Parallel Programming (IN2147)
Optimization of Sequential Programs

Martin Schulz
Alexis Engelke

Chair for Computer Architecture and Parallel Systems
Department of Informatics
Technical University of Munich

June 25, 2018

New TOP500 List

And the winner is:

Summit

Oak Ridge National Laboratory

2,282,544 cores – 122.3 PFlop/s – 8.8 MW
13.889 GFlops/Watt (rank 5 in Green500)

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/SC/Oak Ridge National Laboratory United States	2,282,544	122,300.0	187,659.3	8,806
2	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
3	**Sierra** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/NNSA/LLNL United States	1,572,480	71,610.0	119,193.6	
4	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 , NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482

Which strategy?

$$\text{Parallel} \longrightarrow \text{Seq. Opt} \rightleftarrows \text{Par. Opt}$$

- First ensure scalability
- Optimizing sequential code gives constant factors
- Optimization is an iterative process

Programming Languages I

- Major differences in:
  - Efficiency
  - Ease of programming
  - Abstractions
- Classic HPC languages: C, Fortran
  - Rather low-level, allow for manual optimizations
  - Efficient machine code (mostly)
  - Rather low programming comfort

Programming Languages II

- Byte-code compiled languages: Java
  - More easy to program
  - Machine code not as efficient
- Scripting languages: Python, Perl, JavaScript, ...
  - Very easy to program (typically)
  - Don't expect performance
- New languages: C++, Swift, Rust, Go, ...
  - C++: gaining traction in HPC
  - Go: Compile-time is important, runtime is not
  - Others: To be seen...

Detecting optimization potential

- Optimizing code taking 1% of total time?
  - Probably not worth the effort
- Analysis of optimization potential is important
- Profiling helps in analysis
  - Tools: `perf`, Gprof, etc.
- Start with part having the biggest impact on performance

Algorithms

- Last week: choose algorithms which scale
- Today: care about sequential performance

- Vectorization, super-scalarity
  - Independent parallel computations
  - Data-parallelism
- Cache efficiency
  - Regular access patterns
  - Few indirections

Abstractions

- Abstractions make (programmer's) life easier
  - No need to care about technical details
  - Increases portability
  - That's why they are all over the place ☺
- Often, many abstractions are stacked
- Abstractions (often) introduce overhead
- Abstractions at some point leak
  - E.g. lead to strange performance effects

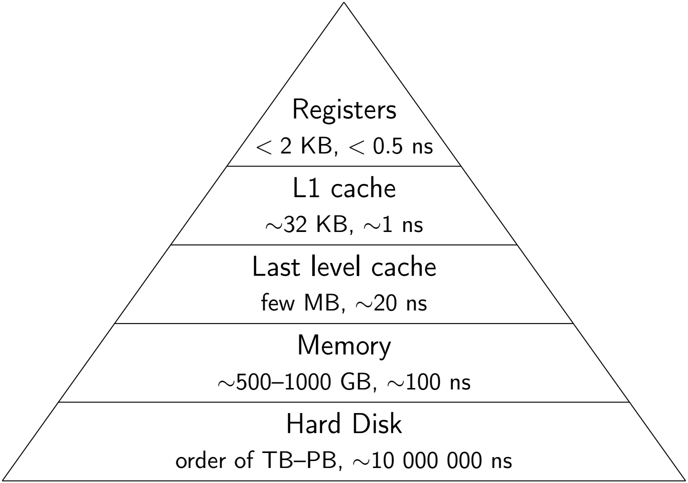
Abstractions

- C++ makes it easy to use complex abstractions
  - Bounds checking (indexing)
  - Hidden function calls (operator overloading)
  - Indirect function calls (vtables)
  - Many (hidden) pointer dereferences (references)
  - Random access in memory (`std::list`)
  - Huge code size (templates)
  - ...

Abstractions

- Be aware of (hidden) abstractions
- Know their impact
- Use abstractions with care,
  avoid if possible with reasonable effort

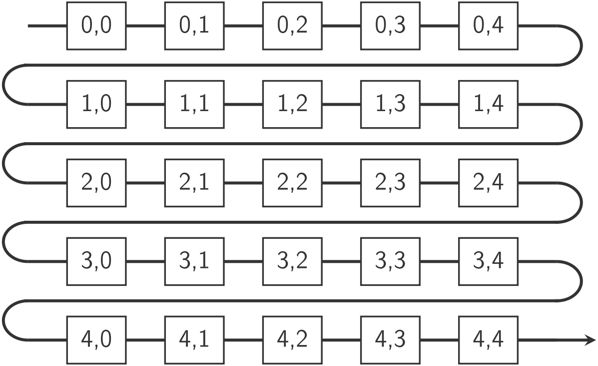
Memory Hierarchy (simplified)



Cache Optimizations

- Exploit spatial and temporal locality
- Data layout in memory
  - Row-major vs. column-major arrays – hello Fortran
- Predictable, regular access pattern allows prefetching
- Prefetch instructions (use with care)
- "Blocking" in loops
- Avoid cache pollution
  - Streaming instructions don't write to the cache

Layout of Matrices in Memory



- What's better? Row-wise vs. column-wise access?
- Row-wise access > 50% faster (in C)

Cache Optimizations

- Exploit spatial and temporal locality
- Data layout in memory
  - Row-major vs. column-major arrays – hello Fortran
- Predictable, regular access pattern allows prefetching
- Prefetch instructions (use with care)
- “Blocking” in loops
- Avoid cache pollution
  - Streaming instructions don’t write to the cache

Cache Optimization

- Exploit spatial and temporal locality
- Cache optimization may require large code changes
- Cache optimization can yield large speed-ups
- Tools may help: Cachegrind, KCachegrind
    - KCachegrind developed by Josef Weidendorfer (LRZ)

Compiler Optimizations

- Compilers generate and optimize machine code
- Compilers (usually) apply (complex) code transformations
  - Only if proven to be correct
- Compilers (usually) **don't** change data structures
  - Typically impossible to prove correctness automatically
- Compilers (usually) **don't** optimize maths
  - If they do, don't trust the result
  - Mathematical optimizations can change accuracy

Common Optimization Options

- `-O0` – no optimization
- `-O1` – "optimize"
  - Better register allocation, dead code elimination, . . .
- `-O2` – "optimize even more"
  - More aggressive CSE, remove redundant instructions, . . .
- `-O3` – "optimize yet more"
  - Aggressive inlining, vectorization, . . .
- `-Os` – "optimize for size"
- `-Og` – "optimize debugging experience"
- `-Ofast` – "disregard strict standards compliance."
  - Floating-point optimizations
- `-march=native` (in addition) – architecture tuning

Some Optimizations

- Loop-invariant Code Motion (LIM/LICM)
  - Statements independent of the loop moved outside
  - Avoid redundant execution of code
- Loop Unrolling
  - Loop is known to be executed 5 times
  - Copy the loop body 5 times
  - No loop overhead, but code size grows
- Inlining
  - Body of other function is copied into the caller
  - No overhead through call, calling convention, etc.

Vectorization

- Auto-Vectorization…
  - Works well for simple cases
  - High overhead for complex code (if vectorized at all)
  - May require `restrict` keyword
- Manual vectorization can yield high performance improvement
  - Even compared to the Intel compiler
- Manual vectorization is target-dependent
  - Portability? Development time?

Floating-point Optimizations

- IEEE-754 defines floating-point numbers and operations
- Possible to optimize $x + 0 \rightarrow x$? — **No!**
  - Signed zeros, $(-0) + (+0) = (+0)$
- Possible to optimize $x - x \rightarrow 0$? — **No!**
  - If x is `NaN`, result is `NaN`

- Options for relaxing IEEE semantics
- Trade-off: performance vs. accuracy
- Note: enabling `-ffast-math` can make code slower

Compilers: Miscellaneous

- Providing Hints
  - `restrict` keyword
    - Pointers don't overlap each other
  - `inline` keyword
  - `__attribute__((aligned(32)))`
- Intrinsics (see lecture on SIMD)
- Inter-procedural Optimization
  - Compile and link with `-flto`
  - Unified builds (combined with `-fwhole-program`)
- In doubt, analyze generated assembly code

Hand-written Assembly Routines

Should you write assembly by hand?

NO

(unless you have a good reason)

- Possible reasons for writing assembly routines:
  - Hot code that compiler *really* fails to optimize
  - Intrinsics don't yield intended effect
  - "Research Code"

Time Measurement

- Preferred functions: `MPI_Wtime` or `clock_gettime(CLOCK_MONOTONIC_RAW, ...)`
- Common pitfalls:
  - Measured time too short
  - No repetitions to exclude external influences
  - Measurement of I/O or other syscalls (unless you want to measure I/O)
  - Wrong clock, e.g. CPU time instead of wall-clock time

Ongoing Research

Research: Dynamic Code Generation

- Limitations of classic compile-execute model:
  Overhead through indirections and missing runtime data
  - Input data/Configuration
  - Previous computations
  - Data distribution, scheduling
  - Highly irregular data structures
  - ...
- Idea: Incorporate data in machine code

Dyn. Code Gen.: Approaches

- Dedicated languages – failed 20 years ago

- LLVM: full-featured compilation framework
- LIBXSMM: generate code for matrix multiplications and convolutions
  - Developed by Intel
  - Generates highly tuned code

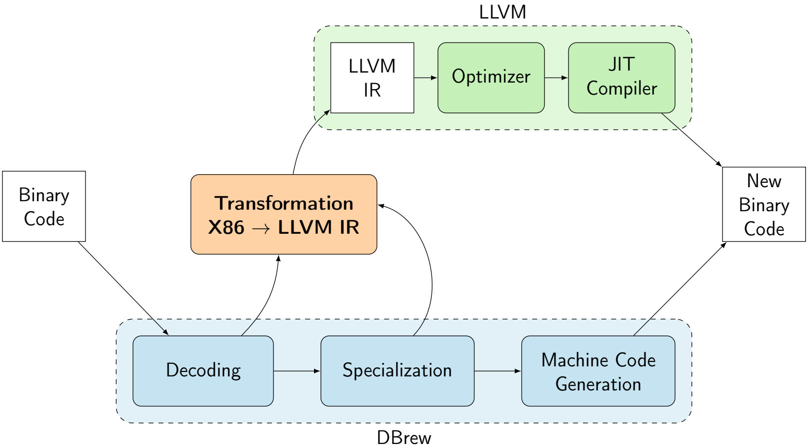
- DBrew: dynamic binary rewriting
  - Developed at CAPS, TUM
  - Specialize existing compiled functions at runtime

DBrew

- Library for binary rewriting at runtime
- Operates on functions, producing drop-in replacements
- Targets x86-64
- Specialization by fixing parameters and memory regions

- Very simple optimizations only (fast code generation)
- Advanced optimizations available via LLVM

DBrew: Overview

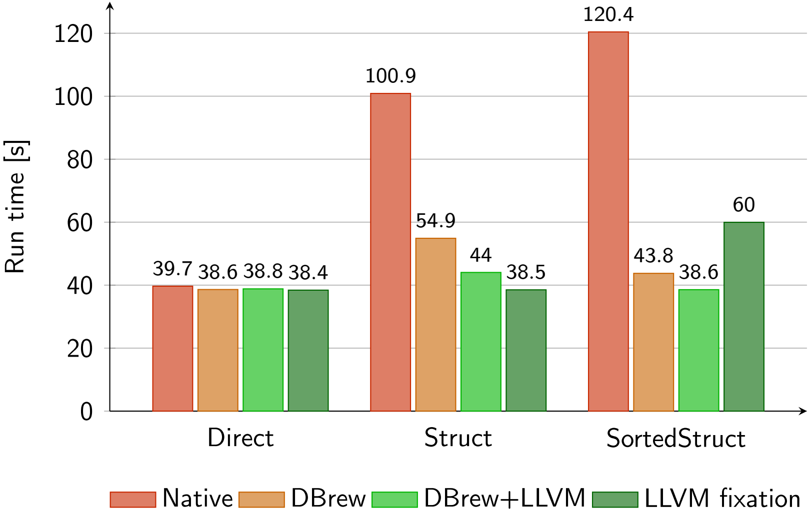


DBrew: Example

```
Rewriter* r = dbrew_new(fn);
dbrew_const_param(r, 0, 42);
dbrew_const_mem_nested(r, stencil, sizeof(Stencil));
Func new_fn = dbrew_rewrite(r);
```

- Possible to approach “native” performance (?)

Results when specializing Stencil



Summary

- Care about scalability first
- Choice of programming languages, algorithms, data structures has high impact on performance
- Complex abstractions slow down
- Cache optimization can bring high constant factors
- Compilers do technical optimizations only

- Code generation at runtime can improve performance, new techniques are under research

Thank you!