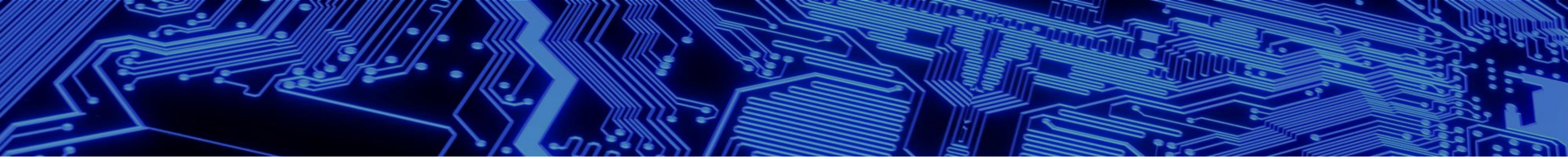


Numerical Simulation and Scientific Computing I

Lecture 2:
Serial Code Optimization

Clemens Etl, Paul Manstetten, and
Josef Weinbub

Institute for Microelectronics
TU Wien

nssc@iue.tuwien.ac.at

WS 2023

Quiz – Question 1

- What is the minimum sum of exercise points you need to be eligible for the exam?

Quiz – Question 1

- What is the minimum sum of exercise points you need to be eligible for the exam?
  - Answer: at least 21 (of 30=3x10)

Quiz – Question 2

- Read the specification of this CPU

  https://en.wikichip.org/wiki/intel/core_i9/i9-13900k

and classify it according to:

- Computer category
- Supported forms of architectural parallelism
- Flynn’s taxonomy
- Instruction set architecture
- Memory hierarchy (cache levels)
- Type of multithreading
- Type of multicore architecture

Quiz – Question 2

Intel i9-13900k

- Brand new 2022, x86, 64bit Desktop CPU
  - 8 “big cores” (3-5.5GHZ), L1 I32K/D48K, L2 8x2M, L3 8x3M
  - 16 “small cores” (2-4.4GHZ), L1 I64K/D32K, L2 4x4, L3 4x3M
- Flynn’s Taxonomy
  - SISD: yes (single processor, ILP)
  - SIMD: yes (vector extensions: AVX2)
  - MIMD: yes (8 + 16 cores)
- Multithreading: SMT and thread-level
- Tightly-coupled MIMD: two-way SMP

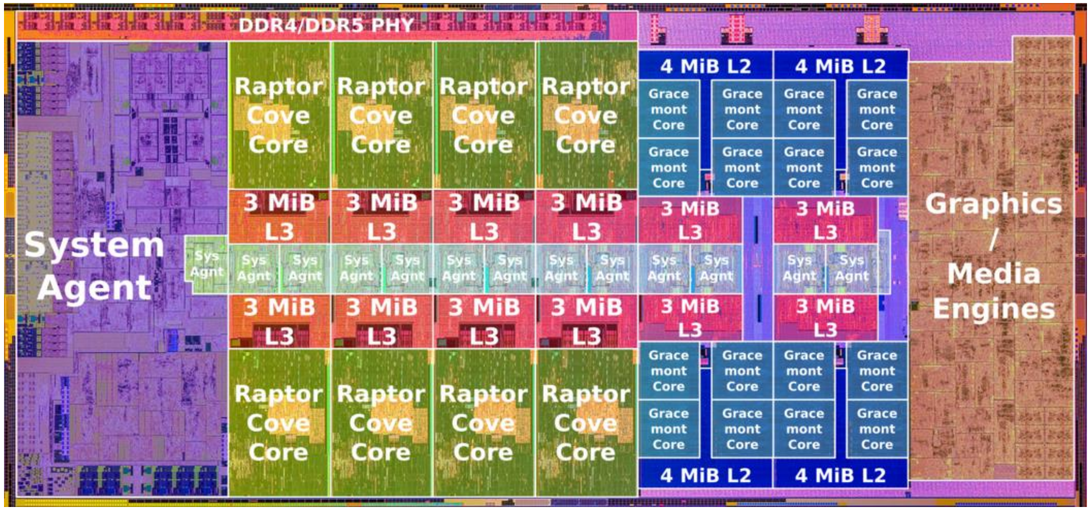


Image: https://en.wikichip.org

Quiz – Question 3

- Caches are very expensive. Why do we need them at all?
  - Situation without caches (or very small caches)?

Quiz – Question 4

- What are simple ways you can use a compiler to make your program run faster?

Quiz – Question 5

- Why are SIMD extensions (e.g. AVX-512) not considered instruction-level parallelism?

Outline

Optimization stages

Performance modeling for a
“cache-based stored-program microprocessor”

The Optimizing Compiler

“It is easier

to make a correct program fast than

to make a fast program correct.”

Outline: Optimization Stages

- Basic Choices
  - General purpose vs. specialized hardware
  - Language / language “infrastructure”
  - Algorithms
  - Data structures
- Compile time
  - Static compilation ahead of time (AOT)
- Link time
  - Link time optimization (LTO) across module boundaries
- Run time
  - Using run time statistics for profile guided optimization (PGO)
- Summary

Compute Platform

- Choice of suitable and available compute platform
  - General purpose CPU: Intel, AMD, ARM, …
  - General purpose GPU: Nvidia, AMD, …
  - Special hardware, e.g., Google's TPUs, Bitcoin Miner ASICs, …
- Example for specified theoretical max. performance metrics

*table data from 2019	CPU W-2195	GPU Tesla T4	TPU Google v2	ASIC Antminer S9
Floating point operations	1710 GFlops/s (FP64)	8100 Gflops/s (FP32)	180 000 GFlops/s (FP16)	14 000 GHashes/s sha256(sha256(80byte))
Memory bandwidth	85.3 GByte/s	320 GByte/s	--	--
TDP	140W	70W	75W	1300W
Price	~3000USD	~3000USD	--	~3000USD

Programming Language

- Choice of programming language
  - Experience
  - Features of the language
  - Available runtimes/compilers/libraries
  - Interfacing to required resources or other language/runtimes
  - Portability
  - Performance requirements

Language	Implementations/ Runtimes/Compilers	Recent Versions	Comment
Python	CPython, PyPy	(python2), python3	interpreted, native extensions
JavaScript	V8,JSC,JS	ECMAScript15("ES6")	interpreted, JIT comp. for optimization
C++	libc++ with clang, libstdc++ with gcc	C++11/14,C++17	statically typed native

Algorithms / Data Structures

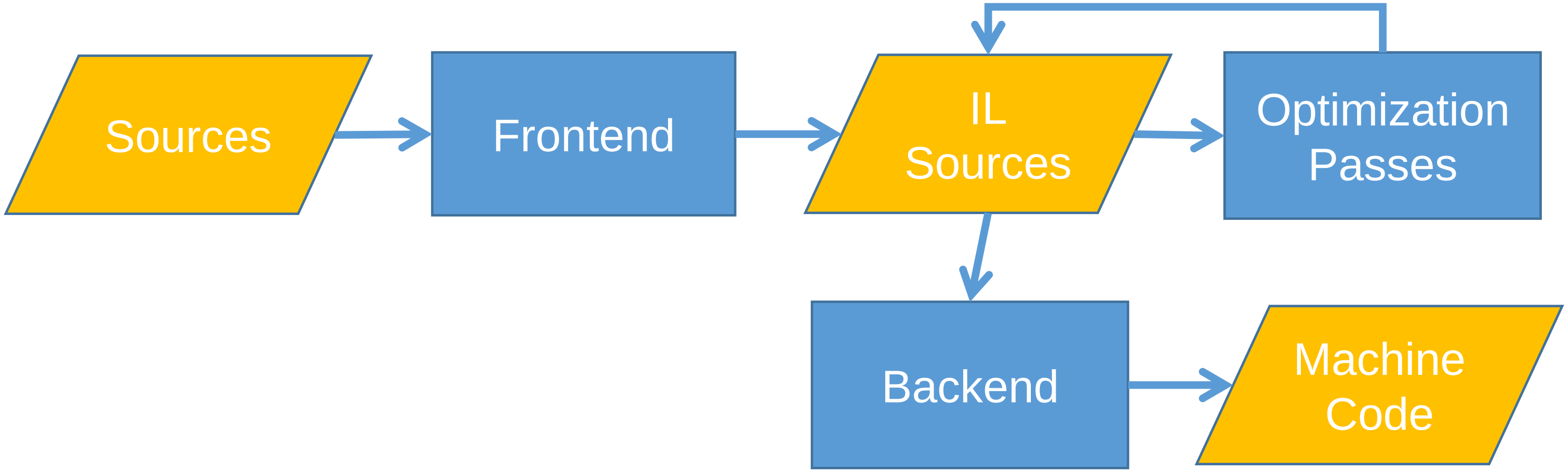
- Choice of suitable algorithm for each computational task
  - Consider the algorithmic complexity w.r.t. problem size
  - Consider the underlying data structure w.r.t.
    - memory footprint for anticipated problem sizes
    - anticipated access patterns
  - Tradeoff with implementation effort
- Practically achieved performance greatly influenced by …
  - the implementation on a particular platform,
  - the considered problem size,
  - run time access patterns,
  - which is all “hidden” in a prefactor “K * O (…)”.

Examples of complexity guarantees for containers in the C++ standard library

Container	Insert	Read	Erase
std::vector<T>	O(1) , O(N)	O(1)	O(1) , O(N)
std::map<T1,T2>	O(log N)	O(log N)	O(log N)
std::unordered_map<T1,T2>	O(1) , O(N)	O(1) , O(n)	O(1) , O(n)

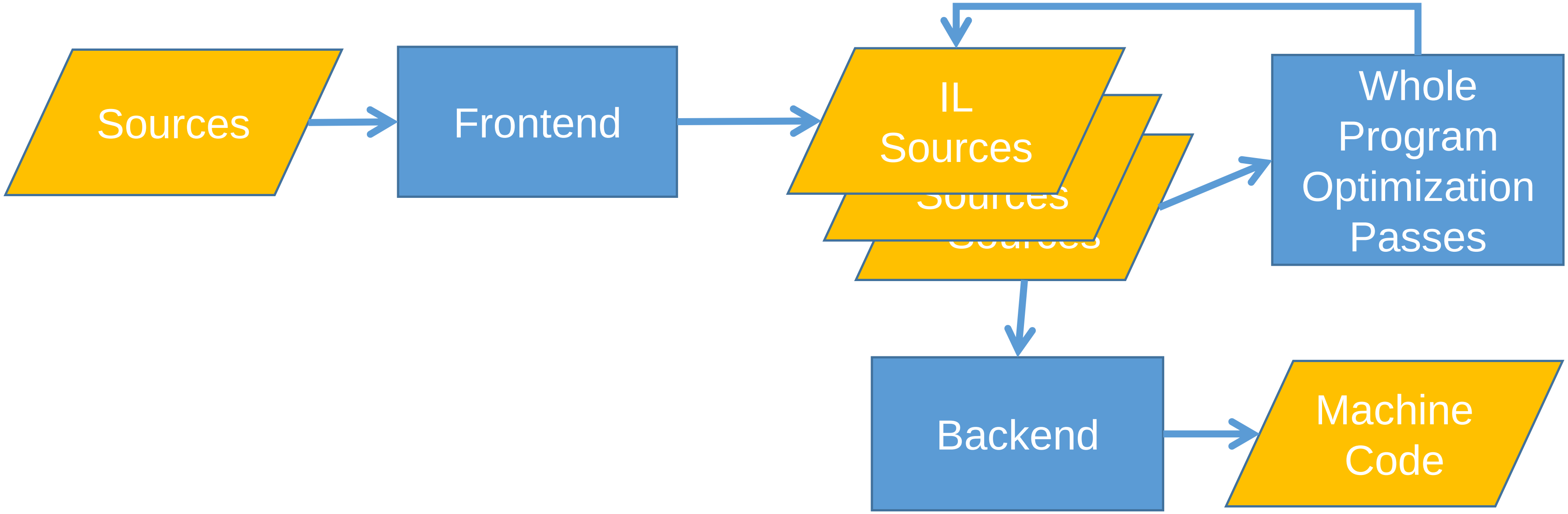
Optimization At Compile Time (C++)

- Optimization flags control static code optimization during AOT compilation
  - Various optimizations for loops (unroll, interchange, vectorize), function calls (inlining), reordering of independent instructions
- Optimization is affected by target architecture
  - # of Registers, special registers (SIMD), # functional units, pipelining capabilities, cache hierarchy/sizes
- Optimization scope is limited to individual compilation units



Optimization At Link Time (C++)

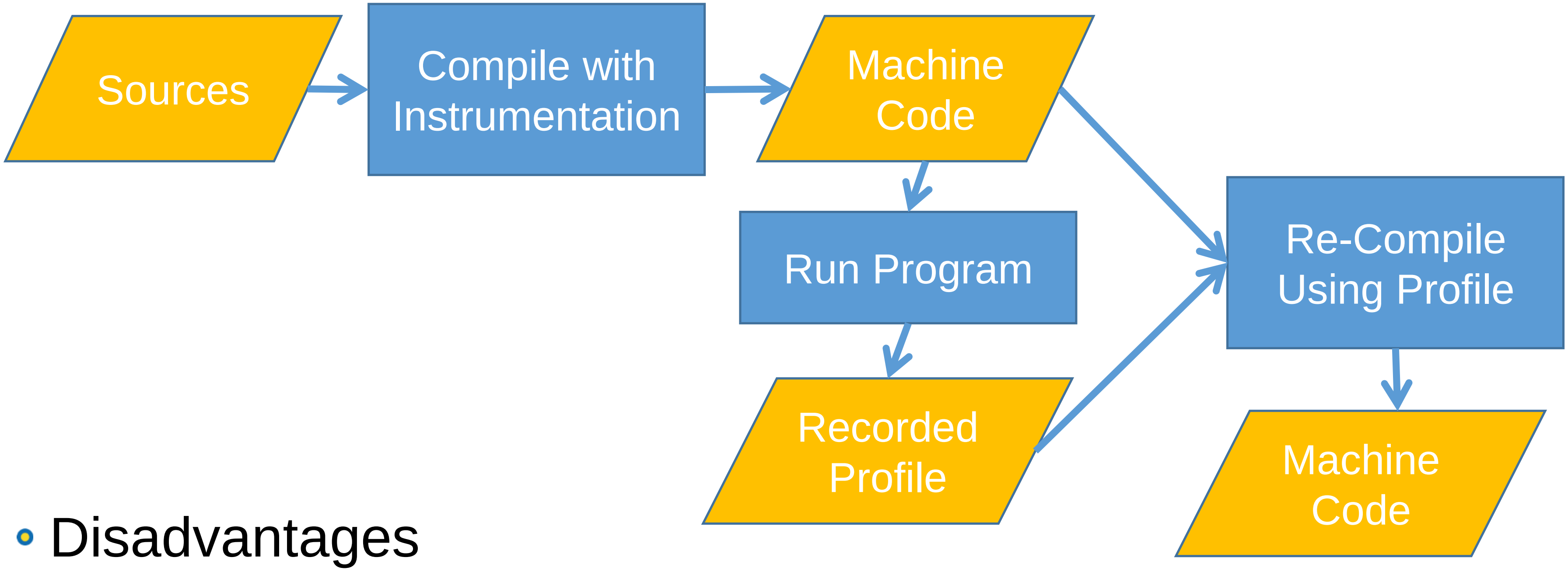
- Whole program optimization ("-flto")
  - Function inlining across module boundaries, dead code elimination



- Disadvantages (as with larger "header-only" projects)
  - Long compilation times
  - Requires a lot of memory, hard to parallelize

Optimization At Run Time (C++)

- Profile guided optimization (PGO)
  - Optimization decisions based on run time profile
  - Also attractive for JIT compilation used by interpreters: use run time statistics to statically compile “hot” regions of the code



- Disadvantages
  - Optimized for input used to generate profiles: unpredictable performance for different inputs
  - Multiple compiles necessary

Summary: Optimization Stages

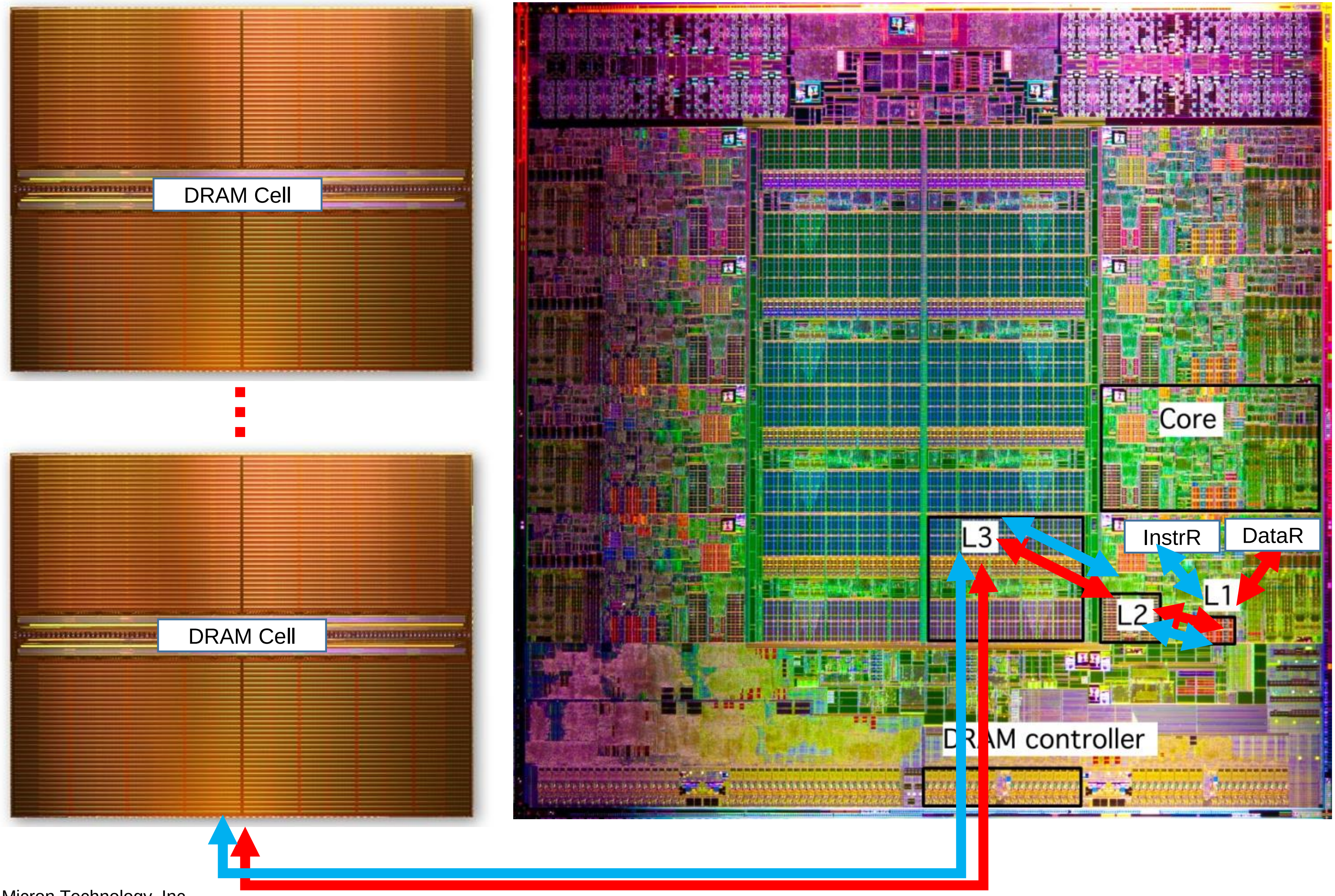
- Optimization Stages
  - Platform / Data Structures / Algorithm / Language
    - Often this choice is constrained
  - Compile Time (AOT)
    - Static optimizations on module level
  - Link Time (LTO)
    - Static optimizations between modules
  - Run Time (PGO)
    - Optimization using run time statistics to guide opt. decisions
- Python (interpreted)
  - For performance: native C/C++ extensions (e.g., numpy)
- JavaScript (interpreted)
  - For performance: JIT=~PGO, precompiled/typed extensions
- C++ (compiled)
  - For performance: AOT compiled, LTO, PGO

Outline: Performance Modeling

- Considered Architecture:

  “Cache-based stored-program” x86 microprocessor
- Bandwidth-based modeling
- Data sizes
- Optimization potential estimation
- Effective bandwidth benchmarking: “Vector Triad”

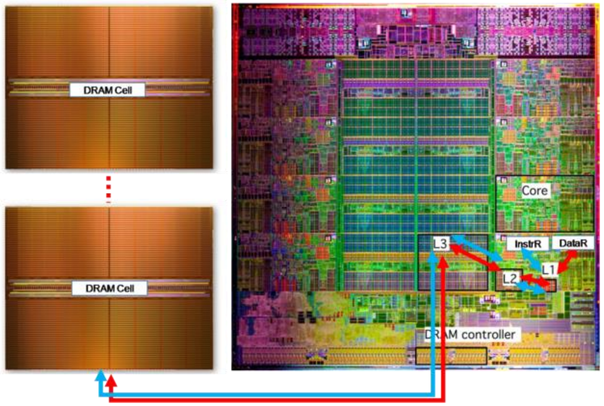
Cache-Based Stored-Program x86 MC



Micron Technology, Inc

Sandybridge, http://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-basics.html

Bandwidth-Based Modeling



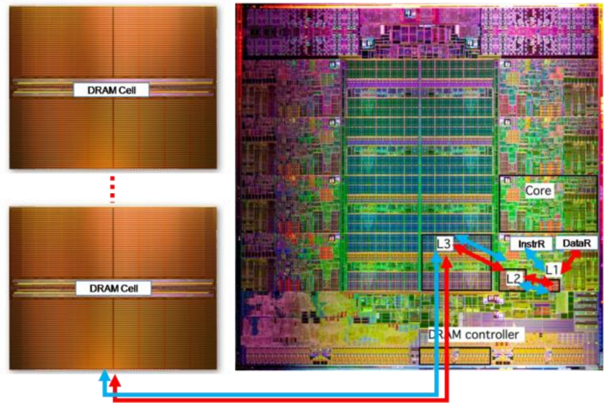
- Readily accessible for most systems:
  - Peak performance = maximum theoretical possible Flops/s
    - Including full SIMD utilization, including “FMA”
    - Underlying floating point representation depends on platform
  - Memory bandwidth = maximum theoretical main memory bandwidth
    - A single core might not be able to utilize this maximum
- Machine Balance

$$B_{machine} = \frac{memory\ bandwidth\,[Bytes/s]}{peak\ \ performance\,[Flops/s]} = \frac{Bytes}{Flops}$$

- Code Balance

$$B_{code} = \frac{data\ traffic\,[Bytes]}{floating\ \ p.\,operations\,[Flops]} = \frac{Bytes}{Flops}$$

Bandwidth-Based Modeling



- Machine Balance

$$B_{machine} = \frac{memory\ bandwidth\,[Bytes/s]}{peak\ \ performance\,[Flops/s]} = \frac{Bytes}{Flops}$$

- i7-4790K (2014), all cores

$$B_{machine} = \frac{25.6\,[GBytes/s]}{256\,[GFlops/s]} = 0.1\frac{Bytes}{Flops}$$

- i7-940 (2008), all cores

$$B_{machine} = \frac{25.6\,[GBytes/s]}{47\,[GFlops/s]} = 0.54\frac{Bytes}{Flops}$$

- Tesla T4 (2018), single precision FP

$$B_{machine} = \frac{85.3\,[GBytes/s]}{1710\,[GFlops/s]} = 0.05\frac{Bytes}{Flops}$$

Bandwidth-Based Modeling

- Example
  - i7-4790K (2014), all cores

$$B_{machine} = \frac{25.6\,[GBytes/s]}{256\,[GFlops/s]} = 0.1\frac{Bytes}{Flops} \qquad I_{machine} = 10\frac{Flops}{Bytes}$$

  - Benchmark code: “Vector Addition” (Large N)

$$B_{code} = \frac{3\cdot 8\cdot N\,[Bytes]}{1\cdot N\,[Flops]} = 24\frac{Bytes}{Flops} \qquad I_{code} = 0.04\frac{Flops}{Bytes}$$

  - Expected fraction of theoretical peak performance

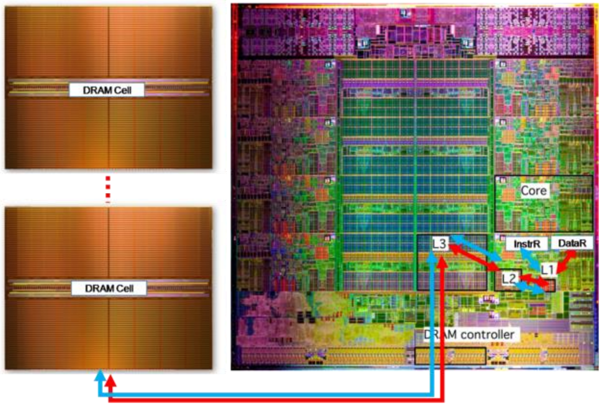
$$\min\left(1, \frac{B_{machine}}{B_{code}}\right) = \min\left(1, \frac{I_{code}}{I_{machine}}\right) = \min\left(1, \frac{0.04}{10}\right) = 0.004$$

  - Estimate peak performance and memory bandwidth (single core) more realistic (no SIMD & single core)

$$P_{peak(fullSIMD,allcores)} = 4[GHz]\cdot 4[Cores]\cdot 2[SIMD_{units}]\cdot 4[SIMD_{width}]\cdot 2[Flops(FMA)] = 256[GFlops/s]$$

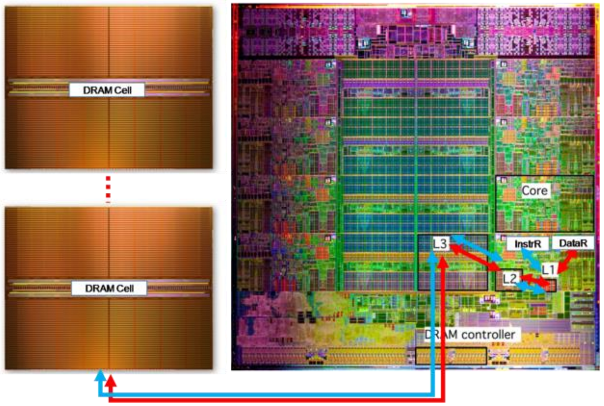
$$P_{peak(noSIMD,onecore)} = 4[GHz]\cdot 1[Cores]\cdot 2[SIMD_{units}]\cdot 1[SIMD_{width}]\cdot 1[Flop] = 8[GFlops/s]$$

$$I_{machine} = \frac{256/32\,[GFlops/s]}{25.6/2\,[GBytes/s]} = 0.6\frac{Flops}{Bytes} \qquad \min\left(1, \frac{I_{code}}{I_{machine}}\right) = \min\left(1, \frac{0.04}{0.6}\right) = 0.07$$



```
Vector Addition:
do i=1,N
  A[i] = B[i] + C[i]
enddo
```

Data Sizes



- RISC Instructions
  - Size: 32bit = 4bytes
  - Types: data handling, arithmetic/logic, control flow
- Integer types
  - From 8bit to 64bit, signed and unsigned
  - Unsigned range 8bit: 2**8=256
  - Unsigned range 64bit: 2**64=18,446,744,073,709,551,615
- Floating point types
  - FP64, 64bit, double precision: “standard for scientific computing”
    - 1 – 1/3 = 1.3333333333333333
  - FP32, 32bit, single precision: “standard for computer graphics”
    - 1 – 1/3 = 1.3333334
  - FP16, 16bit, half precision: supported on TPUs/recent GPGPUs
    - 1 – 1/3 = 1.333

Optimization Potential

- Scalar vector product

$$B_{code} = \frac{O(N)\,[Bytes]}{O(N)\,[Flops]} = \frac{3 \cdot 8 \cdot N\,[Bytes]}{2 \cdot N\,[Flops]} = 12\,\frac{Bytes}{Flops}$$

```
Scalar product:
do i=1,N
  A[i] += B[i] * C[i]
enddo
```

- Matrix addition

$$B_{code} = \frac{O(N^2)\,[Bytes]}{O(N^2)\,[Flops]} = \frac{3 \cdot 8 \cdot N^2\,[Bytes]}{1 \cdot N^2\,[Flops]} = 24\,\frac{Bytes}{Flops}$$

```
Matrix addition:
do i=1,N
  do j=1,N
    A[i,j] = B[i,j] + C[i,j]
  enddo
enddo
```

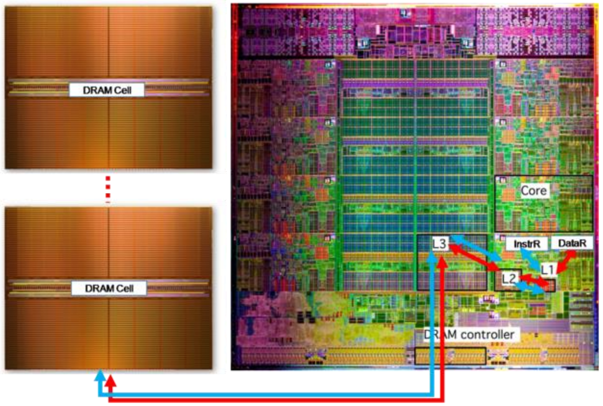
- Matrix multiplication

$$B_{code} = \frac{O(N^2)\,[Bytes]}{O(N^3)\,[Flops]} = \frac{3 \cdot 8 \cdot N^2\,[Bytes]}{2 \cdot N^3\,[Flops]} = 12/N\,\frac{Bytes}{Flops}$$

```
Matrix Multiplication:
do i=1,N
  do j=1,N
    do k=1,N
      A[i,j] += B[k,j] * C[i,k]
    enddo
  enddo
enddo
```

Example: “Vector Triad”

- Smallest data unit: cache line
  - On x86: 64 Bytes (= 8 FP64, =16 FP32)
- Hierarchy of bandwidths/latencies
  - L1/L2/L3/memory
- Performance bounds for “Vector Triad”
  - Large N, non-sequential access, memory bound, latency hidden
  - Large N, sequential access, memory bound, latency hidden
  - Infinite bandwidth, computationally bound



$$runtime = \frac{N \cdot 4 \cdot 64\,[Bytes]}{bandwidth\,[Bytes/s]}$$

$$runtime = \frac{N \cdot 4 \cdot 8\,[Bytes]}{bandwidth\,[Bytes/s]}$$

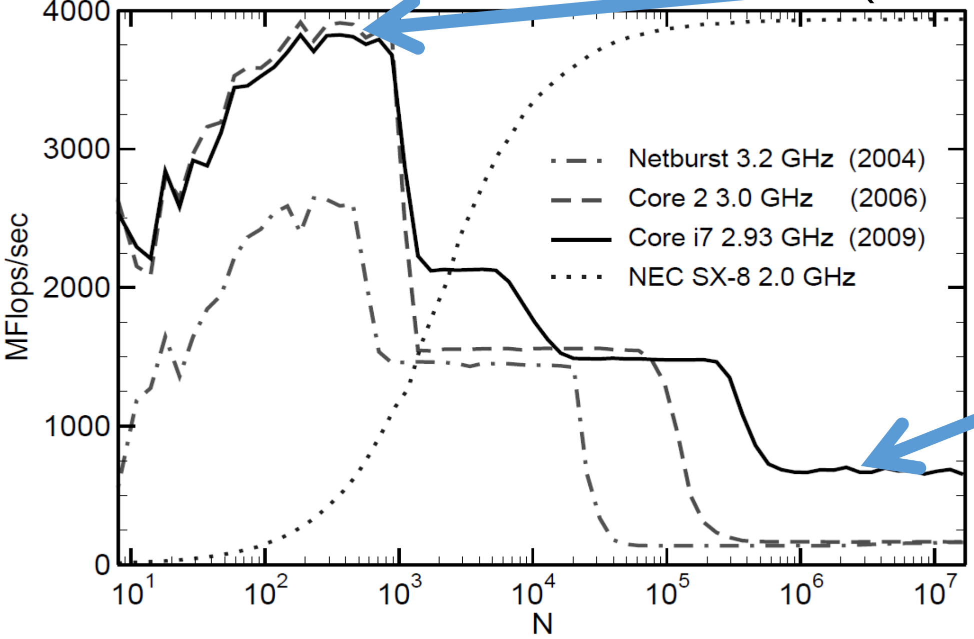
$$runtime = \frac{N \cdot 2\,[Flops]}{peak\ \ performance\,[Flops/s]}$$

```
“Vector Triad”:
do i=1,N
  A[i] = B[i] + C[i] * D[i]
enddo
```

$$B_{triad} = \frac{O(N)\,[Bytes]}{O(N)\,[Flops]} = \frac{4 \cdot 8 \cdot N\,[Bytes]}{2 \cdot N\,[Flops]} = 16\,\frac{Bytes}{Flops}$$

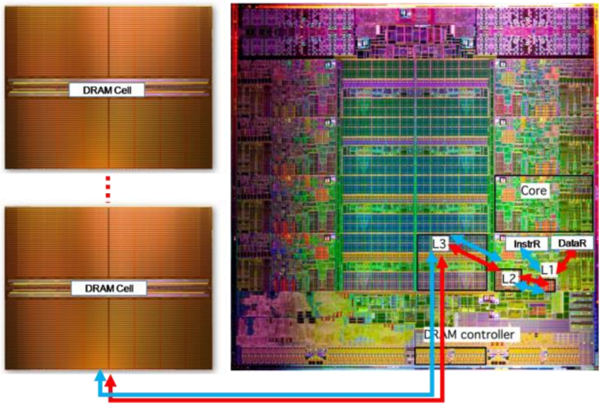
Example: “Vector Triad”

- Results for “Vector Triad”



Wellein,Hager : Intro. to HPC.

Peak perf. ?
(47GFlops/s)



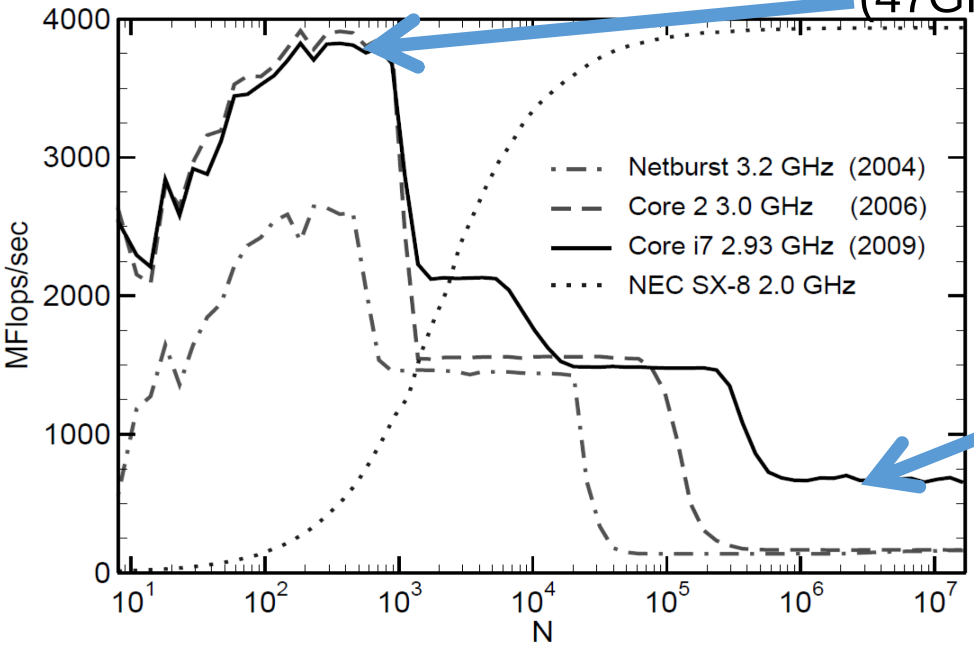
Max. bandwidth ?
(25.6GB/s)

```
“Vector Triad”:
do i=1,N
  A[i] = B[i] + C[i] * D[i]
enddo
```

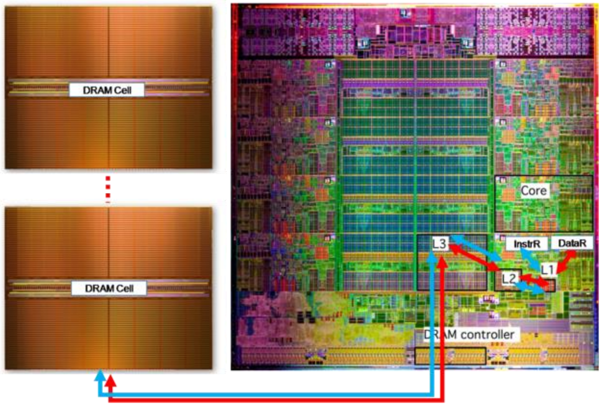
- Interpretation
  - Very small N: pipelines not yet efficient
  - Small N: data contained in L1 cache, cache bandwidth bound
  - Medium N: steps manifest cache bandwidth levels
  - Large N: bound by streaming bandwidth from memory
  - Vector processor: complementary results

Example: “Vector Triad”

- Results for “Vector Triad”



Peak perf. ?
(47GFlops/s)



Max. bandwidth ?
(25.6GB/s)

Wellein,Hager : Intro. to HPC.

“Vector Triad”:

```
do i=1,N
  A[i] = B[i] + C[i] * D[i]
enddo
```

- N~500: 3.8 Gflops/s
  - Effective L1 bandwidth: $3.8\,[GFlops/s] \cdot B_{triad} = 61\,[GBytes/s]$
- N=Large: 0.75Gflops/s
  - Effective memory bandwidth: $0.75\,[GFlops/s] \cdot B_{triad} = 12\,[GBytes/s]$
    (for a single core)

Summary: Performance Modeling

- Smallest data unit: cache line (64bytes)
- Data sizes
  - Instructions (4Bytes)
  - Integral types (1-8Bytes)
  - Floating point types (2-8Bytes)
- Bandwidth-based modeling
  - Model for bandwidth limited code (disk, memory, caches)
  - Machine balance / Code balance
- Optimization potential estimation
  - Ratio of data traffic vs. floating point operations
- Example Code: “Vector Triad”
  - Benchmarking effective bandwidths of a system (single core)

The Optimizing Compiler

- C++ compiler flags for performance optimization
  - “-O0”: no opt., most debugable code
  - “-O2”: enabling most optimization
  - “-O3”: more optimization (longer compile time, pot. larger code)
  - “-O3 -ffast-math”: allows optimizations violating IEEE FP standards
  - “-flto”: enable link time optimization
  - “-march=native”: select compiling machine arch. as target
- Some important automatic compile time optimizations
  - Local optimization
  - Inlining
  - Vectorization
  - Loop optimizations
- Goals of compile time optimizations for performance
  - Optimally utilize target capabilities: Register, pipelines, functional units, caches
  - Avoid performance decrease for any possible run time input

Quiz

- Q1: How to deal with “turbo frequencies” when estimating the peak performance for a CPU?
- Q2: Why did Intel drop AVX-512 from the P-Cores (e.g. i9-13900k)?
- Q3: For very large N, is the performance of a dense matrix-matrix multiplication memory bound or computationally bound?

Quiz (for next Lecture)

- Q4: What are options to discretize the first and second derivatives of a one-dimensional function using discretization points of distance h?

- Q5: Using the trapezoidal rule to integrate

$$A = \int_{-\pi/2}^{+\pi/2} \cos(x)dx$$

does the approximation overestimate or underestimate the integral A?

Next Lecture: C++ Optimization, Finite Difference Method