Performance versus Design

- Advanced Programming School 2014 -

Peter Steinbach

Max Planck Institute of Molecular Cell Biology and Genetics
Scientific Computing Facility

Part 1

Performance

Part 2

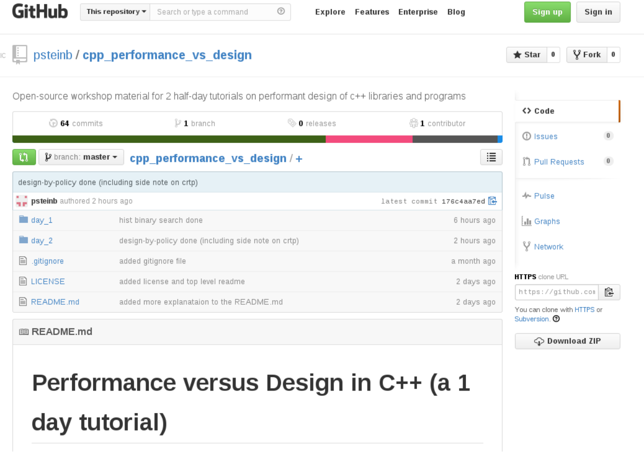
Good Code and Bad Code

Part 3

Exercises

Disclaimer

This teaching is open-source!



repository on github

Warning!

source: wikimedia commons

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97 % of the time: **premature optimization is the root of all evil**.

Donald E. Knuth, [1]

Warning!

source: wikimedia commons

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97 % of the time: **premature optimization is the root of all evil**. *Yet we should not pass up our opportunities in that critical 3 %.*

Donald E. Knuth, [1]

Warning!

source: homepage of Martin Grötschel

. . . observes that a benchmark production planning model solved using linear programming would have taken 82 years to solve in 1988, using the computers and the linear programming algorithms of the day. Fifteen years later - in 2003 - this same model could be solved in roughly 1 minute, an improvement by a factor of roughly 43 million. Of this, a factor of roughly 1,000 was due to increased processor speed, whereas a factor of roughly 43,000 was due to improvements in algorithms!

Martin Grötschel, cited in [2]

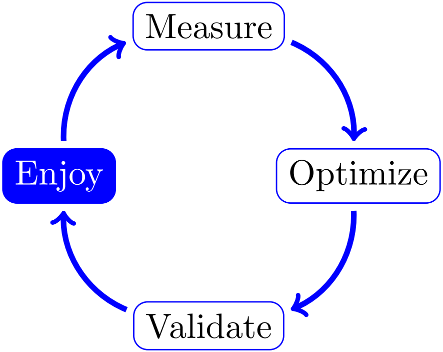
Warning!

source: homepage of Martin Grötschel

. . . observes that a benchmark production planning model solved using linear programming would have taken 82 years to solve in 1988, using the computers and the linear programming algorithms of the day. Fifteen years later - in 2003 - this same model could be solved in roughly 1 minute, an improvement by a factor of roughly 43 million. Of this, a factor of roughly 1,000 was due to increased processor speed, **whereas a factor of roughly 43,000 was due to improvements in algorithms**!

Martin Grötschel, cited in [2]

If and only if you have the feeling that your code runs slow for some reasons. . .



Adapted from the APOD Cycle [3]

Part I

Performance

1. A Definition

2. The resources used

3. Profiling

A Definition : Performance

Etymology

perform from Middle English *performen*, *parfournen* ("to perform"), from Anglo-Norman *performer*, *parfourmer*, alteration of Old French *parfornir*, *parfurnir* ("to complete, accomplish, perform"), from *par-* + *fornir*, *furnir* ("to accomplish, furnish"), ...

ance added to the stem of a verb to form a noun indicating a state or condition, such as result or capacity, associated with the verb.

(source: wiktionary)

A Definition : Performance

Etymology

perform from Middle English *performen*, *parfournen* ("to perform"), from Anglo-Norman *performer*, *parfourmer*, alteration of Old French *parfornir*, *parfurnir* ("to complete, accomplish, perform"), from *par-* + *fornir*, *furnir* ("to accomplish, furnish"), . . .

ance added to the stem of a verb to form a noun indicating a state or condition, such as result or capacity, associated with the verb.

(source: wiktionary)

Computer Performance

. . . is characterized by the amount of **useful work** accomplished by a computer system or computer network compared to the time and resources used.

(source: wikipedia)

A Definition : Performance

Etymology

perform: from Middle English *performen*, *parfournen* ("to perform"), from Anglo-Norman *performer*, *parfourmer*, alteration of Old French *parfornir*, *parfurnir* ("to complete, accomplish, perform"), from *par-* + *fornir*, *furnir* ("to accomplish, furnish"), ...

ance: added to the stem of a verb to form a noun indicating a state or condition, such as result or capacity, associated with the verb.

(source: wiktionary)

Computer Performance

...is characterized by the amount of **useful work** accomplished by a computer system or computer network compared to the **time** and resources used.

(source: wikipedia)

A Definition : Performance

Etymology

perform from Middle English *performen*, *parfournen* ("to perform"), from Anglo-Norman *performer*, *parfourmer*, alteration of Old French *parfornir*, *parfurnir* ("to complete, accomplish, perform"), from *par-* + *fornir*, *furnir* ("to accomplish, furnish"), ...

ance added to the stem of a verb to form a noun indicating a state or condition, such as result or capacity, associated with the verb.

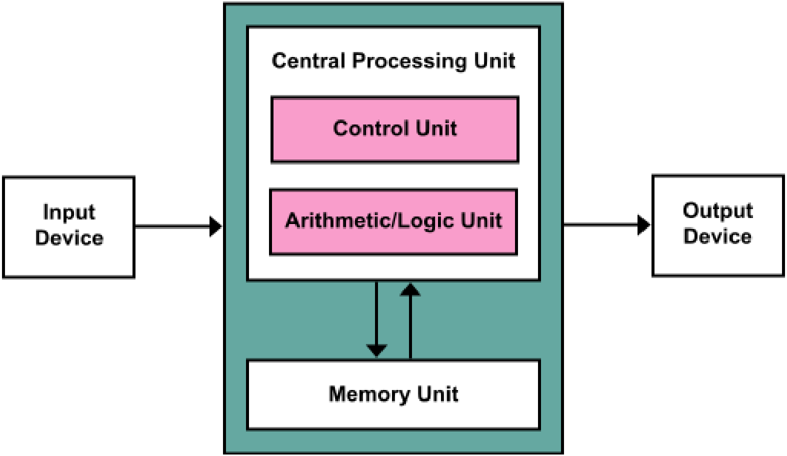
(source: wiktionary)

Computer Performance

...is characterized by the amount of **useful work** accomplished by a computer system or computer network compared to the **time** and **resources** used.

(source: wikipedia)

Resources: Computers?



Stored-Program Computer Architecture ([4], source: wikipedia)

A simple app

```
#include <iostream>

int main(int argc, char *argv[])
{
  int i = 42;
  std::cout << "i = " << i << "\n";
  return 0;
}
```

A simple app

```
#include <iostream>

int main(int argc, char *argv[])
{
  int i = 42;
  std::cout << "i = " << i << "\n";
  return 0;
}
```

g++ simple_app.cpp -o simple_app

```
00000000004007e0 <main>:
% ...
4007ef: c7 45 fc 2a 00 00 00  movl   $0x2a,-0x4(%rbp)
4007f6: be 10 09 40 00        mov    $0x400910,%esi
4007fb: bf 60 10 60 00        mov    $0x601060,%edi
400800: e8 db fe ff ff        callq  4006e0 <_ZStlsISt11char_traitsIcEERSt13basi
% ...
400825: c3                    retq
```

Before anything you want is executed on the CPU . . .

Developer

Writes code in high level language with some intentions!

Before anything you want is executed on the CPU . . .

Developer

Writes code in high level language with some intentions!

Compiler

Translates source into opcodes given good knowledge of the underlying CPU architecture, heuristic and analytic optimisations, . . .

Before anything you want is executed on the CPU . . .

Developer

Writes code in high level language with some intentions!

Compiler

Translates source into opcodes given good knowledge of the underlying CPU architecture, heuristic and analytic optimisations, . . .

Hardware

Deciphers opcodes to execute them (given clever hardware) on die

Resources: From Source To Instruction

Before anything you want is executed on the CPU . . .

Developer

Writes code in high level language with some intentions!

Compiler

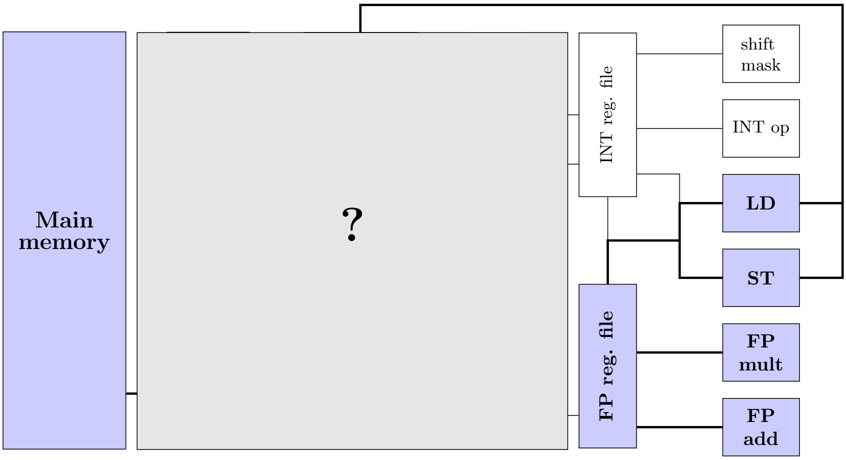
Translates source into opcodes given good knowledge of the underlying CPU architecture, heuristic and analytic optimisations, . . .

Hardware

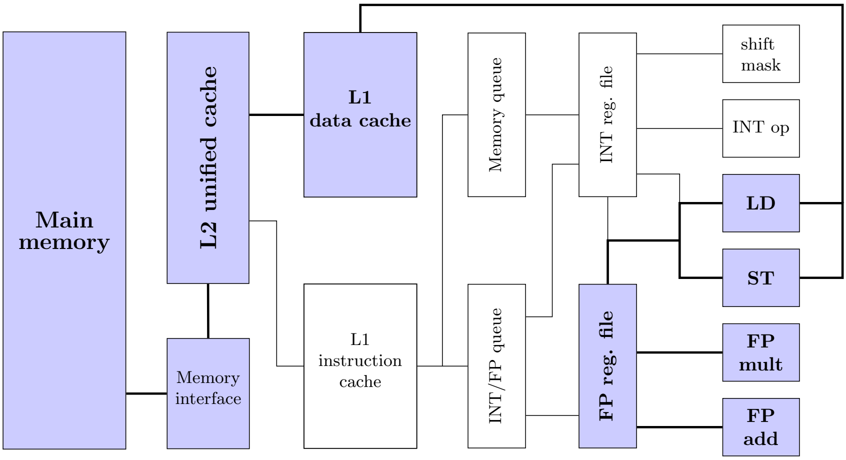
Deciphers opcodes to execute them (given clever hardware) on die

Let's have a look at the hardware!

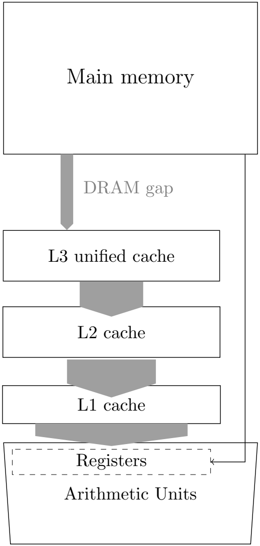
Resources: From Source To Instruction



Block diagram Cache-based microprocessor (adapted from [5], Fig. 1.2)



Block diagram Cache-based microprocessor (adapted from [5], Fig. 1.2)



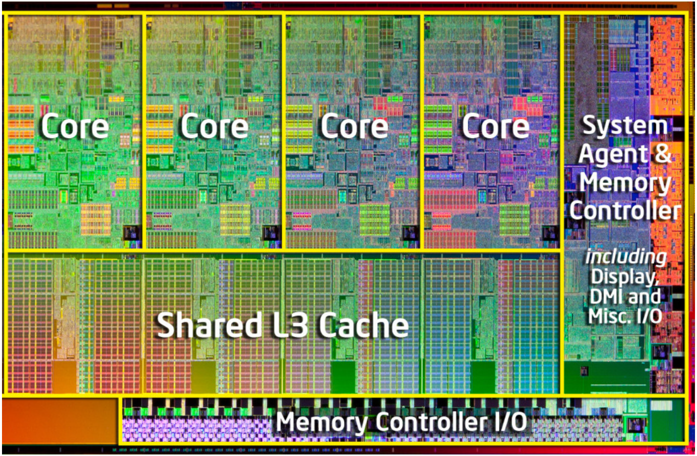
(adapted from [5], Fig. 1.3)

Memory hierarchy of a cache-based microprocessor

- direct access to RAM offers lowest bandwidth (slow)
- transfer bandwidth increases the closer the CPU is
- caches hide low RAM bandwidth by buffering hot data
- smallest transfer datum between caches: **cache line** $= 64\,\mathrm{B}$

Common Dimensions

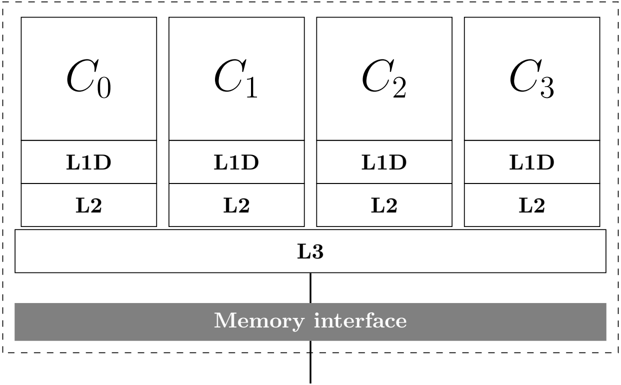
L3 unified cache	$4 - 15\,\mathrm{MB}$
L2 cache	$N_c \times 256\,\mathrm{KB}$
L1 cache	$N_c \times 2 \times 32\,\mathrm{KB}$



Intel® Sandy Bridge ® die map (from bit-tech.net)

Memory related parts of die make up $> 50\,\%$ of die area!

Resources: Wrapping up



CPUs are complicated beasts

- pipelined functional units
- superscaler arithmetic units
- out-of-order execution
- symmetric multi-threading
- turbo-boost
- ...

Resources: Exploring it

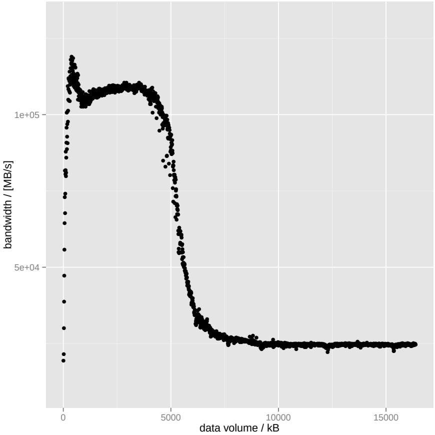
stream benchmark [6, 7]

- (one) standard candle of HPC benchmark(s)
- contains 4 functions that run on synthetic arrays
- get it at: cs.virginia.edu/stream/

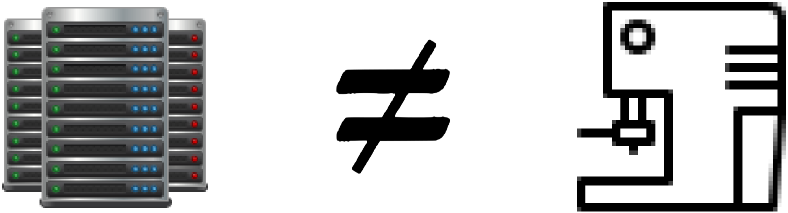
```
float a[MAX], b[MAX], c[MAX];

for(long i = 0;i<MAX;++i) {
  a[i] = b[i]; //Copy
  a[i] = d*b[i]; //Scale
  a[i] = b[i]+c[i]; //Add
  a[i] = b[i]+d*c[i]; //Triad
}
```

Resources: Exploring it

```
a[:]  = b[:]+d*c[:];
```



Triad results for different `float` array sizes on Intel® Xeon® E5-2630 12-core CPU

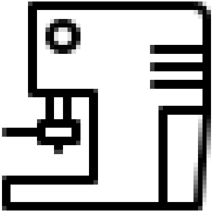


(source: iconfinder.com, openclipart.org)

- today's hardware composed of very advanced microprocessors (many years of engineering, moore's law, physics and material's science)

Resources: Summary

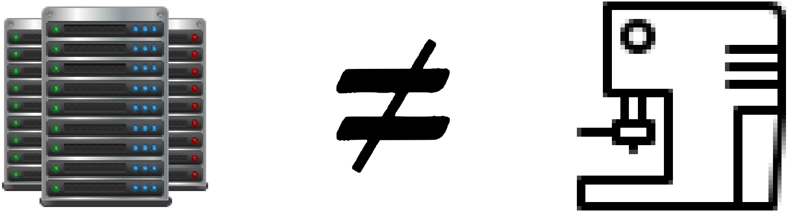
$\neq$



(source: iconfinder.com, openclipart.org)

- today's hardware composed of very advanced microprocessors (many years of engineering, moore's law, physics and material's science)
- depending on the goal of your software development practise, intimate knowledge of the hardware is vital, beneficial or an extra

Resources: Summary



(source: iconfinder.com, openclipart.org)

- today's hardware composed of very advanced microprocessors (many years of engineering, moore's law, physics and material's science)
- depending on the goal of your software development practise, intimate knowledge of the hardware is vital, beneficial or an extra
- to understand performance bottlenecks, knowing the hardware and the compiler concepts is crucial

In software engineering, **profiling** . . . is a form of dynamic program analysis that **measures**, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or frequency and duration of function calls. *The most common use of profiling information is to aid program optimization.*

(source: wikipedia)

Profiling: time et al

time

```
$ time sleep 10 #bash built-in

real    0m10.001s
user    0m0.000s
sys     0m0.000s
```

```
$ `which time` -p sleep 10 #app

real 10.00
user 0.00
sys 0.00
```

- simple and effective
- use the "user" time (or the wallclock time) as central measurement unit
- CPU time is an accumulated time (does not account for waits/sleeps)
- does apply for library based timers (`boost::cpu_time`, `std::chrono`, ...)
- use monitoring tools (`ps`, `top`, ...) to get a feeling of an application

Profiling: `valgrind`

- originally memory profiling/debugging tool
- now: tool suite and framework for dynamic program analysis
- open-source tool under GPL
- runs “instrumented” application inside a VM
- apps take $> 2 - 10\times$ longer inside `valgrind`
- applications should contain debugging symbols

Profiling: `valgrind`

- originally memory profiling/debugging tool
- now: tool suite and framework for dynamic program analysis
- open-source tool under GPL
- runs "instrumented" application inside a VM
- apps take $> 2 - 10\times$ longer inside `valgrind`
- applications should contain debugging symbols

parts concerning profiling

memcheck unallowed memory access, use of uninitialised values, memory leaks, bad `free`/`delete` calls

callgrind simulates L1i/d and L2 caches, records callgraph and logs memory access and instruction calls by line of source code (use kcachegrind for visiualisation)

massif heap profiling by taking regular snapshots of a program's heap (use massif-visualizer

Profiling: `valgrind`

- originally memory profiling/debugging tool
- now: tool suite and framework for dynamic program analysis
- open-source tool under GPL
- runs "instrumented" application inside a VM
- apps take $> 2 - 10\times$ longer inside `valgrind`
- applications should contain debugging symbols

parts concerning profiling

memcheck unallowed memory access, use of uninitialised values, memory leaks, bad `free`/`delete` calls

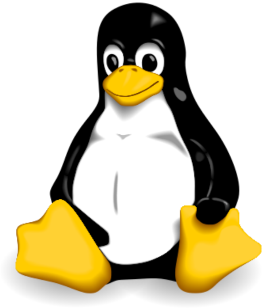
callgrind simulates L1i/d and L2 caches, records callgraph and logs memory access and instruction calls by line of source code (use kcachegrind for visiualisation)

massif heap profiling by taking regular snapshots of a program's heap (use massif-visualizer

Let's take a tour! (stream, ellbow-out)

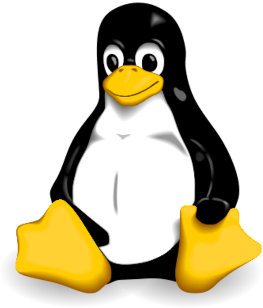
Profiling: `perf`

- performance analysis tool integrated into Linux kernel (since 2.6.31)
- samples software/hardware performance counters at fixed rate
- open-source tool under GPL
- per application or system-wide profiling possible
- more details: perf wiki



Profiling: perf

- performance analysis tool integrated into Linux kernel (since 2.6.31)
- samples software/hardware performance counters at fixed rate
- open-source tool under GPL
- per application or system-wide profiling possible
- more details: perf wiki

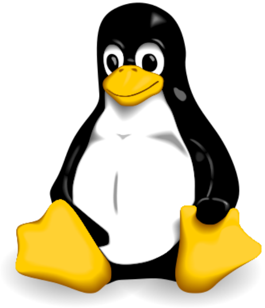


subcommands

perf stats	receive quick performance summary of an application
perf record	record performance profile of an application (`perf.data` is created)
perf report	browse performance profile

Profiling: `perf`

- performance analysis tool integrated into Linux kernel (since 2.6.31)
- samples software/hardware performance counters at fixed rate
- open-source tool under GPL
- per application or system-wide profiling possible
- more details: perf wiki



subcommands

perf stats	receive quick performance summary of an application
perf record	record performance profile of an application (`perf.data` is created)
perf report	browse performance profile

Let's take a tour! (again: stream, ellbow-out)

Part II

Good Code, Bad Code

4. Usual Suspects

5. The Burdens of Design

6. The Free Lunch

Examples of performance problematic use of C++

What do you think?

Examples of performance problematic use of C++

What do you think?

- pay attention to what you write in performance critical sections of your code
- compilers might detect and fix slow code (no guarantee!)
- C++ offers great control (power) and great possibilities for errors (responsibilities)
- if in doubt, **profile** and **talk** to fellow developers/scientists!

Examples of performance problematic use of STL

What do you think?

Examples of performance problematic use of STL

What do you think?

- use the right tools for the right task (containers, algorithms)
- if in doubt, **profile** and **talk** to fellow developers/scientists!

The Burdens of Design: Inheritance

```
struct Direct
{
  int Perform(int &ia) { return --ia; }
};

struct AbstrBase
{
  virtual int Perform(int &ia)=0;
};

struct Derived: public AbstrBase
{
  virtual int Perform(int &ia) { return --ia; }
};

int main(int argc, char* argv[]){
  //...
  int begin = 1 << 30;
  while( direct_ptr->Perform(ia) );
  //...
}
```

First Profile!

g++ 4.8.2

```
Direct          2.66 ms from 1073741824 iterations
Virtual         12.355 ms from 1073741824 iterations
```

First Profile!

g++ 4.8.2

```
Direct          2.66 ms from 1073741824 iterations
Virtual         12.355 ms from 1073741824 iterations
```

g++ 4.8.2, -O2

```
Direct       0 ms from 1073741824 iterations
Virtual  7.317 ms from 1073741824 iterations
```

First Profile!

g++ 4.8.2

```
Direct          2.66 ms from 1073741824 iterations
Virtual         12.355 ms from 1073741824 iterations
```

g++ 4.8.2, -O2

```
Direct        0 ms from 1073741824 iterations
Virtual   7.317 ms from 1073741824 iterations
```

What's going on?

The Burdens of Design: Inheritance

Virtual Table (vtable)

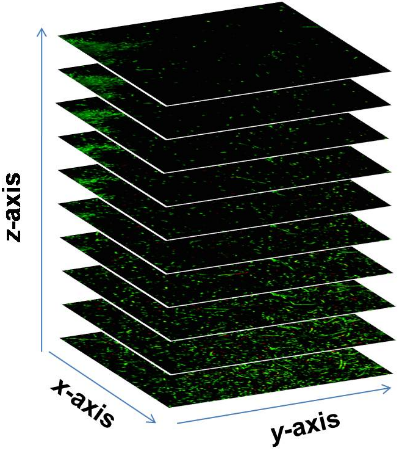
```
*** Dumping AST Record Layout
   0 | class Derived
   0 |   class AbstrBase (primary base)
   0 |     (AbstrBase vtable pointer)
   0 |     (AbstrBase vftable pointer)
     | [sizeof=8, dsize=8, align=8
     |  nvsize=8, nvalign=8]
```

- `vtable` = binary tree of function pointers
- evaluated at **runtime**
- adds 64 bit pointer to class memory footprint
- adds arbitrary amounts of pointer indirections to program flow
- imposes runtime barrier for compiler optimisations

l_2 norm of 3D image stacks

- image stacks made from $n_x \times n_y \times n_z$ pixels
- typically obtained in microscopy, NMR, CT, ...
- compare two stacks $f(x, y, z), g(x, y, z)$ of N pixels by l_2-norm:

$$l_2 = \sum_{i \in N} (f_i - g_i)^2$$



(source bioimagel.com)

Array of Structures

```cpp
struct pixel {

  int x_;
  int y_;
  int z_;

  float intensity_;

};

struct pixel_stack {
  //...
  std::vector<pixel> data_;
  //...
};
```

The Burdens of Design: Over-Objectivism

CBG

Array of Structures

```cpp
   struct pixel {

     int x_;
     int y_;
     int z_;

     float intensity_;

};

   struct pixel_stack {
   //...
   std::vector<pixel> data_;
   //...
};
```

Structure of Arrays

```cpp
   struct flat_stack   {
   //...
   std::vector<float> data_;
   std::vector<int> x_;
   std::vector<int> y_;
   std::vector<int> z_;
   //...
};
```

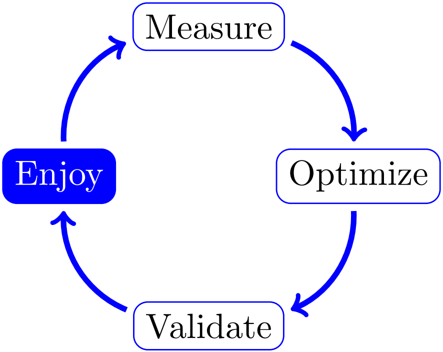
Demo: timings for single core

Demo: timings for single core

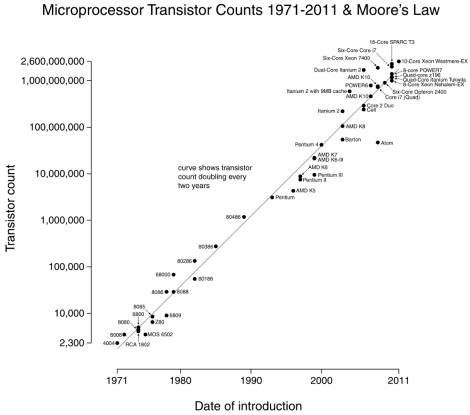
- prime example of “nice” design on small scales, performance problem on large scales
- segmentation of `pixel_stack` causes non-optimal use of caches
- prominent issue with hardware accelerators (they work extremely well with structure of arrays)

The Burdens of Design: Summary

- object-oriented programming was invented to provide a structure
- common tools for object-orientism (e.g. virtual inheritance) impose a performance penalty
- never optimise upon assumption or prematurely, always measure



MOVE!



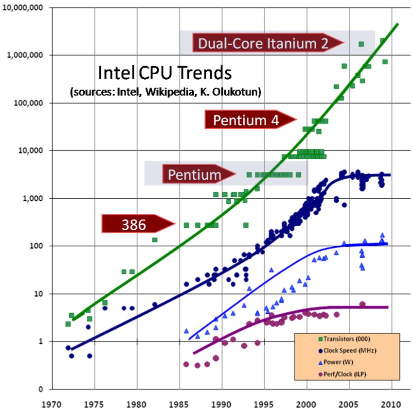
(from wikipedia)

Want speed?

Wait and upgrade your cluster![a]

The Free Lunch: Back in the days

Herb Sutter (herbsutter.com)

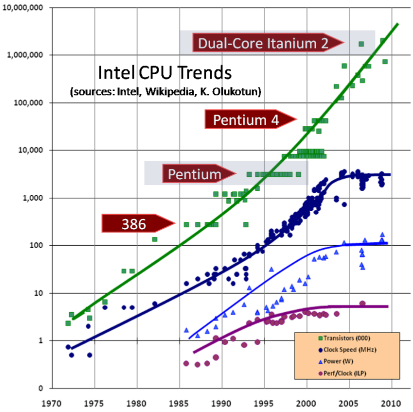


(from [8])

"The Free Lunch is Over" [8]

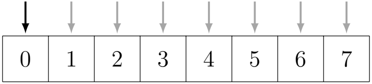
- processor manufactures will focus on products that better support multithreading (such as multi-core processors),
- software developers will be forced to develop massively multithreaded programs as a way to better use such processors.

Herb Sutter (herbsutter.com)



(from [8])

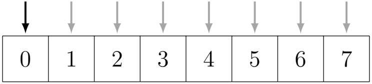
Serial Programming



Single Instruction, Single Data (after [9])

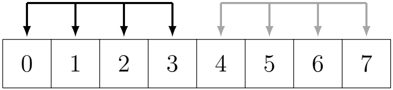
The Free Lunch: A Paradigm Change

Serial Programming

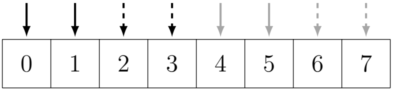


Single Instruction, Single Data (after [9])

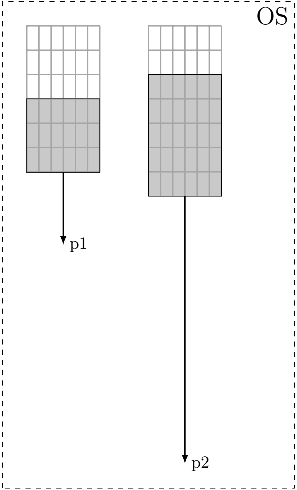
Single Instruction, Multiple Data (SIMD)



Multiple Instruction, Multiple Data (MIMD)

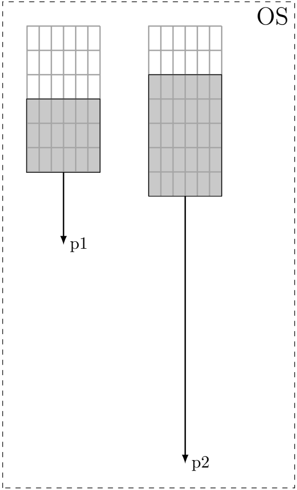


A Process

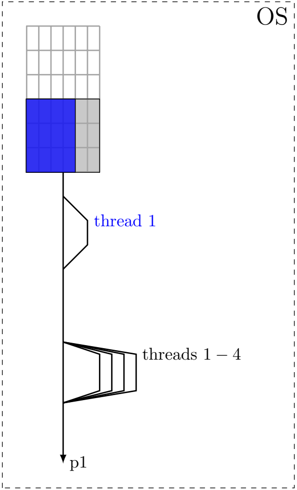


The Free Lunch: A Paradigm Change

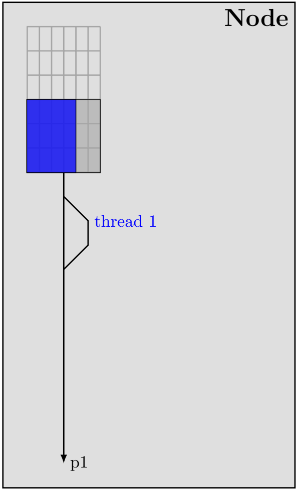
A Process



A Process With Threads

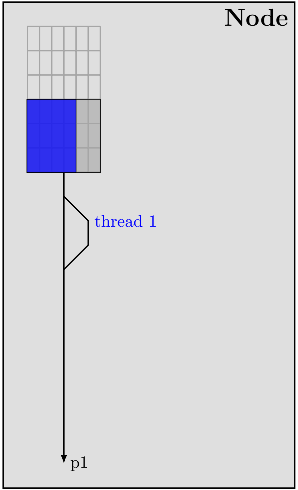


Shared-Memory Parallelisation

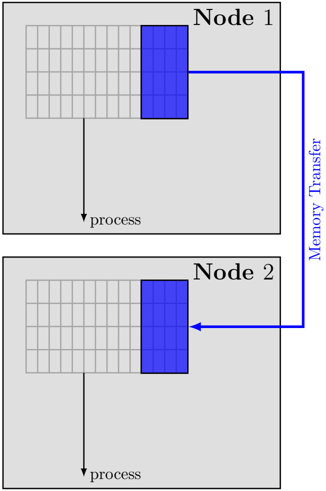


The Free Lunch: A Paradigm Change

Shared-Memory Parallelisation

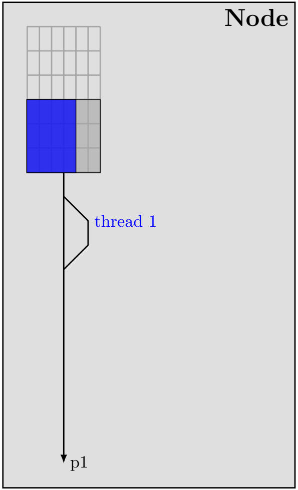


Distributed Parallelisation

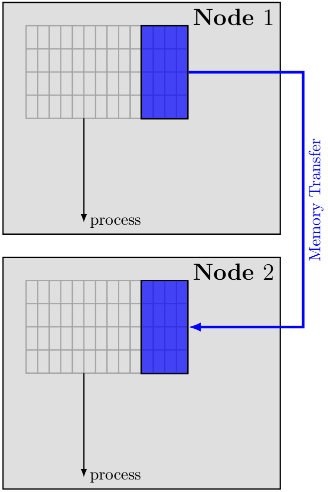


The Free Lunch: A Paradigm Change

Shared-Memory Parallelisation



Distributed Parallelisation



Let's have a look at Shared-memory parallelisation!

Implicit Parallelisation

```
std::vector<float> data(huge_number);
parallel_suite::num_threads = 42;
parallel_suite::parallel_for(data,
                             expensive_operation());
```

Implicit Parallelisation

```
std::vector<float> data(huge_number);
parallel_suite::num_threads = 42;
parallel_suite::parallel_for(data,
                             expensive_operation());
```

Explicit Parallelisation

```
std::vector<float> data(huge_number);
std::vector<thread_t*> threads(42);
for(i in 42){
  threads[i] = new thread_t(data_chunk_begin,
                            data_chunk_end,
                            expensive_operation);
}
//wait some time (synchronisation)
for(i in 42){
  threads[i].join();
  delete threads[i];
}
```

- OpenMP
- Threading Building Blocks
- Intel Cilk
- ...

Implicit Parallelisation

```
std::vector<float> data(huge_number);
parallel_suite::num_threads = 42;
parallel_suite::parallel_for(data,
                             expensive_operation());
```

- POSIX threads (pthreads)
- GrandCentralDispatch
- Boost.Thread
- std::thread (C++11)
- ...

Explicit Parallelisation

```
std::vector<float> data(huge_number);
std::vector<thread_t*> threads(42);
for(i in 42){
  threads[i] = new thread_t(data_chunk_begin,
                            data_chunk_end,
                            expensive_operation);
}
//wait some time (synchronisation)
for(i in 42){
  threads[i].join();
  delete threads[i];
}
```

The Free Lunch: Shared-memory

OpenMP

- C/C++ and Fortran API
- compiler directives (`#pragma omp`), library routines and environment methods
- managed by non-profit technology consortium (AMD, IBM, Intel, Cray, HP, Fujitsu, Nvidia, NEC, Red Hat, . . .)
- OpenMP 4.0 released July 2013 (waiting to be implemented in main-stream compilers)

(from wikipedia)

The Free Lunch: Shared-memory

OpenMP

- C/C++ and Fortran API
- compiler directives (`#pragma omp`), library routines and environment methods
- managed by non-profit technology consortium (AMD, IBM, Intel, Cray, HP, Fujitsu, Nvidia, NEC, Red Hat, . . .)
- OpenMP 4.0 released July 2013 (waiting to be implemented in main-stream compilers)

(from wikipedia)

Your Hello World

```
//compile: g++ -fopenmp omp_hello.cpp -o omp_hello
#include <iostream>

int main(void)
{
  #pragma omp parallel
  std::cout << "Hello, world.\n";
  return 0;
}
```

The Free Lunch: Shared-memory

Computing the sum of a vector

See the example code in the repository!

What to expect?

- given a fixed working set size, parallelisation (multi-threading) can reduce time-to-solution
- having an application with p fraction of code (or fraction of working set) that can be parallelised and s that cannot
- application speedup:

$$S = \frac{1}{s + \frac{1-s}{N}}$$

Amdahl's Law, [10]

- if my source has no serial parts in it anymore (totally unrealistic), the speed-up I can expect is N

Does shared memory parallelisation solve all my problems?

Does shared memory parallelisation solve all my problems?

No!

- **If you can, don't do it!** (Parallelisation should be a last resort)
- not for free (developer time, memory synchronisation, side effects)
- Multi-threading can only boost CPU bound problems
- use implicit multi-threading libraries as much as you can

Loops are everywhere

```
void MatrixSum(float *left,float *right,float *result, size_t size){

  for(int key = 0;key < size;key++){
    result[key] = left[key]+right[key];
  }
}
```

- they come in all variations, sizes and frequencies

Loops are everywhere

```
void MatrixSum(float *left,float *right,float *result, size_t size){

  for(int key = 0;key < size;key++){
    result[key] = left[key]+right[key];
  }
}
```

- they come in all variations, sizes and frequencies
- central building block of programming

Loops are everywhere

```
void MatrixSum(float *left,float *right,float *result, size_t size){

  for(int key = 0;key < size;key++){
    result[key] = left[key]+right[key];
  }
}
```

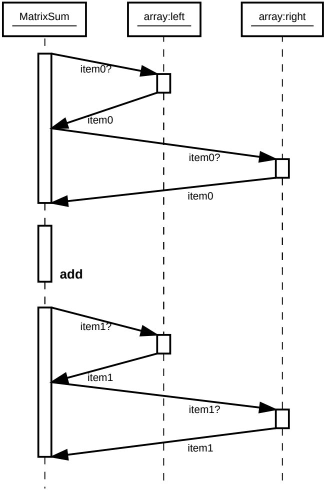
- they come in all variations, sizes and frequencies
- central building block of programming
- do the same (CPU intensive) task repeatedly

Loops are everywhere

```
void MatrixSum(float *left,float *right,float *result, size_t size){

  for(int key = 0;key < size;key++){
    result[key] = left[key]+right[key];
  }
}
```

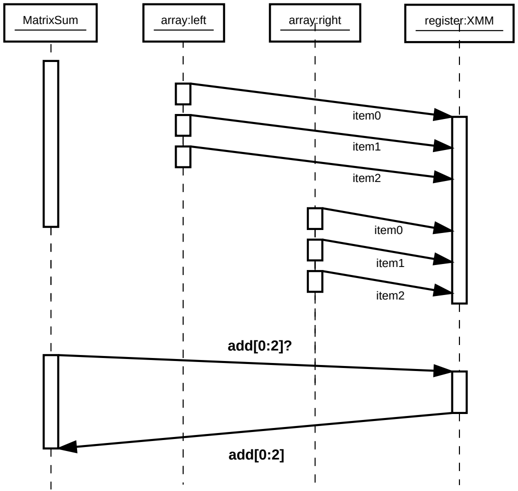
- they come in all variations, sizes and frequencies
- central building block of programming
- do the same (CPU intensive) task repeatedly
- **hot spot with regard to performance**

Behind the scenes

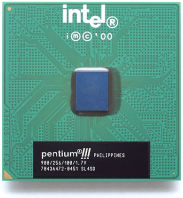


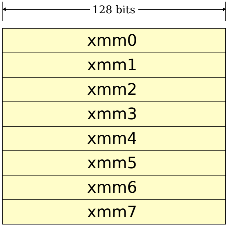
The Free Lunch: Vectorisation

Instead of going one-by-one ...



- "Streaming **S**ingle **I**nput **M**ultiple **D**ata Extensions"
- Intel introduces extra 128 bit (16 Byte) registers to `Pentium III` chipsets (450 – 1400 MHz,1999)
    - CPU can work on extra registers independent of other caches (parallelism!)
- registered called `XMM0-XMM7` based on former MMX designs
- 70 new `x86` assembler instructions for single precision floating point data





The Free Lunch: Vectorisation

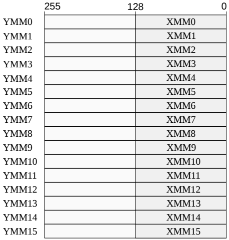
- AMD added 8 more registers for `x86_64` architecture to `SSE`
- initial version `SSE` was followed by
  - `SSE2` (Intel P4, 2001),
  - `SSE3` (Intel P4 revised, 2004),
  - `SSSE3` (Intel Core architecture, 2006)
  - `SSE4` (Intel Core architecture, 2006)
  - `SSE5` (AMD, 2007)
- adding more arithmetic, conversion, data-movement etc instructions

The Free Lunch: Vectorisation

- AMD added 8 more registers for `x86_64` architecture to `SSE`
- initial version `SSE` was followed by
  - `SSE2` (Intel P4, 2001),
  - `SSE3` (Intel P4 revised, 2004),
  - `SSSE3` (Intel Core architecture, 2006)
  - `SSE4` (Intel Core architecture, 2006)
  - `SSE5` (AMD, 2007)
- adding more arithmetic, conversion, data-movement etc instructions

Advanced Vector Extensions (AVX)

- first supported by Intel Sandy Bridge (Q1 2011) and AMD Bulldozer CPUs (Q3 2011)
- enlarged registers to 256 bit (32 Byte) width, `YMM0-YMM15`
- supports SIMD floating point operations for `YMM*` registers
- AVX2 added support for integer operations (Intel Haswell architecture, Q2 2013)
- AVX-512 to appear 2015



The Free Lunch: Vectorisation

- writing applications in assembler is **no-go** for common people

- writing applications in assembler is **no-go** for common people
- all common compilers (gcc, MSVS, llvm/clang, Intel, open64) support SIMD

The Free Lunch: Vectorisation

- writing applications in assembler is **no-go** for common people
- all common compilers (gcc, MSVS, llvm/clang, Intel, open64) support SIMD
- **auto-vectorisation**: compiler heuristic to detect loops and convert them to SEE compliant assembler code [11, 12, 13]

The Free Lunch: Vectorisation

- writing applications in assembler is **no-go** for common people
- all common compilers (gcc, MSVS, llvm/clang, Intel, open64) support SIMD
- **auto-vectorisation**: compiler heuristic to detect loops and convert them to SEE compliant assembler code [11, 12, 13]

The Free Lunch: Vectorisation

- writing applications in assembler is **no-go** for common people
- all common compilers (gcc, MSVS, llvm/clang, Intel, open64) support SIMD
- **auto-vectorisation**: compiler heuristic to detect loops and convert them to SEE compliant assembler code [11, 12, 13]

```
# standard optimisations on
$ g++ -O2 ... MatrixSum.cpp -o MatrixSum
# auto-vectorisation on
$ g++ -O2 ... -ftree-vectorize MatrixSum.cpp -o MatrixSum
```

Does SIMD auto-vectorisation solve all my problems?

Does SIMD auto-vectorisation solve all my problems?

No!

Does SIMD auto-vectorisation solve all my problems?

No!

- SIMD will only help with CPU bound problems

Does SIMD auto-vectorisation solve all my problems?

No!

- SIMD will only help with CPU bound problems
- SIMD only works for unit stride for-loops

```
for(int i = 0;i<max;++i) //...
```

Does SIMD auto-vectorisation solve all my problems?

No!

- SIMD will only help with CPU bound problems
- SIMD only works for unit stride for-loops

```
for(int i = 0;i<max;++i) //...
```

- heuristics only work with finger's crossed (sometimes extra code required to have a loop vectorised at all)

Does SIMD auto-vectorisation solve all my problems?

No!

- SIMD will only help with CPU bound problems
- SIMD only works for unit stride for-loops

```
for(int i = 0;i<max;++i) //...
```

- heuristics only work with finger's crossed (sometimes extra code required to have a loop vectorised at all)
- auto-vectorisation requires checking the assembler output to obtain optimal performance

Does SIMD auto-vectorisation solve all my problems?

No!

- SIMD will only help with CPU bound problems
- SIMD only works for unit stride for-loops

```
for(int i = 0;i<max;++i) //...
```

- heuristics only work with finger's crossed (sometimes extra code required to have a loop vectorised at all)
- auto-vectorisation requires checking the assembler output to obtain optimal performance
- particular to `C++`: **SIMD intrinsics** low-level SIMD API (available inside the compiler)

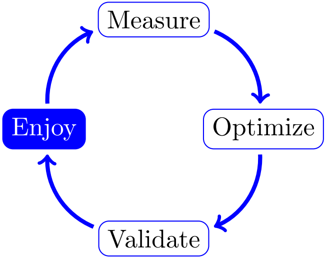
Standard Source Code

```
void MatrixSum(float *left,float *right,float *result, size_t size){
  for(int key = 0;key < size;key++){
    result[key] = left[key]+right[key];
  }
}
```

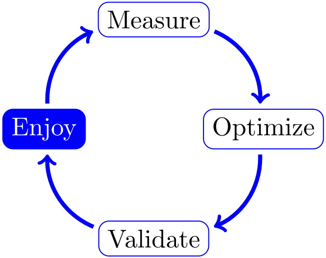
Source Code With Intrinsics

```
#include "x86intrin.h"
void MatrixSum(float *left,float *right,float *result, size_t size){
  for(size_t i = 0;i<size;i+=4){
    __m128 lhs=_mm_load_ps(&_lhs[i]);
    __m128 rhs=_mm_load_ps(&_rhs[i]);
    __m128 res=_mm_add_ps(lhs,rhs);
    _mm_store_ps(&_result[i],res);
  }
}
```

- **MOVE**!

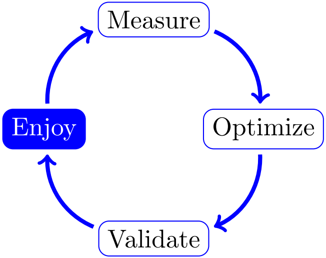


- **MOVE**!
- Shared Memory Parallelisation and SIMD have become standard tools to enhance library performance



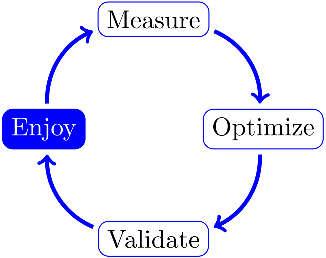
The Free Lunch: Summary

- **MOVE**!
- Shared Memory Parallelisation and SIMD have become standard tools to enhance library performance
- both targeted at CPU bound problems



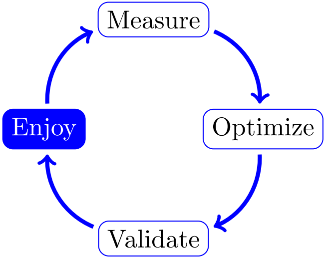
The Free Lunch: Summary

- **MOVE**!
- Shared Memory Parallelisation and SIMD have become standard tools to enhance library performance
- both targeted at CPU bound problems
- both incur developer time overhead (learning, fixing, optimising)



The Free Lunch: Summary

- **MOVE**!
- Shared Memory Parallelisation and SIMD have become standard tools to enhance library performance
- both targeted at CPU bound problems
- both incur developer time overhead (learning, fixing, optimising)
- both required to exploit state-of-the-art CPU architectures



Part III

Exercises

Literature

[1] D. E. Knuth, "Structured programming with go to statements," *Computing Surveys*, vol. 6, pp. 261–301, 1974.

[2] J. P. Holdren *et al.*, "Designing a digital future: Federally funded research and devevelopment in networking and information technology," report to the president and congress, Executive Office of the US President, December 2010.

[3] N. Coorporation, "Cuda c best practices guide."

[4] J. von Neumann, "First draft of a report on the edvac." Contract No. W-670-ORD-4926, Between the United States Army Ordinance Department and the University of Pennsylvania Moore School of Electrical Engineering, June 1945.

[5] *Introduction to High Performance Computing for Scientists and Engineers*.
CRC Press, Taylor & Francis Group, 2011.

[6] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[7] J. D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers," tech. rep., University of Virginia, Charlottesville, Virginia, 1991-2007.
A continually updated technical report. http://www.cs.virginia.edu/stream/.

[8] H. Sutter, "The free lunch is over," *Dr Dobbs*, vol. 30, no. 3, 2005.

[9] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans. Comput.*, vol. 21, pp. 948–960, Sept. 1972.

[10] D. P. Rodgers, "Improvements in multiprocessor system design," *SIGARCH Comput. Archit. News*, vol. 13, pp. 225–231, June 1985.

[11] "Auto-vectorization in llvm."

[12] "A guide to auto-vectorization with intelÂ® c++ compilers."

[13] "Auto-vectorization in gcc."

[14] A. Hunt and D. Thomas, *The Pragmatic Programmer*.
Addison and Wesley, 2000.