapping latencies also works on a "macro" scale

- *load* as "get the data from the Internet"
- *compute* as "process the data"

Another example: Communication Avoiding and Overlapping for Numerical Linear Algebra

https://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-65.html

<table>
<thead>
<tr>
<th>id</th>
<th>symbol</th>
<th>count</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AAPL</td>
<td>565449</td>
<td>1.59043</td>
</tr>
<tr>
<td>2</td>
<td>AXP</td>
<td>731366</td>
<td>3.43745</td>
</tr>
<tr>
<td>3</td>
<td>BA</td>
<td>867366</td>
<td>5.40218</td>
</tr>
<tr>
<td>4</td>
<td>CAT</td>
<td>830327</td>
<td>7.08103</td>
</tr>
<tr>
<td>5</td>
<td>CSCO</td>
<td>400440</td>
<td>8.49192</td>
</tr>
<tr>
<td>6</td>
<td>CVX</td>
<td>687198</td>
<td>9.98761</td>
</tr>
<tr>
<td>7</td>
<td>DD</td>
<td>910932</td>
<td>12.2254</td>
</tr>
<tr>
<td>8</td>
<td>DIS</td>
<td>910430</td>
<td>14.058</td>
</tr>
<tr>
<td>9</td>
<td>GE</td>
<td>871676</td>
<td>15.8333</td>
</tr>
<tr>
<td>10</td>
<td>GS</td>
<td>280604</td>
<td>17.059</td>
</tr>
<tr>
<td>11</td>
<td>HD</td>
<td>556611</td>
<td>18.2738</td>
</tr>
<tr>
<td>12</td>
<td>IBM</td>
<td>860071</td>
<td>20.3876</td>
</tr>
<tr>
<td>13</td>
<td>INTC</td>
<td>559127</td>
<td>21.9856</td>
</tr>
<tr>
<td>14</td>
<td>JNJ</td>
<td>724724</td>
<td>25.5534</td>
</tr>
<tr>
<td>15</td>
<td>JPM</td>
<td>500473</td>
<td>26.576</td>
</tr>
<tr>
<td>16</td>
<td>KO</td>
<td>864903</td>
<td>28.5405</td>
</tr>
<tr>
<td>17</td>
<td>MCD</td>
<td>717021</td>
<td>30.087</td>
</tr>
<tr>
<td>18</td>
<td>MMM</td>
<td>698996</td>
<td>31.749</td>
</tr>
<tr>
<td>19</td>
<td>MRK</td>
<td>733948</td>
<td>33.2642</td>
</tr>
<tr>
<td>20</td>
<td>MSFT</td>
<td>475451</td>
<td>34.3134</td>
</tr>
<tr>
<td>21</td>
<td>NKE</td>
<td>556344</td>
<td>36.4545</td>
</tr>
</tbody>
</table>
### Overlapped Timings

<table>
<thead>
<tr>
<th>id</th>
<th>symbol</th>
<th>count</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AAPL</td>
<td>565449</td>
<td>2.00713</td>
</tr>
<tr>
<td>2</td>
<td>AXP</td>
<td>731366</td>
<td>2.09158</td>
</tr>
<tr>
<td>3</td>
<td>BA</td>
<td>867366</td>
<td>2.13468</td>
</tr>
<tr>
<td>4</td>
<td>CAT</td>
<td>830327</td>
<td>2.19194</td>
</tr>
<tr>
<td>5</td>
<td>CSCO</td>
<td>400440</td>
<td>2.19197</td>
</tr>
<tr>
<td>6</td>
<td>CVX</td>
<td>687198</td>
<td>2.19198</td>
</tr>
<tr>
<td>7</td>
<td>DD</td>
<td>910932</td>
<td>2.51895</td>
</tr>
<tr>
<td>8</td>
<td>DIS</td>
<td>910430</td>
<td>2.51898</td>
</tr>
<tr>
<td>9</td>
<td>GE</td>
<td>871676</td>
<td>2.51899</td>
</tr>
<tr>
<td>10</td>
<td>GS</td>
<td>280604</td>
<td>2.519</td>
</tr>
<tr>
<td>11</td>
<td>HD</td>
<td>556611</td>
<td>2.51901</td>
</tr>
<tr>
<td>12</td>
<td>IBM</td>
<td>860071</td>
<td>2.51902</td>
</tr>
<tr>
<td>13</td>
<td>INTC</td>
<td>559127</td>
<td>2.51902</td>
</tr>
<tr>
<td>14</td>
<td>JNJ</td>
<td>724724</td>
<td>2.51903</td>
</tr>
<tr>
<td>15</td>
<td>JPM</td>
<td>500473</td>
<td>2.51904</td>
</tr>
<tr>
<td>16</td>
<td>KO</td>
<td>864903</td>
<td>2.51905</td>
</tr>
<tr>
<td>17</td>
<td>MCD</td>
<td>717021</td>
<td>2.51906</td>
</tr>
<tr>
<td>18</td>
<td>MMM</td>
<td>698996</td>
<td>2.51907</td>
</tr>
<tr>
<td>19</td>
<td>MRK</td>
<td>733948</td>
<td>2.51908</td>
</tr>
<tr>
<td>20</td>
<td>MSFT</td>
<td>475451</td>
<td>2.51908</td>
</tr>
<tr>
<td>21</td>
<td>NKE</td>
<td>556344</td>
<td>2.51909</td>
</tr>
</tbody>
</table>
Visualizing & Monitoring Performance

https://github.com/Celtoys/Remotery

A real-time CPU/GPU profiler hosted in a single C file with a viewer that runs in a web browser.

Supported features:

- Lightweight instrumentation of multiple threads running on the CPU.
- Web viewer that runs in Chrome, Firefox and Safari. Custom WebSockets server transmits sample data to the browser on a latent thread.
- Profiles itself and shows how it's performing in the viewer.
- Can optionally sample CUDA/D3D11 GPU activity.
- Console output for logging text.
- Console input for sending commands to your game.
## Timeline: With Overlapping

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>6:01:19 AM</td>
<td>Connecting to ws://127.0.0.1:17815/rmt</td>
</tr>
<tr>
<td>6:01:23 AM</td>
<td>Connection Error</td>
</tr>
<tr>
<td>6:01:23 AM</td>
<td>Disconnected</td>
</tr>
<tr>
<td>6:01:25 AM</td>
<td>Connecting to ws://127.0.0.1:17815/rmt</td>
</tr>
<tr>
<td>6:01:27 AM</td>
<td>Connected</td>
</tr>
<tr>
<td>6:02:02 AM</td>
<td>Connection Error</td>
</tr>
<tr>
<td>6:02:02 AM</td>
<td>Disconnected</td>
</tr>
<tr>
<td>6:02:03 AM</td>
<td>Connecting to ws://127.0.0.1:17815/rmt</td>
</tr>
<tr>
<td>6:02:04 AM</td>
<td>Connected</td>
</tr>
<tr>
<td>6:02:12 AM</td>
<td>MSFT</td>
</tr>
<tr>
<td>6:02:12 AM</td>
<td>NKE</td>
</tr>
<tr>
<td>6:02:12 AM</td>
<td>PFE</td>
</tr>
<tr>
<td>6:02:12 AM</td>
<td>PG</td>
</tr>
<tr>
<td>6:02:12 AM</td>
<td>TRV</td>
</tr>
<tr>
<td>6:02:12 AM</td>
<td>UNH</td>
</tr>
<tr>
<td>6:02:12 AM</td>
<td>UTX</td>
</tr>
<tr>
<td>6:02:12 AM</td>
<td>V</td>
</tr>
<tr>
<td>6:02:12 AM</td>
<td>VZ</td>
</tr>
<tr>
<td>6:02:12 AM</td>
<td>WMT</td>
</tr>
<tr>
<td>6:02:12 AM</td>
<td>XOM</td>
</tr>
</tbody>
</table>
Cache Miss Penalty: Leading Edge & Trailing Edge

Miss Penalty = Leading Edge + Effects(Trailing Edge)

http://www.hpcaconf.org/hpca12/Phil_HPCA_06.pdf
How Are TE and BW Related?

Miss

Access

Trailing Edge

Time

TE = \frac{\text{Line Size}}{\text{Bus Width}} \times \frac{\text{Proc. Freq.}}{\text{Bus Freq.}}

Bus Utilization = \frac{\text{Trailing Edge}}{\text{Intermiss Distance}}


http://www.hpcacconf.org/hpca12/Phil_HPCA_06.pdf
Memory utilization even more important - contention for capacity & bandwidth!

Amdahl's Law Optimistic

Assumes *perfect parallelism* of the *parallel portion*: Only Serial Bottlenecks, No Parallel Bottlenecks

Counterpoint:

Multicore: Synchronization, Actual Scaling

Figure 6. Example for analyzing impact of critical sections

Figure 11. Example for analyzing bandwidth limited systems

#include <cstdlib>
#include <future>
#include <iostream>
#include <random>
#include <vector>
#include <boost/timer/timer.hpp>

struct contract
{
    double K;
    double T;
    double P;
};

using element = contract;
using container = std::vector<element>;
double sum_if(const container & a, const container & b, const std::vector<std::size_t> & index) {
    double sum = 0.0;
    for (std::size_t i = 0, n = index.size(); i < n; ++i) {
        std::size_t j = index[i];
        if (a[j].K == b[j].K) sum += a[j].K;
    }
    return sum;
}

template <typename F>
double average(F f, std::size_t m) {
    double average = 0.0;
    for (std::size_t i = 0; i < m; ++i)
        average += f() / m;
    return average;
}
```cpp
std::vector<std::size_t> index_stream(std::size_t n) {
    std::vector<std::size_t> index;
    index.reserve(n);
    for (std::size_t i = 0; i < n; ++i)
        index.push_back(i);
    return index;
}

std::vector<std::size_t> index_random(std::size_t n) {
    std::vector<std::size_t> index;
    index.reserve(n);
    std::random_device rd;
    static std::mt19937 g(rd());
    std::uniform_int_distribution<std::size_t> u(0, n - 1);
    for (std::size_t i = 0; i < n; ++i)
        index.push_back(u(g));
    return index;
}
```
```cpp
int main(int argc, char * argv[]) {
    const std::size_t n = (argc >= 2) ? std::atoll(argv[1]) : 1000;
    const std::size_t m = (argc >= 3) ? std::atoll(argv[2]) : 10;
    std::cout << "n = " << n << 'n';
    std::cout << "m = " << m << 'n';
    const std::size_t threads_count = 4;

    // thread access locality type
    // 0: none (default); 1: stream; 2: random
    std::vector<std::size_t> thread_type(threads_count);
    for (std::size_t thread = 0; thread != threads_count; ++thread) {
        thread_type[thread] = (argc >= 4 + thread)
            ? std::atoll(argv[3 + thread])
            : 0;
        std::cout << "thread_type[" << thread << "] = " << thread_type[thread] << 'n';
    }
}
```
```
endl(std::cout);

std::vector<std::vector<std::size_t>> index(threads_count);
for (std::size_t thread = 0; thread != threads_count; ++thread)
{
    index[thread].resize(n);
    if (thread_type[thread] == 1) index[thread] = index_stream(n);
    else if (thread_type[thread] == 2) index[thread] = index_random(n);
}

const container v1(n, {1.0, 0.5, 3.0});
const container v2(n, {1.0, 2.0, 1.0});

const auto thread_work = [m, &v1, &v2](const auto & thread_index)
{
    const auto f = [&v1, &v2, &thread_index] {
        return sum_if(v1, v2, thread_index);
    };
    return average(f, m);
};
```
boost::timer::auto_cpu_timer timer;

std::vector<std::future<double>> results;
results.reserve(threads_count);
for (std::size_t thread = 0; thread != threads_count; ++thread)
{
    results.emplace_back(std::async(std::launch::async,
[thread, &thread_work, &index] { return thread_work(index[thread]); })
);
}
for (auto &&result : results) if (result.valid()) result.wait();
for (auto &&result : results) std::cout << result.get() << 'n';
Multicore & DRAM: AoS Timings

1 thread, sequential access

$ ./DRAM_CMP 10000000 10 1
n = 10000000
m = 10
thread_type[0] = 1

1e+007
0.395408s wall, 0.406250s user + 0.000000s system = 0.406250s CPU (102.7%)
1 thread, random access

$ ./DRAM_CMP 10000000 10 2
n = 10000000
m = 10
thread_type[0] = 2

1e+007
5.348314s wall, 5.343750s user + 0.000000s system = 5.343750s CPU (99.9%)
4 threads, sequential access

$ ./DRAM_CMP 10000000 10 1 1 1 1
n = 10000000
m = 10
thread_type[0] = 1
thread_type[1] = 1
thread_type[2] = 1
thread_type[3] = 1

1e+007
1e+007
1e+007
1e+007
1e+007

0.508894s wall, 2.000000s user + 0.000000s system = 2.000000s CPU (393.0%)
4 threads: 3 sequential access + 1 random access

$ ./DRAM_CMP 10000000 10 1 1 1 2
n = 10000000
m = 10
thread_type[0] = 1
thread_type[1] = 1
thread_type[2] = 1
thread_type[3] = 2

1e+007
1e+007
1e+007
1e+007
1e+007

5.666049s wall, 7.265625s user + 0.000000s system = 7.265625s CPU (128.2%)
Memory Access Patterns & Multicore: Interactions Matter

Inter-thread Interference

Sharing - Contention - Interference - Slowdown

Threads using a shared resource (like on-chip/off-chip interconnects and memory) contend for it, interfering with each other’s progress, resulting in slowdown (and thus negative returns to increased threads count).

#include <cstddef>
#include <cstdlib>
#include <future>
#include <iostream>
#include <random>
#include <vector>
#include <boost/timer/timer.hpp>

// SoA (structure-of-arrays)
struct data
{
    std::vector<double> K;
    std::vector<double> T;
    std::vector<double> P;
};
double sum_if(const data & a, const data & b, const std::vector<std::size_t> & index)
{
    double sum = 0.0;
    for (std::size_t i = 0, n = index.size(); i < n; ++i)
    {
        std::size_t j = index[i];
        if (a.K[j] == b.K[j]) sum += a.K[j];
    }
    return sum;
}

template <typename F>
double average(F f, std::size_t m)
{
    double average = 0.0;
    for (std::size_t i = 0; i < m; ++i)
    {
        average += f() / m;
    }
```cpp
return average;
}

std::vector<std::size_t> index_stream(std::size_t n) {
    std::vector<std::size_t> index;
    index.reserve(n);
    for (std::size_t i = 0; i < n; ++i)
        index.push_back(i);
    return index;
}

std::vector<std::size_t> index_random(std::size_t n) {
    std::vector<std::size_t> index;
    index.reserve(n);
    std::random_device rd;
    static std::mt19937 g(rd());
    std::uniform_int_distribution<std::size_t> u(0, n - 1);
    for (std::size_t i = 0; i < n; ++i)
        index.push_back(u(g));
    return index;
}
```
```cpp
for (std::size_t i = 0; i < n; ++i)
    index.push_back(u(g));
return index;
}

int main(int argc, char * argv[]) {
    const std::size_t n = (argc >= 2) ? std::atoll(argv[1]) : 1000;
    const std::size_t m = (argc >= 3) ? std::atoll(argv[2]) : 10;
    std::cout << "n = " << n << 'n';
    std::cout << "m = " << m << 'n';
    const std::size_t threads_count = 4;

    // thread access locality type
    // 0: none (default); 1: stream; 2: random
    std::vector<std::size_t> thread_type(threads_count);
    for (std::size_t thread = 0; thread != threads_count; ++thread) {
```
thread_type[thread] = (argc >= 4 + thread) ? std::atoll(argv[3 + thread]) : 0;
std::cout << "thread_type[" << thread << "] = " << thread_type[thread] << 
} endl(std::cout);

std::vector<std::vector<std::size_t>> index(threads_count);
for (std::size_t thread = 0; thread != threads_count; ++thread)
{
    index[thread].resize(n);
    if (thread_type[thread] == 1) index[thread] = index_stream(n);
    else if (thread_type[thread] == 2) index[thread] = index_random(n);
    //for (auto e : index[thread]) std::cout << e; endl(std::cout);
}

data v1;
v1.K.resize(n, 1.0);
v1.T.resize(n, 0.5);
v1.P.resize(n, 3.0);
```cpp
data v2;
v2.K.resize(n, 1.0);
v2.T.resize(n, 2.0);
v2.P.resize(n, 1.0);

const auto thread_work = [m, &v1, &v2](const auto & thread_index) {
    const auto f = [&v1, &v2, &thread_index] { return sum_if(v1, v2, thread_index); }
    return average(f, m);
};

boost::timer::auto_cpu_timer timer;

std::vector<std::future<double>> results;
results.reserve(threads_count);
for (std::size_t thread = 0; thread != threads_count; ++thread) {
    results.emplace_back(std::async(std::launch::async,
        [thread, &thread_work, &index] { return thread_work(index[thread]); });
```

for (auto && result : results) if (result.valid()) result.wait();
for (auto && result : results) std::cout << result.get() << 'n';
1 thread, sequential access

$ ./DRAM_CMP.SoA 10000000 10 1 1 1 1
n = 10000000
m = 10
thread_type[0] = 1

1e+007
0.211877s wall, 0.203125s user + 0.000000s system = 0.203125s CPU (95.9%)
$ ./DRAM_CMP.SoA 10000000 10 2
n = 10000000
m = 10
thread_type[0] = 2

1e+007
4.534646s wall, 4.546875s user + 0.000000s system = 4.546875s CPU (100.3%)
4 threads, sequential access

$ ./DRAM_CMP.SoA 10000000 10 1 1 1 1
n = 10000000
m = 10
thread_type[0] = 1
thread_type[1] = 1
thread_type[2] = 1
thread_type[3] = 1

1e+007
1e+007
1e+007
1e+007
1e+007
0.256391s wall, 1.031250s user + 0.000000s system = 1.031250s CPU (402.2%)
4 threads: 3 sequential access + 1 random access

$ ./DRAM_CMP.SoA 10000000 10 1 1 1 2
n = 10000000
m = 10
thread_type[0] = 1
thread_type[1] = 1
thread_type[2] = 1
thread_type[3] = 2

1e+007
1e+007
1e+007
1e+007
1e+007

4.581033s wall, 5.265625s user + 0.000000s system = 5.265625s CPU (114.9%)
Better Access Patterns

yield

Better Single-core Performance

but also

Reduced Interference

and thus

Better Multi-core Performance
Figure 4: Arithmetic Intensity [26].

Speedup: Synchronization and Connectivity Bottlenecks

\[ \text{Speedup}(f, n_c) = \frac{1}{1 - f + \frac{f}{n_c} + \frac{f_1(n_c)}{n_c} + f_2(n_c)} \]

\( f \): parallelizable fraction

\( f_1 \): connectivity intensity

\( f_2 \): synchronization intensity

speedup affected by sequential-to-parallel data synchronization and inter-core communication

Hierarchical cycle accounting

https://github.com/David-Levinthal/gooda
Roofline Model(s)

http://www.eecs.berkeley.edu/~waterman/papers/roofline.pdf
Roofline: an insightful visual performance model for multicore architectures Samuel Williams, Andrew Waterman and David Patterson Communications ACM 55(6): 121-130 (2012)

http://www.inesc-id.pt/ficheiros/publicacoes/9068.pdf

http://spiral.ece.cmu.edu:8080/pub-spiral/abstract.jsp?id=181
Takeaways

Principles

Data structures & data layout - fundamental part of design

CPUs & pervasive forms parallelism

• can support each other: PLP, ILP (MLP!), TLP, DLP

Balanced design vs. bottlenecks

Overlapping latencies

Sharing-contention-interference-slowdown

Yale Patt’s Phase 2: Break the layers:

• break through the hardware/software interface
• harness all levels of the transformation hierarchy
Yale N. Patt, Microprocessor Performance, Phase 2: Can We Harness the Transformation Hierarchy
https://youtube.com/watch?v=0fLlDkC625Q
The Answer: Break the Layers

- (We already have in limited cases)

Yale N. Patt, *Microprocessor Performance, Phase 2: Can We Harness the Transformation Hierarchy*

https://youtube.com/watch?v=0fLlDkC625Q
Yale N. Patt at Yale Patt 75 Visions of the Future Computer Architecture Workshop:

”Are you a software person or a hardware person?”
I’m a person
this pigeonholing has to go

We must break the layers

Abstractions are great
- AFTER you understand what’s being abstracted

Yale N. Patt, 2013 IEEE CS Harry H. Goode Award Recipient Interview — https://youtu.be/S7wXivUy-tk

Yale N. Patt at Yale Patt 75 Visions of the Future Computer Architecture Workshop — https://youtu.be/x4LH1cJcvxs
http://www.agner.org/optimize/
https://users.ece.cmu.edu/~omutlu/lecture-videos.html
https://github.com/MattPD/cpplinks/
Thank You!

Questions?