JÜLICH
FORSCHUNGSZENTRUM

Intel Xeon Phi Knights Landing

Jan H. Meinke

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
FORSCHUNGSZENTRUM

Your free Lunch will soon be over!

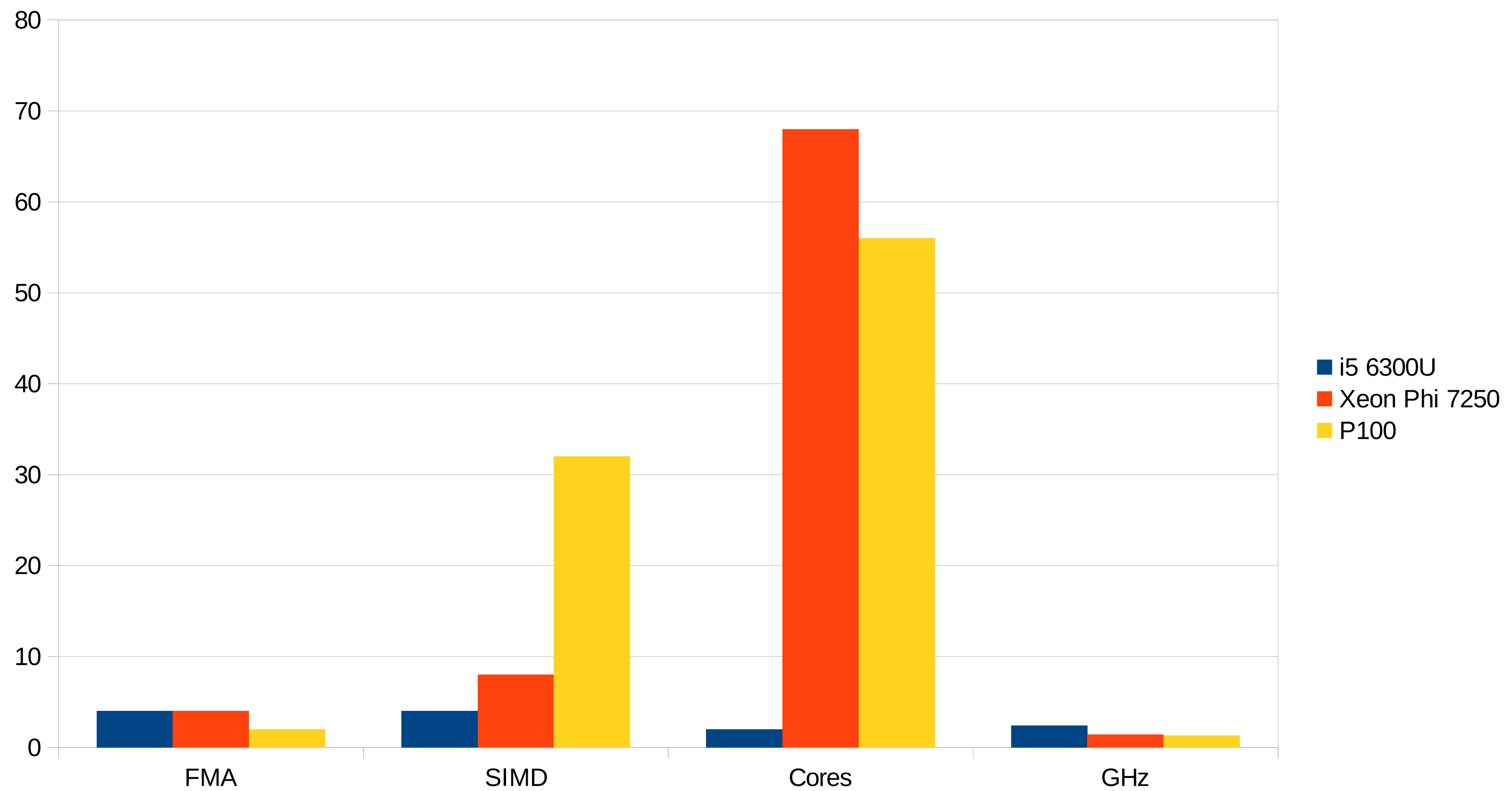
Herb Sutter

Lots of Parallelism

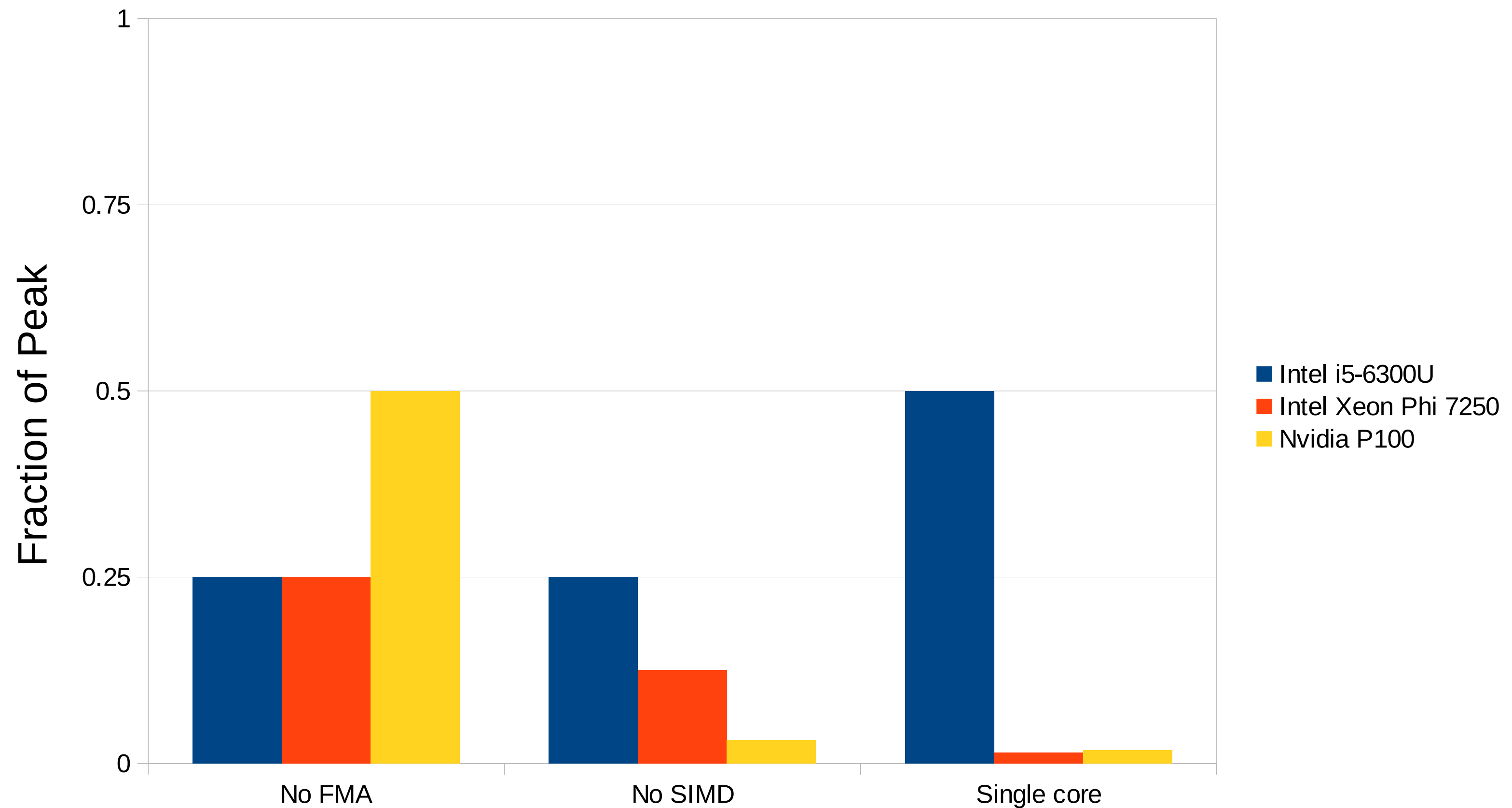
- Instructions per cycle
- Arithmetic and logic units (ALU)
- Vector units
- Cores
- Sockets
- Nodes

JÜLICH FORSCHUNGSZENTRUM

Processor comparison



Floating Point Performance Contributions



What Could This Mean for You?

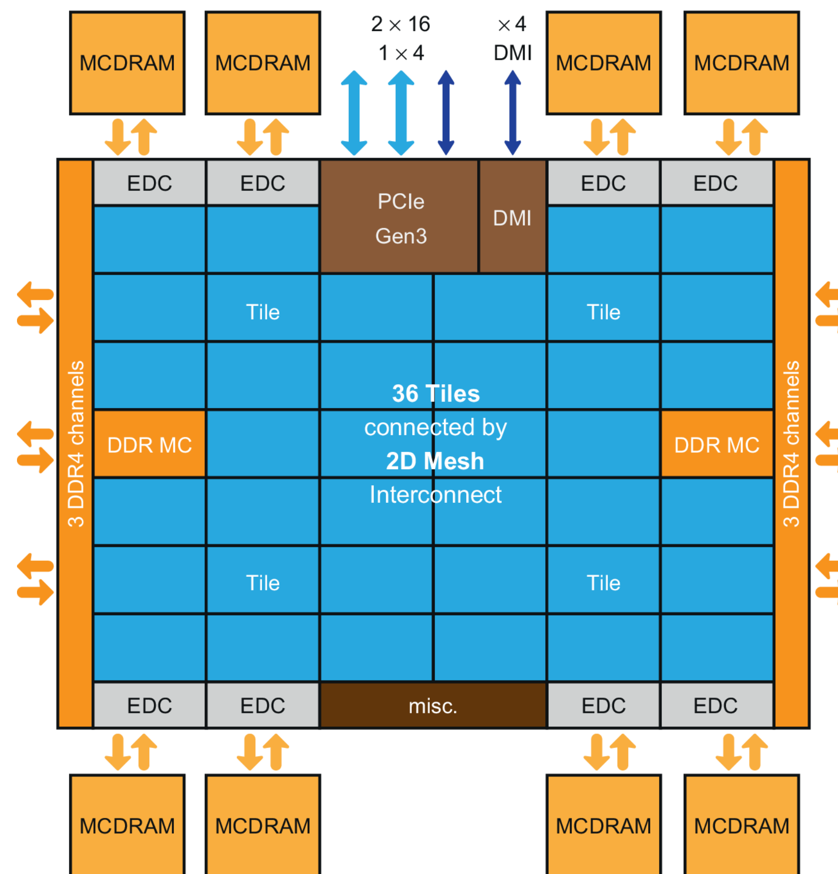
- 12 months of computation (serial, no simd, no FMA)
- → 6 days if computation parallelizes perfectly (68 cores)
- → 17 hours if it can vectorize perfectly, as well (512 bit)
- → 4 hours if it can use FMA

KNL Architecture

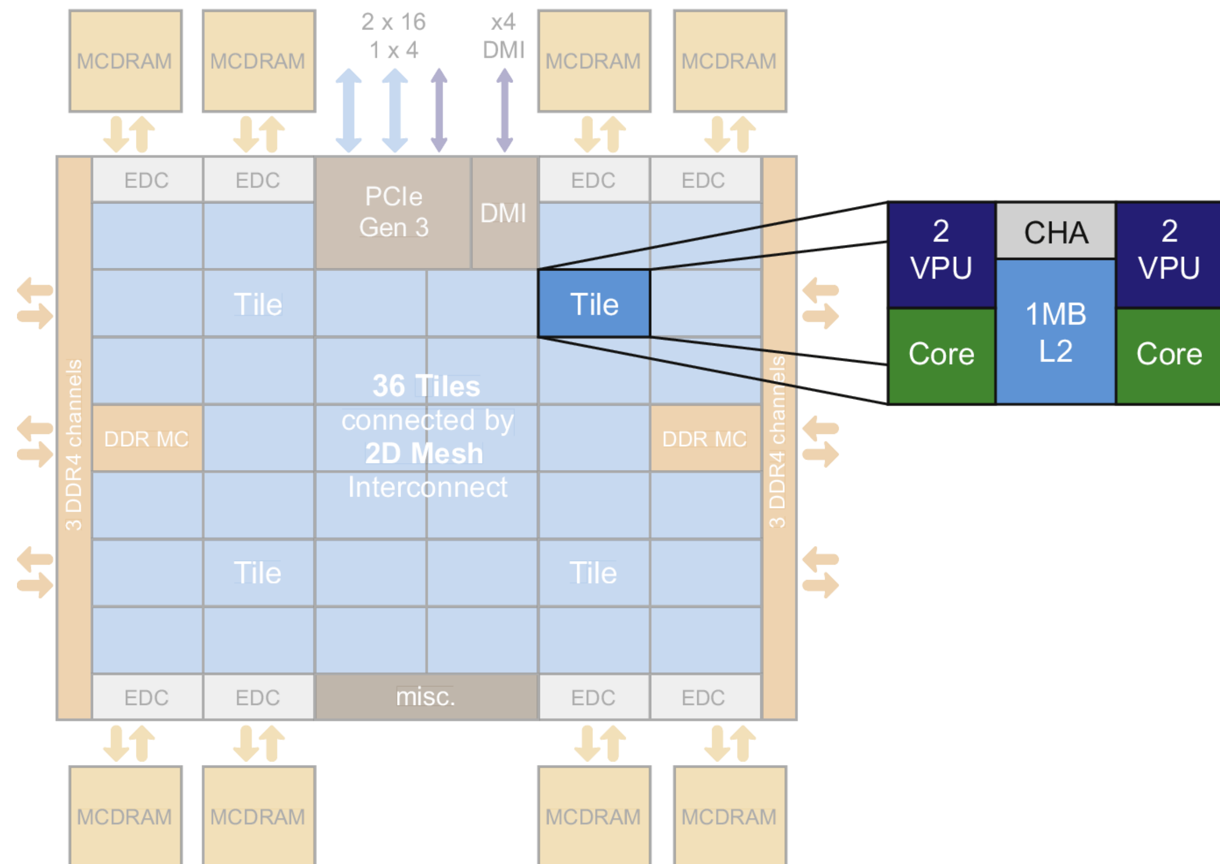
JÜLICH FORSCHUNGSZENTRUM

Mitglied der Helmholtz-Gemeinschaft

Intel Xeon Phi – Architecture

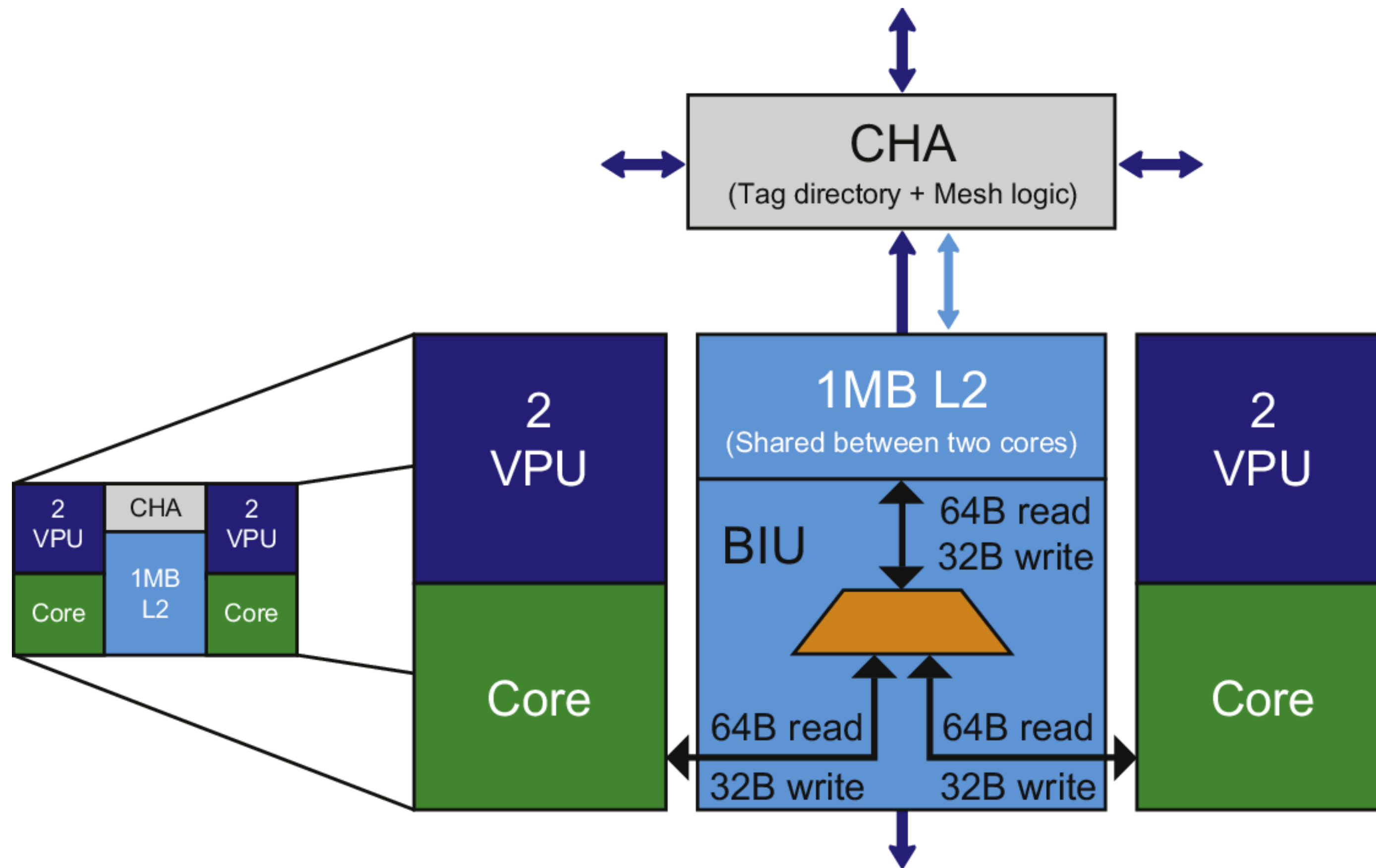


Intel Xeon Phi – Tile



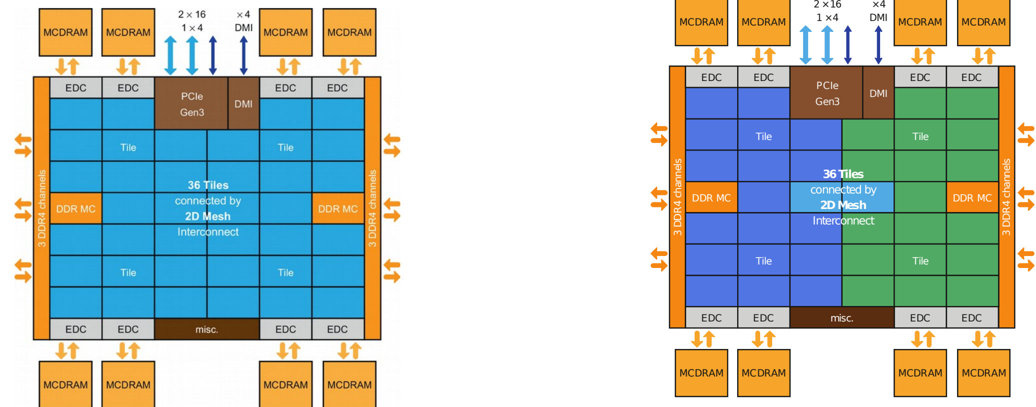
JÜLICH
FORSCHUNGSZENTRUM

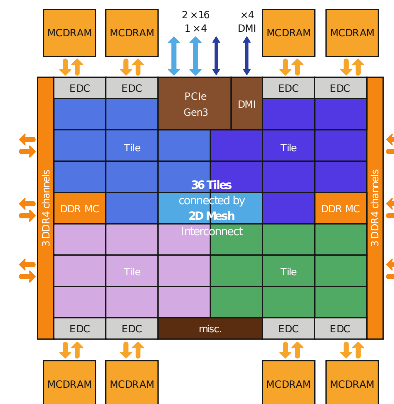
Intel Xeon Phi – Tile



Cluster Modes

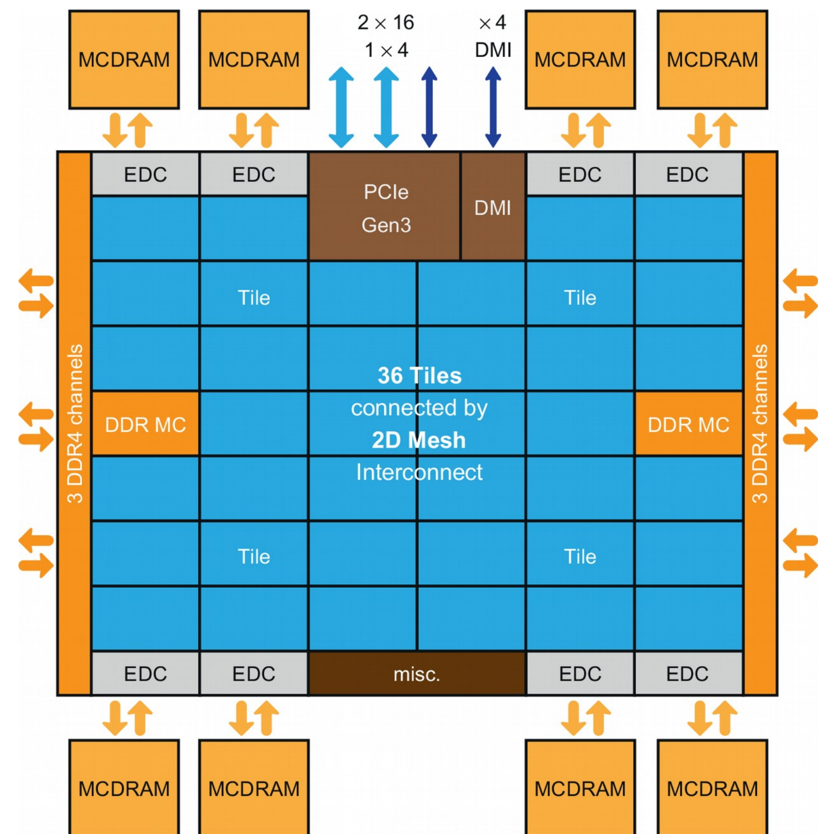
- All-to-all
- Hemisphere
- Quadrant
- SNC-2/4





Cluster Modes: All-to-All

- Any tile
- Any CHA
- Any memory location

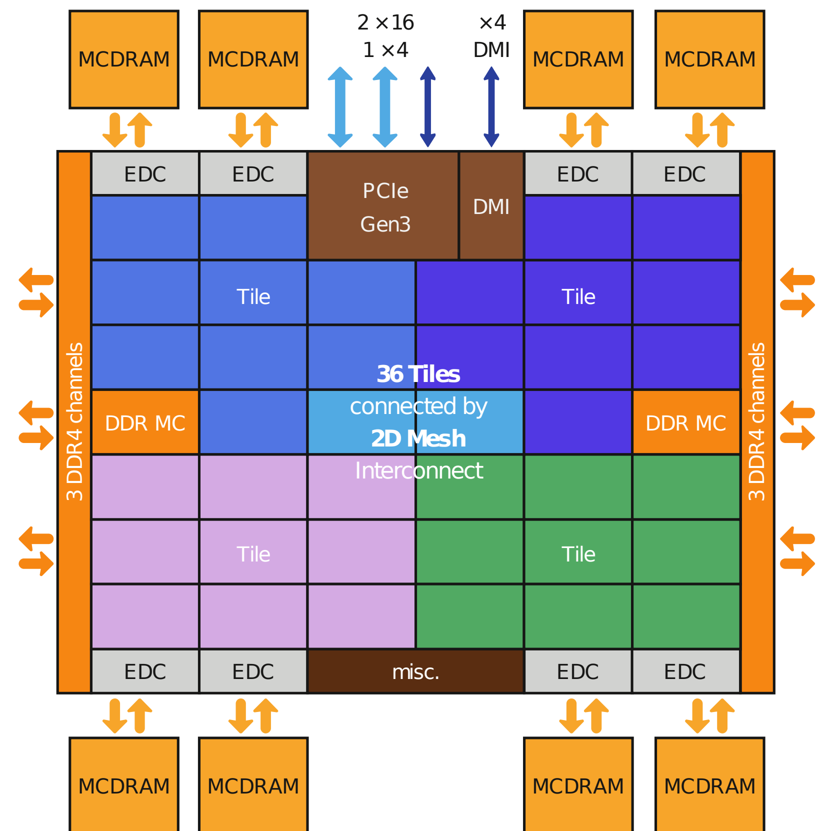


JÜLICH FORSCHUNGSZENTRUM

Cluster Modes: Quadrant/SNC-4

- Any tile
- CHA of the quadrant of the memory location
- Memory in same quadrant as CHA

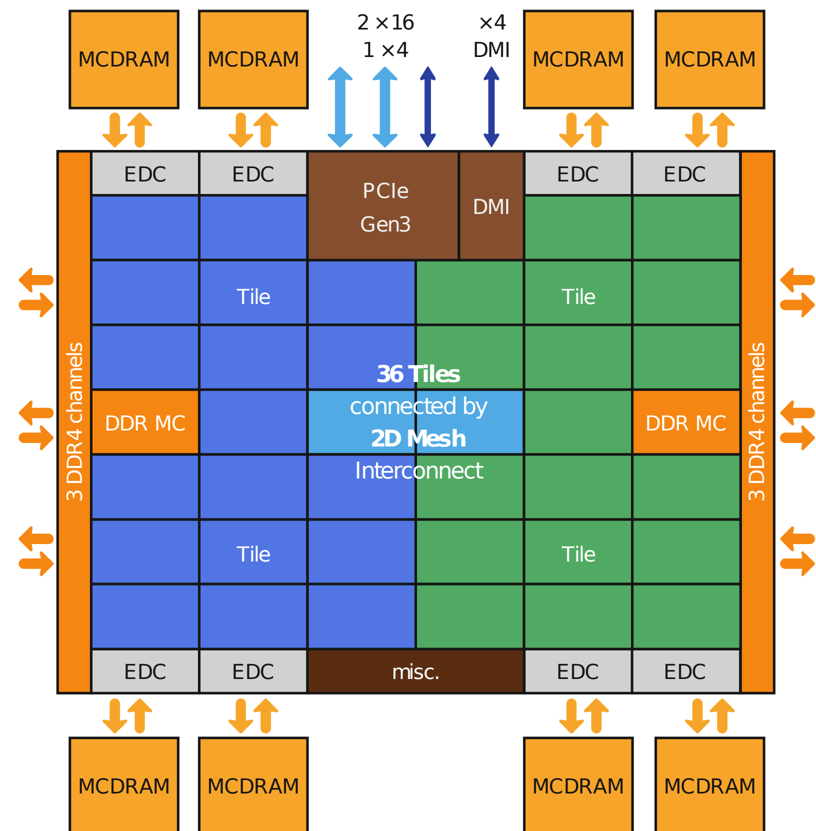
Sub-NUMA-Cluster (SNC): Divide chip into NUMA domains, c.f., multi-socket system. SNC-4 corresponds to a 4-socket node.



Cluster Modes: Hemisphere/SNC-2

- Any tile
- CHA of the hemisphere of the memory location
- Memory in same hemisphere as CHA

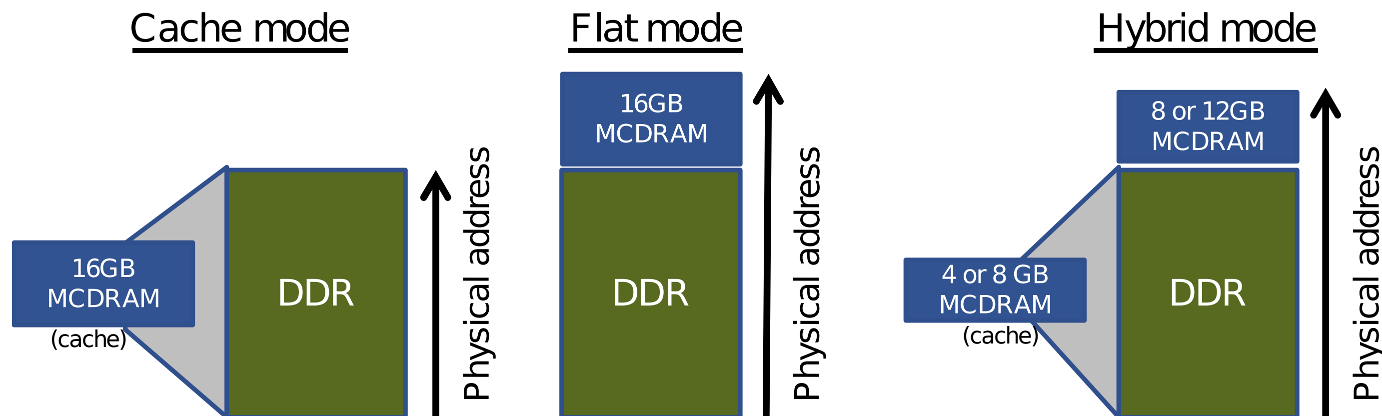
Sub-NUMA-Cluster (SNC): Divide chip into NUMA domains, c.f., multi-socket system. SNC-2 corresponds to a 2-socket node.



JÜLICH
FORSCHUNGSZENTRUM

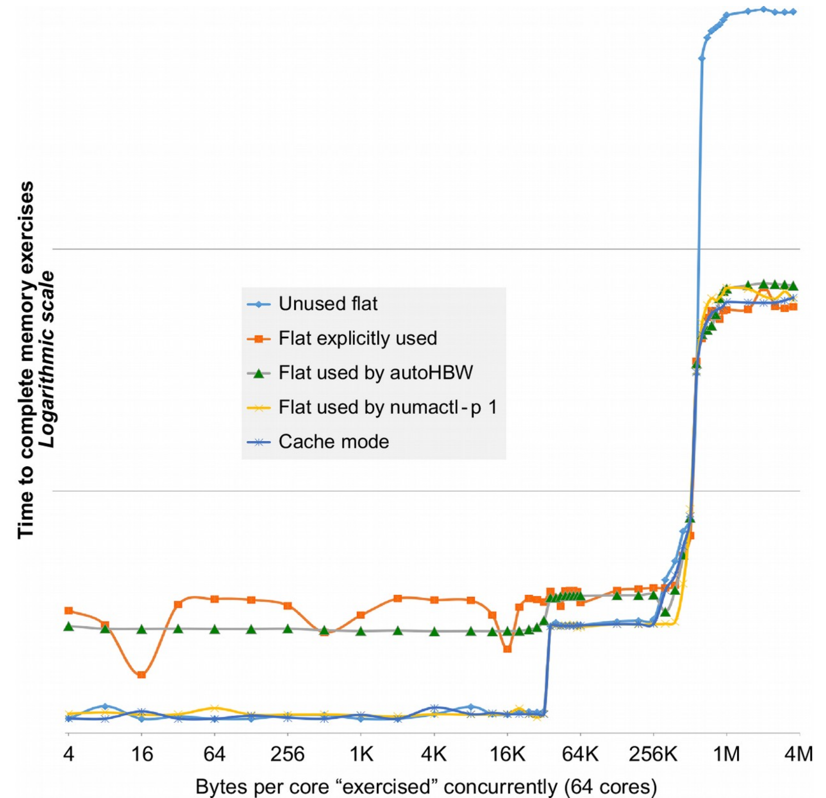
MCDRAM

- 8×2 GB = 16 GB high-bandwidth memory
- Aggregate bandwidth: ~450 GB/s (c.f. 90 GB/s DDR4)
- Lower latency at high load (many accesses)
- 3 modi:



How to Use MCDRAM

- Cache
- Numactl
- hbw_malloc
- …?



Ch 3, Jeffers, Reinders, and Sodani, *Intel Xeon Phi Processor High Performance Programming, 2nd Edition*.

Best Way of taking advantage of KNL

- Optimized applications (for example, Gromacs)
- Optimized libraries (for example, Intel's MKL)

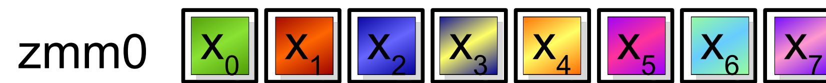
If you find a program or a library that does what you need, use it!

Vectorization

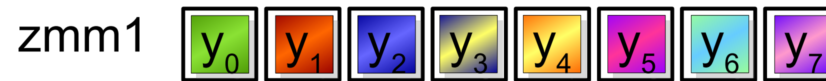
JÜLICH
FORSCHUNGSZENTRUM

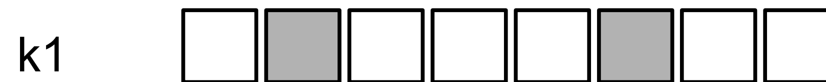
Mitglied der Helmholtz-Gemeinschaft

Vectorization

zmm0


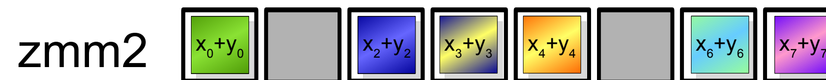
+

zmm1


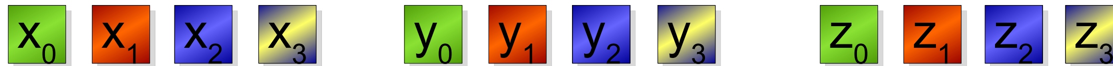
k1


Mask to treat if statement

=

zmm2


Unit stride



ymm0

Array of Structures (C++)

```
struct particle{
    double x;
    double y;
    double z;
};
```

```
std::vector<particle> particles{N};
…
for (int i = 0; i < n; i++){
    for (int j = 0; j < n; j++){
        auto dx = sqrt((particles[j].x – particles[i].x)
                      * (particles[j].x – particles[i]).x);
        …
    }
}
```

Array of Structures (Fortran)

```
type particle
    real :: x
    real :: y
    real :: z
end type particle
```

```
type(particle), dimension(N) :: particles
...
do i, 1, N
    do j, 1, N
        real dx = sqrt((particles(j)%x – particles(i)%x)
                    *(particles(j)%x – particles(i)%x))
        …
    end do
end do
```

Gather

ymm0

JÜLICH FORSCHUNGSZENTRUM

Auto Vectorization

```
for(int i = 0; i < N; ++i){
    c[i] = a * b[i];
}
```

Auto Vectorization

```
#include <vector>
int main(){
    constexpr double a = 2.0;
    constexpr int N= 1024;
    std::vector<double> b(N), c(N);
    for(int i = 0; i < N; ++i){
        c[i] = a * b[i];
    }
}
```

Auto Vectorization

```
#include <vector>
#include <algorithm>
void foo(double a, const std::vector<double>& b,
           std::vector<double>& c){
   int N = std::min(b.size(), c.size());
   for(int i = 0; i < N; ++i){
      c[i] = a * b[i];
   }
}
```

Auto Vectorization

```
void foo(double a,
            double* b,
            double* c,
            int N){
   for(int i = 0; i < N; ++i){
      c[i] = a * b[i];
   }
}
```

Exercise: Vectorization Report

Go to exercises/vectorization. Use

- -qopt-report

and optionally

- -qopt-report-phase=vec,loop,…
- -qopt-report-file=file|stderr|stdout

as additional options to icpc, e.g.,

icpc -qopt-report -qopt-report-phase=vec,loop -qopt-report-file=stderr -O2 -xhost -o simple_loop_in_main simple_loop_in_main.cpp

to compile simple_loop_in_main.cpp and generate the report for vectorization and loop transformation.

> To make sure you compile with all optimization for KNL use *-xmic-avx512* instead of *-xhost*

JÜLICH FORSCHUNGSZENTRUM

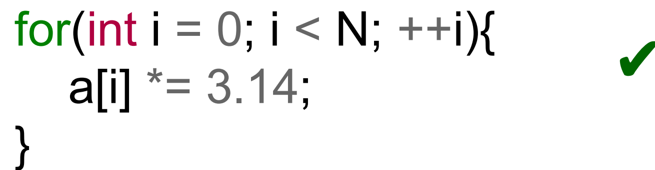
Prerequisites for Vectorization

- Countable
- Single code path
  - No exceptions
  - No jumps
  - No branches (conditional assignment OK)
- No backwards loop-carried dependencies
- Innermost loop of nested loops

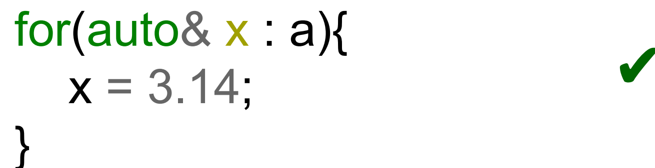
Vectorization Quiz

The vector a is defined in the same function and has size N:

```
std::vector<double> a(N);
```



✔



✔

Vectorization Quiz

The vector a is defined in the same function and has size N:

```
std::vector<double> a(N);
```

```
std::for_each(
    a.begin(), a.end(),
    [=](auto& x){
        x *= N / (3.14 * 3.14);
});
```

```
for(int i = 0; i < N; ++i){
    a[i] /= N;
    if (i % 7 == 0){
        i += 7;
    }
}
```

Vectorization Quiz

The vector a is defined in the same function and has size N:

```
std::vector<double> a(N);

// Flow dependency (read after write)
for(int i = 2; i < N; ++i){
    a[i] = a[i - 1] + a[i - 2];
}
```

```
// Anti-dependency (write after read)
for(int i = 0; i < (N - 2); ++i){
    a[i] = a[i + 1] + a[i + 2];
}
```

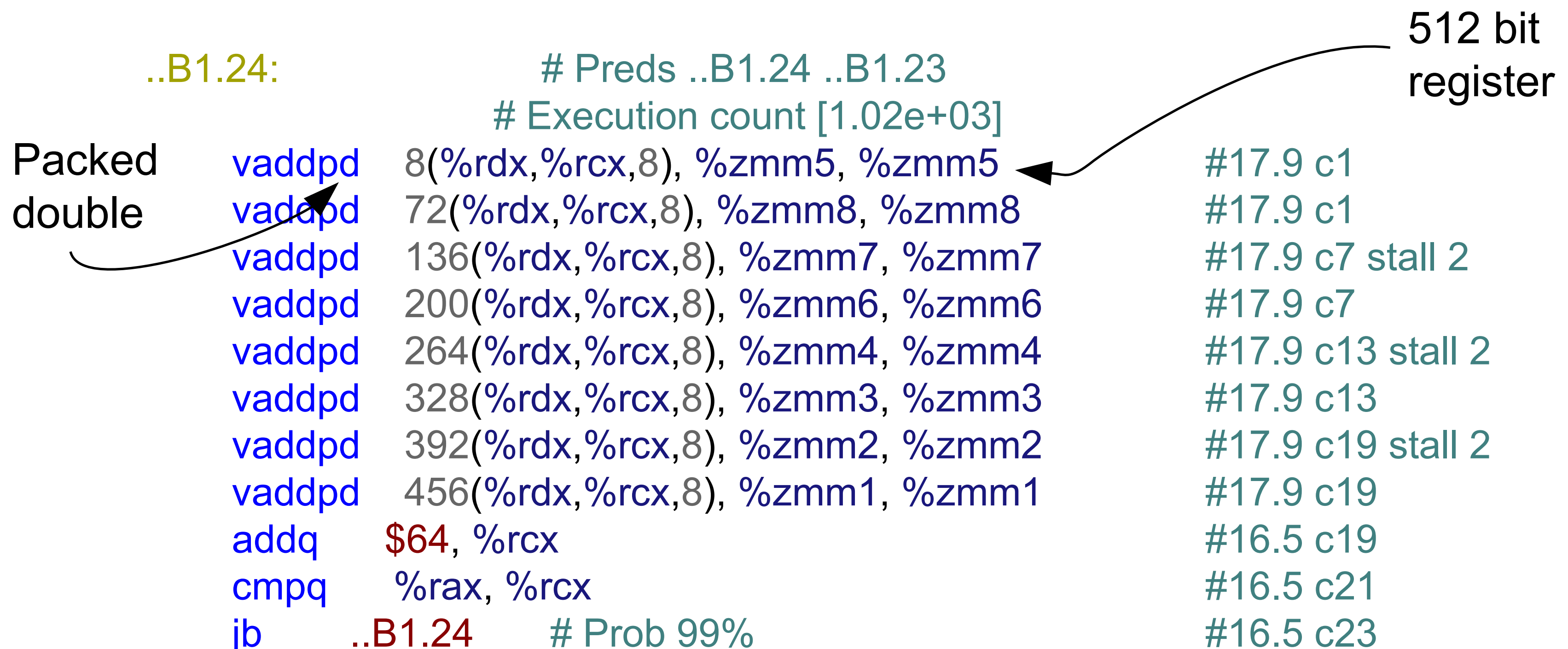
```
// Reduction
auto sum = a[0];
for(int i = 1; i < N; ++i){
    sum += a[i];
}
```

Look at the Assembler–the Hard Way

Compile with -S:

icpc -std=c++14 -O2 -xhost -ansi-alias -fargument-noalias -c order.cpp -S

This generates a file order.s that contains the assembler.



JÜLICH
FORSCHUNGSZENTRUM

Cost Estimate for Vectorization

Operation	Cost	Example
Simple math	1	A*B+C
Load (w/ cache-line split)	1 (2)	A[i]
Store (w/ cache-line split)	1 (2)	A[i] = 2;
Gather (scatter) 8 elements	15 (20)	A[column[i]]
Gather (scatter) 16 elements	20 (25)	A[column[i]]
Horizontal reduction	30	sum += A[i]
Division or square root	15	A/B

Fig. 6.8 from Intel Xeon Phi High Performance Programming [1]

Cost Estimate for Vectorization

```
std::vector<double> v(N, 0.0);
…
double sum = 0.0;
for(auto e : v)
    sum += e;
```

Operation	Cost	Example
Simple math	1	A*B+C
Load (w/ cache-line split)	1 (2)	A[i]
Store (w/ cache-line split)	1 (2)	A[i] = 2;
Gather (scatter) 8 elements	15 (20)	A[column[i]]
Gather (scatter) 16 elements	20 (25)	A[column[i]]
Horizontal reduction	30	sum += A[i]
Division or square root	15	A/B

Serial cost:
N * (1(L) + 1(A)) = 2N

Vector cost:
N/8 * (1(L)) + 30 (HR)=N/8 + 30

$$2N_t = N_t/8 + 30 \Leftrightarrow 1.875\, N_t = 30 \Leftrightarrow N_t = 30/1.875 \approx 16$$

For N<16, a serial loop is better. For N>16 a vectorized loop is faster.

JÜLICH
FORSCHUNGSZENTRUM

But I know my loop should vectorize!

Force Vectorization

```
#include <math.h>
void force(double* x, double* y, double* z, double* m,
        double* Fx, double* Fy, double* Fz,
        int N, int i){
    auto G = 1.0;

    for(int j=0; j < N; ++j){
        auto rD = 1. / sqrt((x[j]-x[i])*(x[j]-x[i])+
                    (y[j]-y[i])*(y[j]-y[i])+
                    (z[j]-z[i])*(z[j]-z[i]));
        Fx[i] += G*m[i]*m[j] * rD*rD*rD* (x[j]-x[i]);
        Fy[i] += G*m[i]*m[j] * rD*rD*rD* (y[j]-y[i]);
        Fz[i] += G*m[i]*m[j] * rD*rD*rD* (z[j]-z[i]);
    }
}
```

```
icpc -std=c++14 -O2 -xhost -ansi-alias -qopt-report=2 -qopt-report-phase=vec,loop -qopt-report-file=stderr -c omp_simd.cpp

LOOP BEGIN at omp_simd.cpp(7,5)

  remark #15344: loop was not vectorized: vector dependence prevents vectorization. First dependence is shown below. Use level 5 report for details

  remark #15346: vector dependence: assumed ANTI dependence between x[j] (8:24) and Fz[i] (13:9)

LOOP END
```

Force Vectorization

```
#include <cmath>
void force(double* x, double* y, double* z, double* m,
        double* Fx, double* Fy, double* Fz,
        int N, int i){
    auto G = 1.0;
#pragma omp simd aligned(x, y, z, Fx, Fy, Fz, m : 32)
    for(int j=0; j < N; ++j){
        auto rD = 1. / sqrt((x[j]-x[i])*(x[j]-x[i])+
                    (y[j]-y[i])*(y[j]-y[i])+
                    (z[j]-z[i])*(z[j]-z[i]));
        Fx[i] += G*m[i]*m[j] * rD*rD*rD* (x[j]-x[i]);
        Fy[i] += G*m[i]*m[j] * rD*rD*rD* (y[j]-y[i]);
        Fz[i] += G*m[i]*m[j] * rD*rD*rD* (z[j]-z[i]);
    }
}
```

```
icpc -std=c++14 -O2 -xhost -ansi-alias -qopt-report=2 -qopt-report-phase=vec,loop -qopt-report-file=stderr -c omp_simd.cpp -qopenmp

LOOP BEGIN at omp_simd.cpp(7,5)
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
LOOP END

...
```

Syntax of OMP SIMD

C++

```
#pragma omp simd [clause[[,] clause] … ] new-line
    For-loop
```

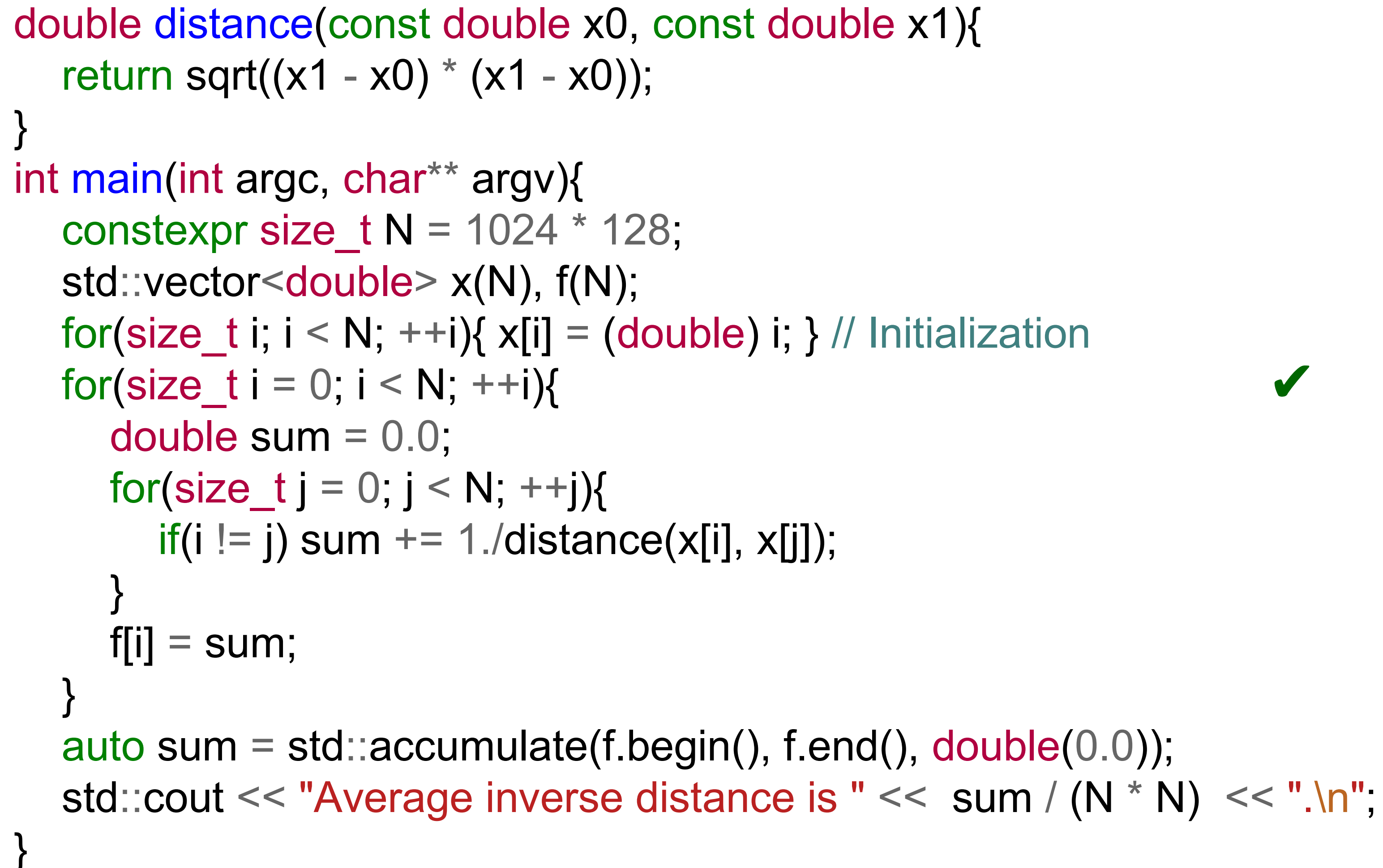
Fortran

```
!$omp simd [clause[[,] clause] … ]
    Do-loops
[!$omp end simd]
```

Available clauses are:

- safelen(length)
- simdlen(length)
- linear(list[:linear step])
- aligned(list[:alignment in bytes])
- private(list)
- lastprivate(list)
- reduction(type : list)
- collapse(n)

Vectorized User Functions



JÜLICH
FORSCHUNGSZENTRUM

Vectorized User Functions

distance.h

```
#ifndef _DISTANCE_H_
/** Calculates the distance between x0 and x1
 * @param x0 position of first point
 * @param x1 position of second point
 * @return distance between x0 and x1
 */

double distance(const double x0, const double x1);
#endif
```

distance.cpp:

```
#include “distance.h”
double distance(const double x0, const double x1){
    return sqrt((x1 - x0) * (x1 - x0));
}
```

Separate header and implementation.

Vectorized User Functions

```
#include "distance.h"
int main(int argc, char** argv){
    constexpr size_t N = 1024 * 128;
    std::vector<double> x(N), f(N);
    for(size_t i; i < N; ++i){ x[i] = (double) i; } // Initialization

    for(size_t i = 0; i < N; ++i){
        double sum = 0.0;

        for(size_t j = 0; j < N; ++j){
            if(i != j) sum += 1./distance(x[i], x[j]);
        }
        f[i] = sum;
    }
    auto sum = std::accumulate(f.begin(), f.end(), double(0.0));
    std::cout << "Average inverse distance is " << sum / (N * N) << ".\n";
}
```

```
Non-optimizable loops:

LOOP BEGIN at user_function.cpp(16,5)
  remark #15333: loop was not vectorized:
exception handling for a call prevents vectorization
[ user_function.cpp(21,27) ]

  LOOP BEGIN at user_function.cpp(19,9)
    remark #15333: loop was not vectorized:
exception handling for a call prevents vectorization
[ user_function.cpp(21,27) ]
  LOOP END
LOOP END
```

Vectorized User Functions

distance.h

```
#ifndef _DISTANCE_H_
/** Calculates the distance between x0 and x1
 * @param x0 position of first point
 * @param x1 position of second point
 * @return distance between x0 and x1
 */
#pragma omp declare simd simdlen(4)
double distance(const double x0, const double x1);
#endif
```

Use *omp declare simd* to let compiler know that function has no side effects.

Vectorized User Functions

```cpp
#include "distance.h"
int main(int argc, char** argv){
    constexpr size_t N = 1024 * 128;
    std::vector<double> x(N), f(N);
    for(size_t i; i < N; ++i){ x[i] = (double) i; } // initialization

    for(size_t i = 0; i < N; ++i){
        double sum = 0.0;

        for(size_t j = 0; j < N; ++j){
            if(i != j) sum += 1./distance(x[i], x[j]);
        }
        f[i] = sum;
    }
    auto sum = std::accumulate(f.begin(), f.end(), double(0.0));
    std::cout << "Average inverse distance is " << sum / (N * N) << ".\n";
}
```

```
LOOP BEGIN at user_function.cpp(16,5)
   remark #15541: outer loop was not auto-vectorized: consider using SIMD directive

   LOOP BEGIN at user_function.cpp(19,9)
      remark #15344: loop was not vectorized: vector dependence prevents vectorization
      remark #15346: vector dependence: assumed ANTI dependence between sum (21:17) and sum (21:17)
      remark #15346: vector dependence: assumed FLOW dependence between sum (21:17) and sum (21:17)
      remark #15346: vector dependence: assumed ANTI dependence between sum (21:17) and sum (21:17)
   LOOP END
LOOP END
```

JÜLICH FORSCHUNGSZENTRUM

Vectorized User Functions

```cpp
#include “distance.h”
int main(int argc, char** argv){
    constexpr size_t N = 1024 * 128;
    std::vector<double> x(N), f(N);
    for(size_t i; i < N; ++i){ x[i] = (double) i; } // Initialization

    for(size_t i = 0; i < N; ++i){
        double sum = 0.0;
        #pragma omp simd reduction(+:sum)
        for(size_t j = 0; j < N; ++j){
            if(i != j) sum += f.distance(x[i], x[j]);
        }
        f[i] = sum;
    }
    auto sum = std::accumulate(f.begin(), f.end(), double(0.0));
    std::cout << "Average inverse distance is " << sum / (N * N) << ".\n";
}
```

```
LOOP BEGIN at user_function.cpp(16,5)
   remark #15542: loop was not vectorized: inner loop
was already vectorized

   LOOP BEGIN at user_function.cpp(19,22)
      remark #15301: OpenMP SIMD LOOP WAS
VECTORIZED
   LOOP END
LOOP END
```

How to Get Aligned Memory

C/C++

- __declspec(align(32)) double p[N]; // Windows
- __attribute__((align(32))) double p[N]; // Linux, OSX (gcc)
- tbb::cache_aligned_allocator<>
- _mm_malloc()

Fortran

- !dir$ attributes align : 64 :: A, B

JÜLICH
FORSCHUNGSZENTRUM

How to tell the compiler memory is aligned

C/C++

- #pragma omp simd aligned(p:32)
- __assume_aligned(p, 32);

Fortran

- !$omp simd aligned(p:32)
- !dir$ assume_aligned p:32

JÜLICH
FORSCHUNGSZENTRUM

Additional Resources for Vectorization

- A Guide to Vectorization with Intel C++ Compilers, https://software.intel.com/sites/default/files/8c/a9/CompilerAutovectorizationGuide.pdf
- NERSC guide to vectorization: http://www.nersc.gov/users/computational-systems/cori/application-porting-and-performance/vectorization/
- OpenMP SIMD examples can be found at OpenMP Examples, https://github.com/OpenMP/Examples

Exercise: Vectorization with User Functions

Look at exercises/vectorization/README.rst and follow the instructions for the second exercise.

Parallel Programming

JÜLICH
FORSCHUNGSZENTRUM

Mitglied der Helmholtz-Gemeinschaft

(Shared-Memory) Parallel Programming

- **OpenMP**
- Fortran 2008 do concurrent
- **Intel TBB**
- PGAS (UPC, Fortran coarrays, ...)
- MPI

Loop semantics

C++

```
for(int i = 0; i < n; ++i){
    f(i);
}
```

Fortran

```
do i = 0, n - 1
    f(i)
end do
```

This is a serial loop! Execute as follows:

i = 0: f(i)
i = 1: f(i)
i = 2: f(i)
…
i = n - 1: f(i)

Loop semantics

C++

```
#pragma omp parallel for
for(int i = 0; i < n; ++i){
    f(i);
}
```

Fortran

```
!$omp parallel do
do i = 0, n - 1
    f(i)
end do
```

This is a parallel loop! Can be executed as follows:
i = 0: f(i) ‖ i = 1: f(i) ‖ i = 2: f(i) ‖ … ‖ i = n - 1: f(i)

No guarantee that the semantics remains the same.
Correctness is responsibility of the programmer.

Loop semantics

Fortran

```
do concurrent (i = 0: n - 1)
   f(i)
end do
```

This is a parallel loop! Can be executed as follows:
i = 0: f(i) ‖ i = 1: f(i) ‖ i = 2: f(i) ‖ … ‖ i = n - 1: f(i)

Correctness is responsibility of the programmer.

Use *-parallel* with ifort to parallelize do concurrent

JÜLICH
FORSCHUNGSZENTRUM

Intel® Threading Building Blocks (TBB)

“Intel® TBB is a **C++ template library** that **adds parallel programming for C++ programmers**. It uses **generic programming** to be efficient. Threading Building Blocks includes **algorithms**, highly **concurrent containers**, **locks** and **atomic operations**, a **task scheduler** and a **scalable memory allocator**.”

From https://www.threadingbuildingblocks.org/faq

JÜLICH
FORSCHUNGSZENTRUM

Intel Threading Building Blocks (TBB)

- First made available in 2006
- Open Source in 2007 (GPL v2)
- Since September 2016 under Apache v2.0 license

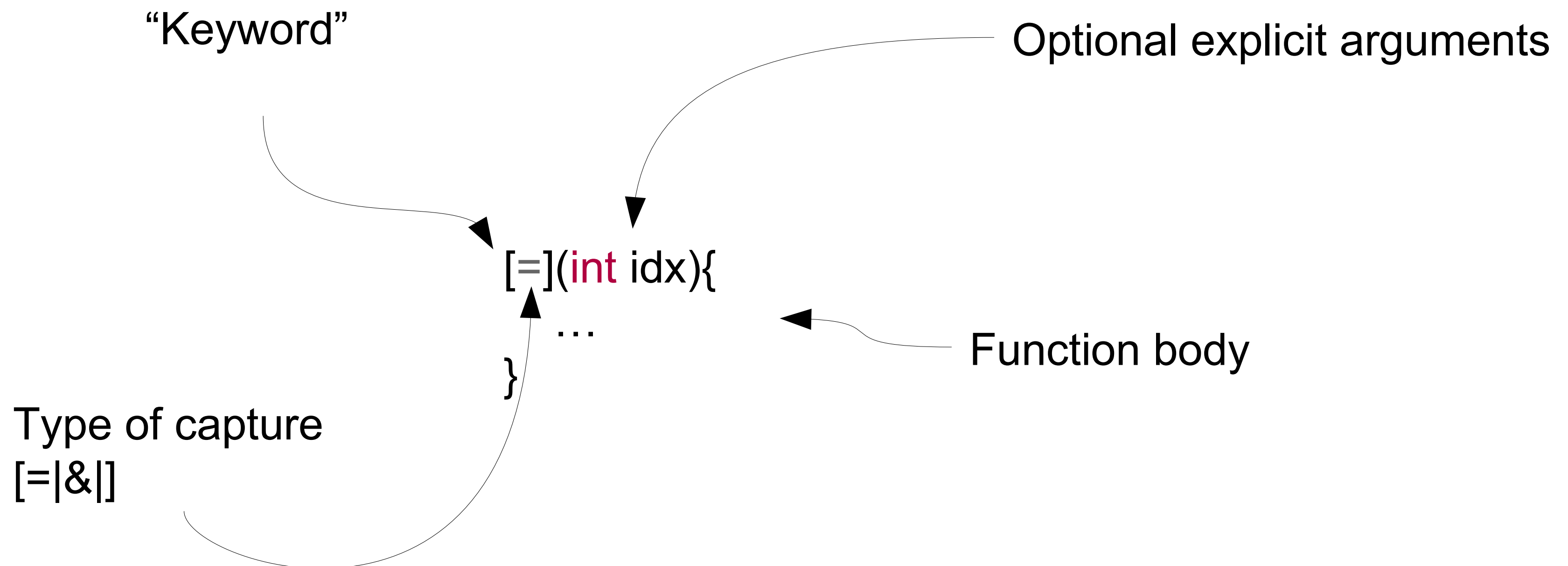
tbb::parallel_for

```
#include <tbb/tbb.h>
...
tbb::parallel_for(0, n, [=](auto i){
    f(i);
});
```

The simplest form is *tbb::parallel_for(first, last, f)*, where f is called for each index. It’s also possible to give a stride.

Short Digression: Lambdas

Lambdas are anonymous functions that can capture variables.



tbb::parallel_for with blocked_range

```
#include <tbb/tbb.h>
...
tbb::parallel_for(0, n, [=](auto i){
    f(i);
});
```

Replace *first* and *last* with a range. The most commonly used range is tbb's blocked_range

tbb::parallel_for with blocked_range

```
#include <tbb/tbb.h>
…
using range_t = tbb::blocked_range<size_t>;
tbb::parallel_for(range_t r(0, n), [=](range_t& r){
    for(auto i = r.begin(); i != r.end(); ++i){
        f(i);
    }
});
```

A *blocked_range* can be subdivided as long as it is bigger than its grain size. A blocked_range of size 30 with grain size 20 can still be divided!

tbb::parallel_for with blocked_range2d

```
#include <tbb/tbb.h>
…
using range_t = tbb::blocked_range2d<size_t>;
tbb::parallel_for(range_t r(0, n, 0, n), [=](range_t& r){
    for(auto i = r.rows().begin(); i != r.rows.end(); ++i){
        for(auto j = r.cols().begin(); j != r.cols.end(); ++j){
            f(i, j);
        }
    }
});
```

There's also a blocked_range3d.

tbb::parallel_reduce with blocked_range.

```
#include <tbb/tbb.h>
…
using range_t = tbb::blocked_range<size_t>;
auto sum = tbb::parallel_reduce(range_t r(0, n),
                  double(0),
                  [=](range_t& r, double in){
                      for(auto i = r.begin(); i != r.end(); ++i){
                          in += i;
                      }
                      return in;
                  },
                  std::sum<double>()
});
```

Range

Identity of reduction

Serial base case

Reduction operation

Example: Average–Initialization (1/4)

```
#include <iostream>
#include <chrono>
#include <random>
#include <numeric>
#include <vector>
#include <tbb/tbb.h>

int main(int argc, char** argv){
    // Get number of elements from first command line argument
    long n = (argc > 1) ? std::stol(argv[1]) : 100000;
```

Example: Average–Random Numbers (2/4)

```
// Setup random numbers
auto r = 1.0;
std::random_device rd;
std::default_random_engine re(rd());
std::uniform_real_distribution<double> uniform(-r, r);

std::vector<double> coordinates(n);
auto gen = std::bind(uniform, re);
std::generate(coordinates.begin(), coordinates.end(), gen);
```

Example: Average–Calculation (3/4)

```
auto start_time = std::chrono::high_resolution_clock::now();
using range_t = tbb::blocked_range<std::vector<double>::iterator>;
auto sum = tbb::parallel_reduce(
        range_t(coordinates.begin(), coordinates.end()), double(0),
        [&](range_t& r, double in){
            return std::accumulate(r.begin(), r.end(), in);
        },
        std::plus<double>()
);
auto stop_time = std::chrono::high_resolution_clock::now();
```

Example: Average–Output (4/4)

```
    std::chrono::duration<double> dt = stop_time - start_time;
    std::cout << "#Parallel average " << (sum / n) << " and took " <<
        dt.count() << " s" << std::endl;

    start_time = std::chrono::high_resolution_clock::now();
    sum = std::accumulate(coordinates.begin(), coordinates.end(), 0.0);
    stop_time = std::chrono::high_resolution_clock::now();
    dt = stop_time - start_time;
    std::cout << "#Serial average is " << (sum / n) << " and took " <<
        dt.count() << " s" << std::endl;
}
```

Exercise: TBB

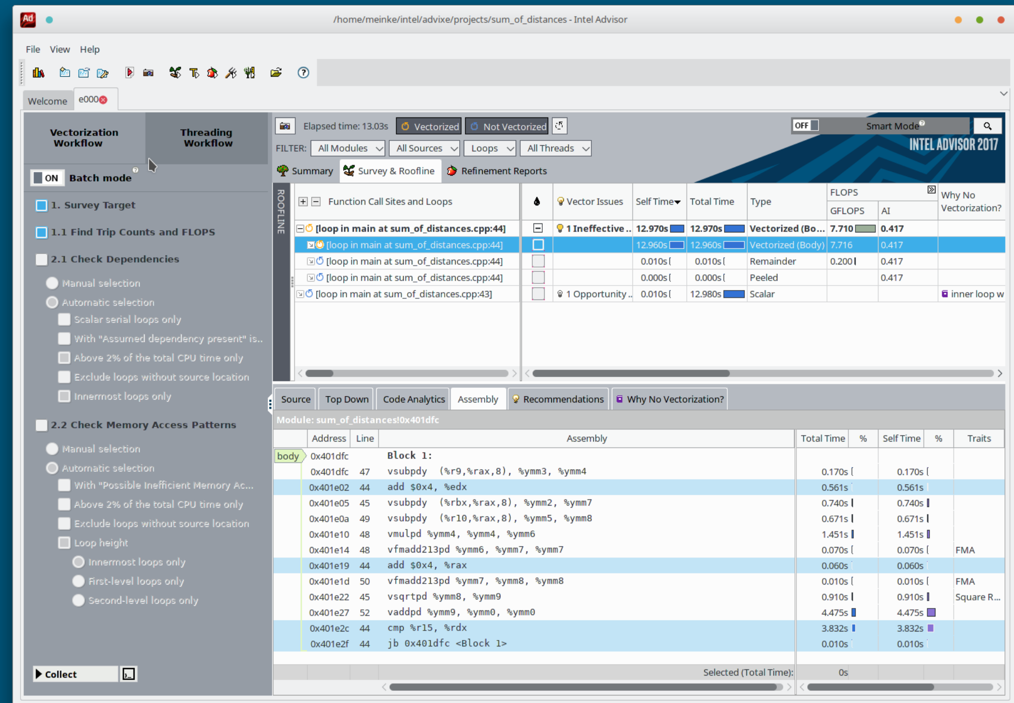
Read, compile and run *average.cpp* in exercises/tbb_intro.

You need to add -ltbb -ltbbmalloc to your compile command.

- Look at the vectorization report
- Compare performance on a “regular” CPU with the performance on a KNL

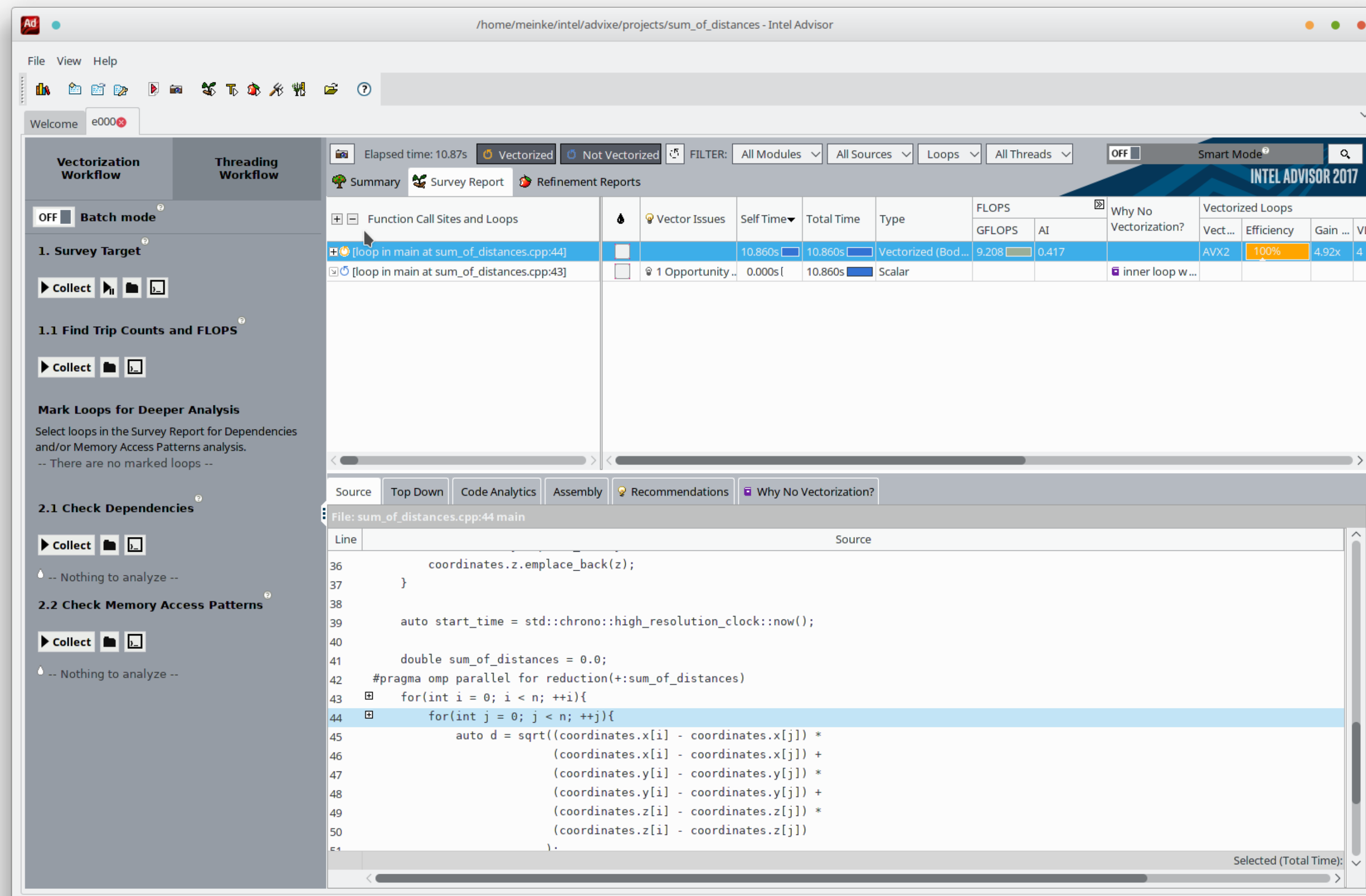
JÜLICH
FORSCHUNGSZENTRUM

Intel Advisor

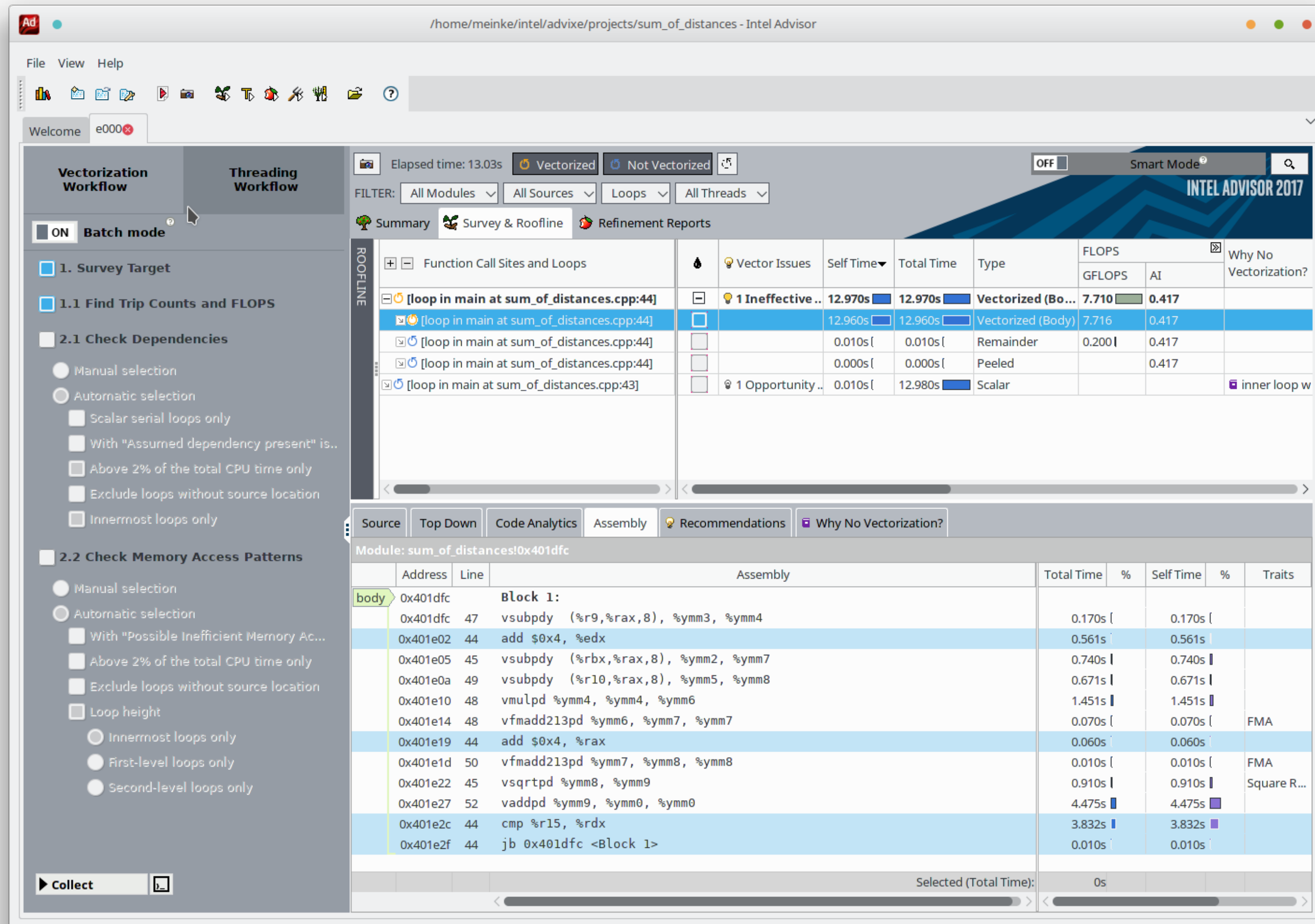


Mitglied der Helmholtz-Gemeinschaft

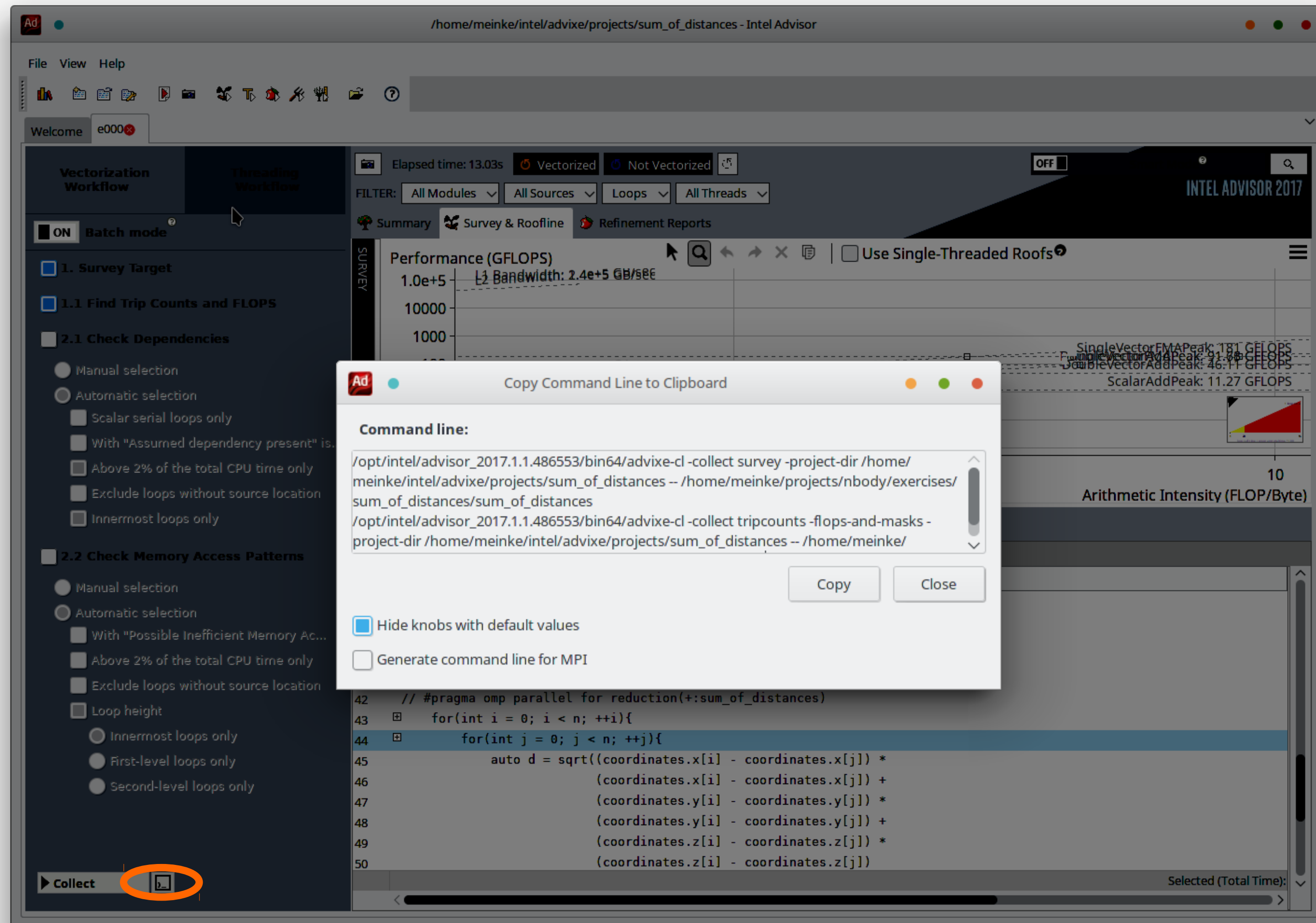
Intel Advisor



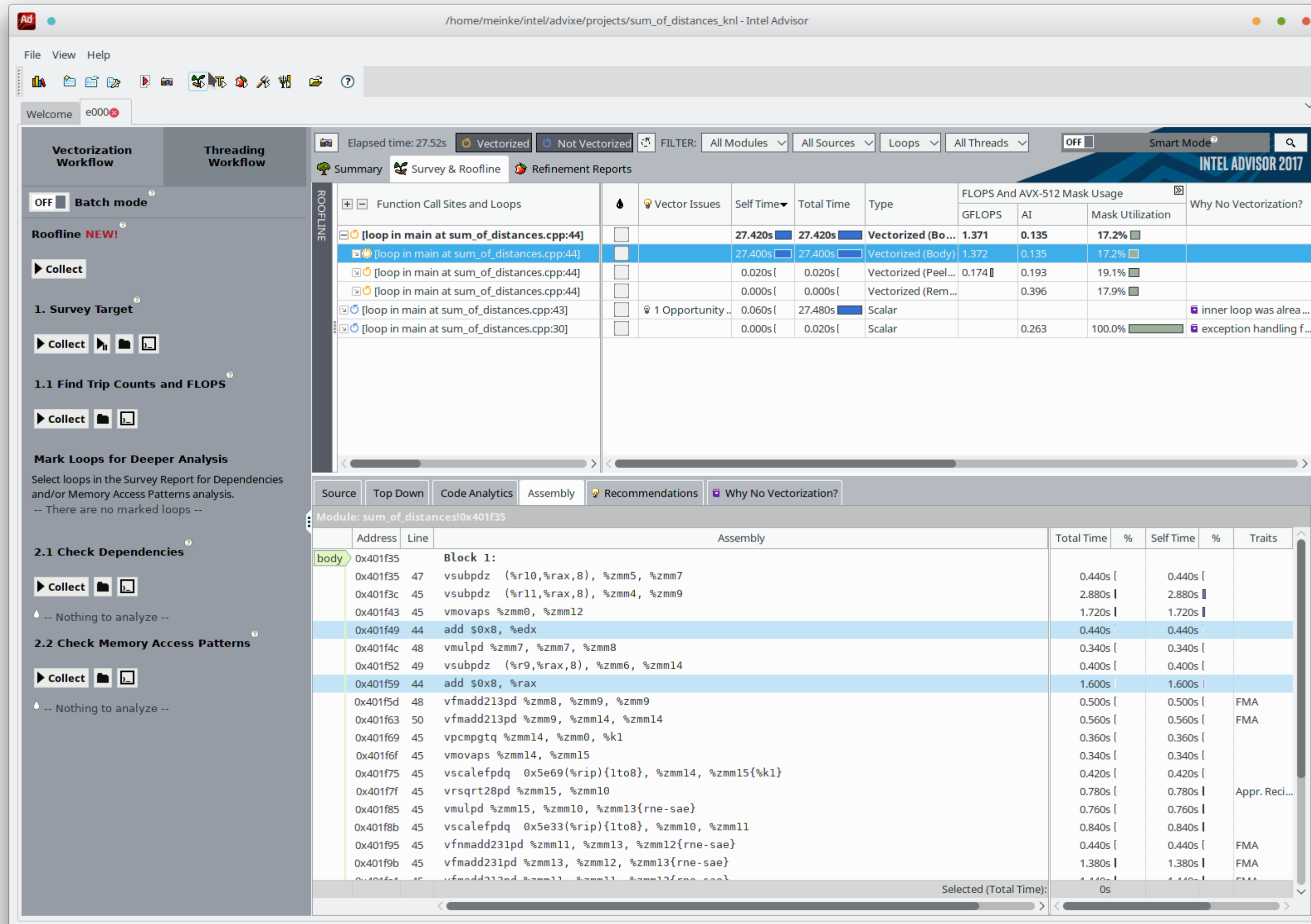
Look at the Assembler–the Easier Way



Intel Advisor



Look at the Assembler–the Easier Way

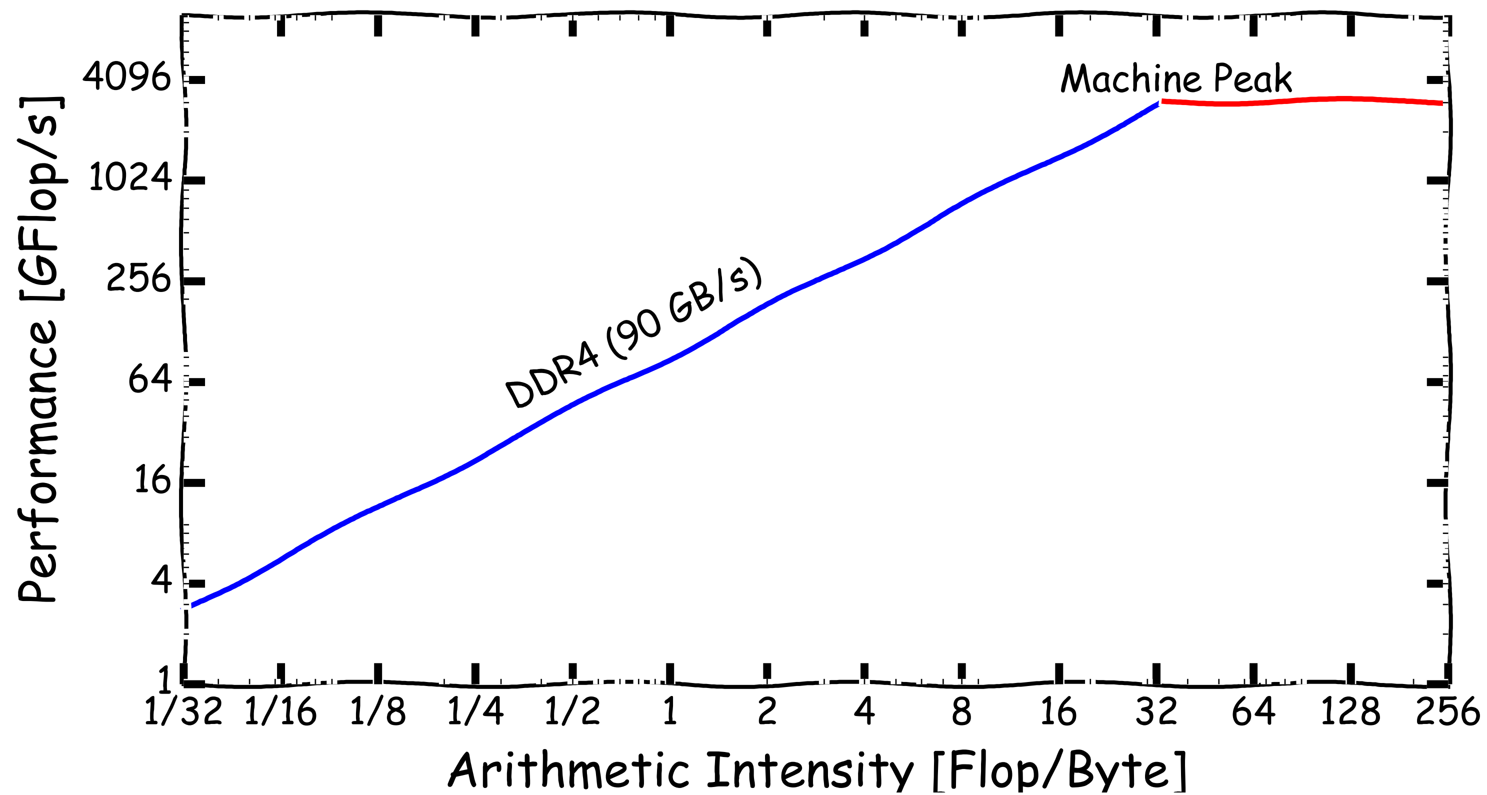


JÜLICH
FORSCHUNGSZENTRUM

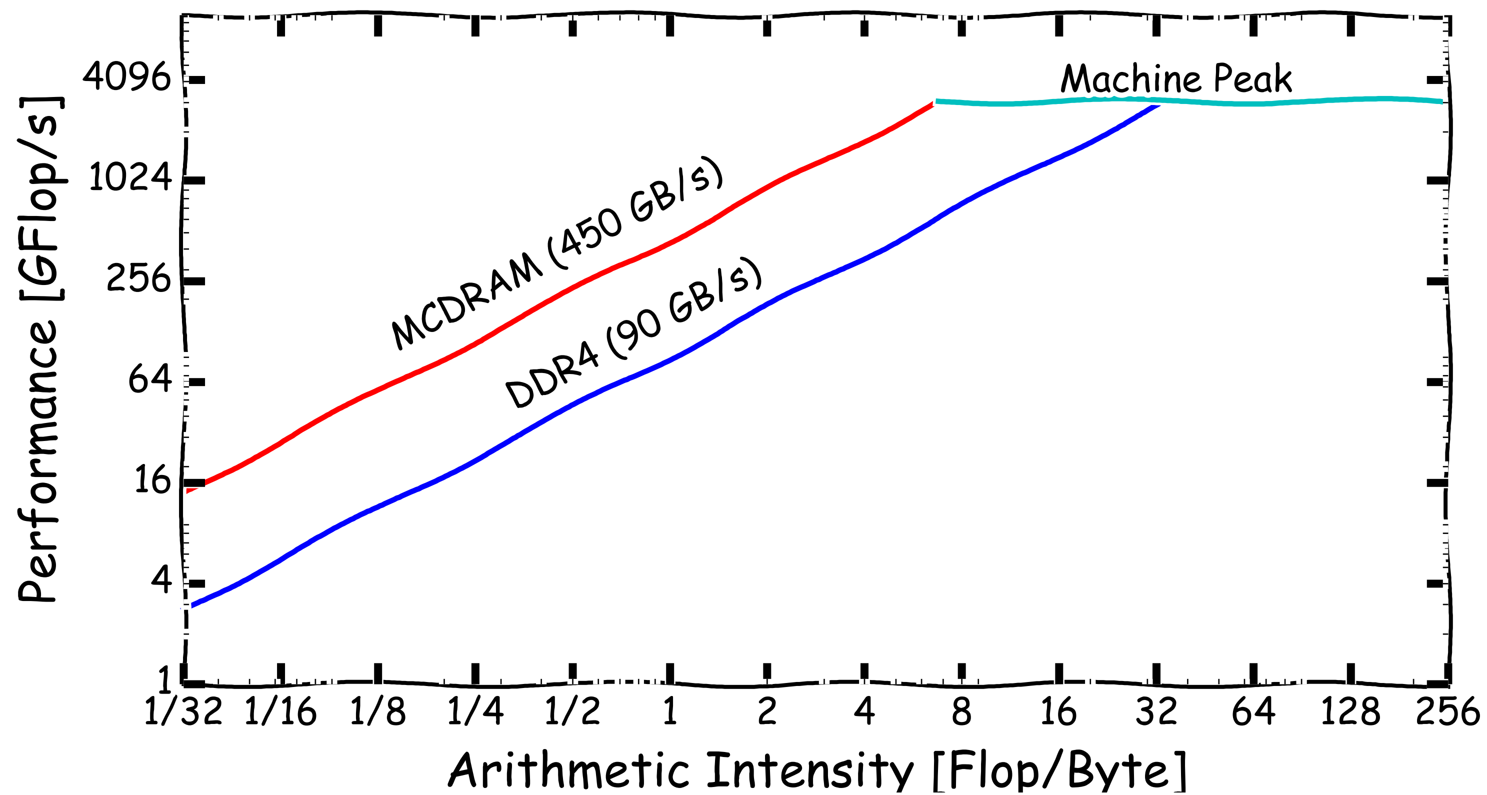
Performance Modeling

Mitglied der Helmholtz-Gemeinschaft

Roofline model

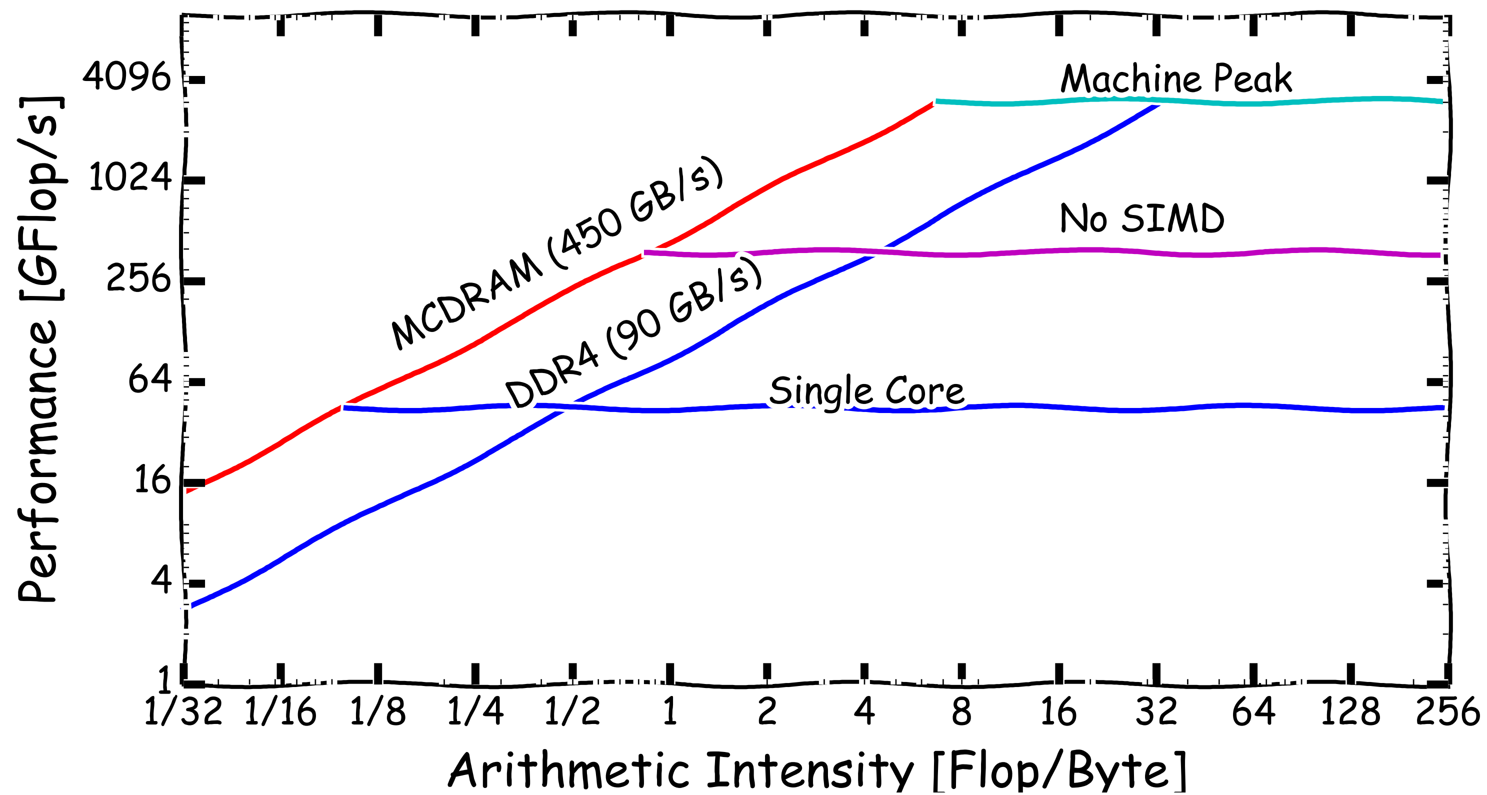


Roofline model

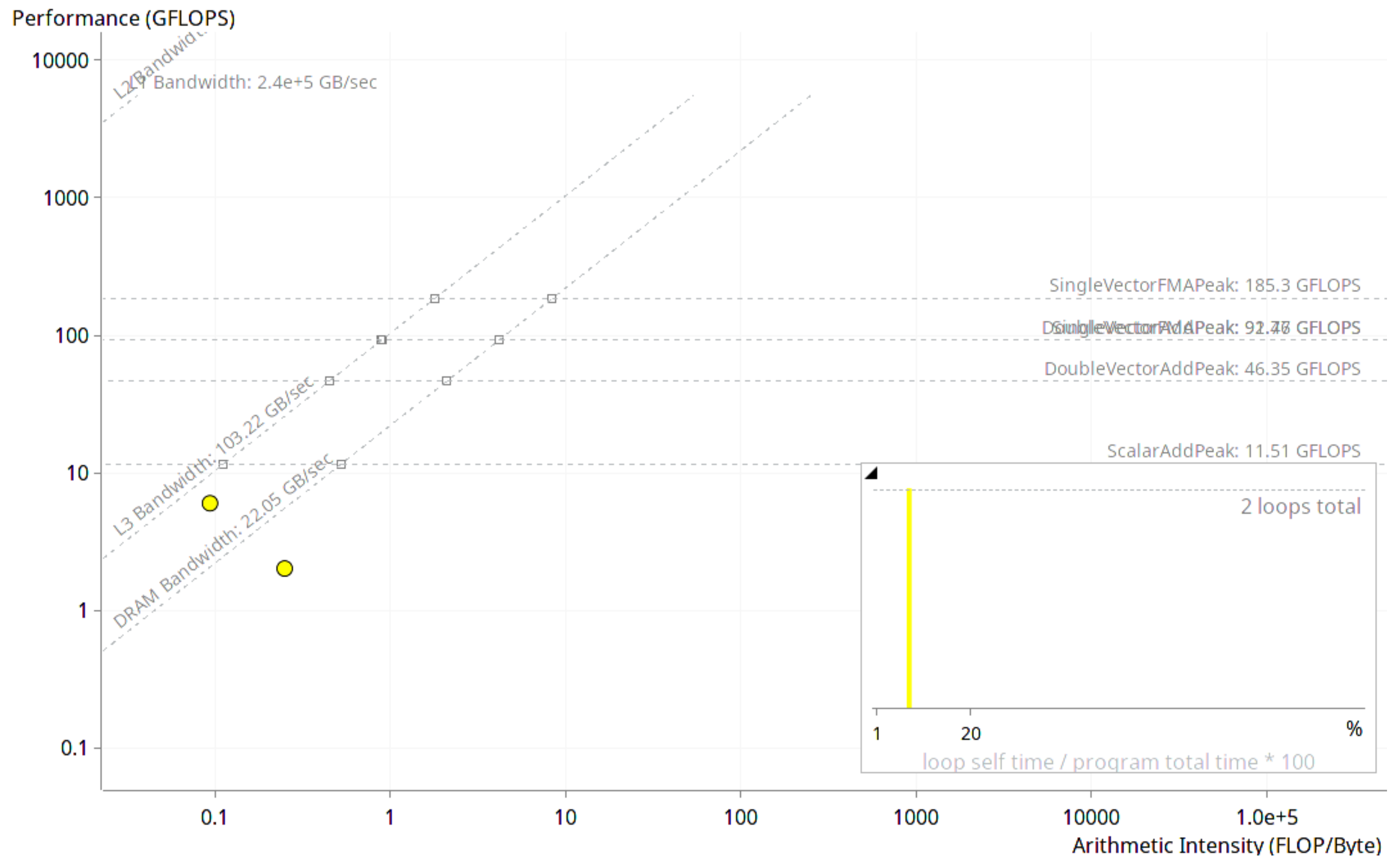


JÜLICH FORSCHUNGSZENTRUM

Roofline model

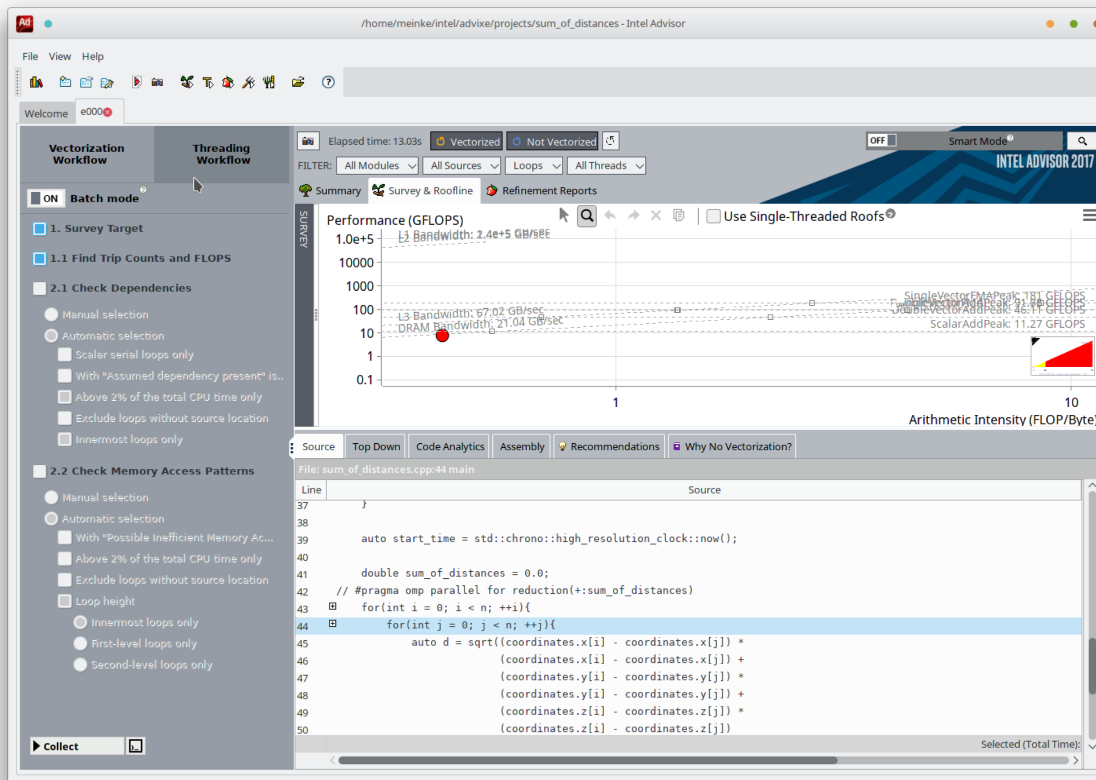


Roofline Model with Intel Advisor



JÜLICH FORSCHUNGSZENTRUM

Roofline Model with Intel Advisor



Experimental

export ADVIXE_EXPERIMENTAL=roofline

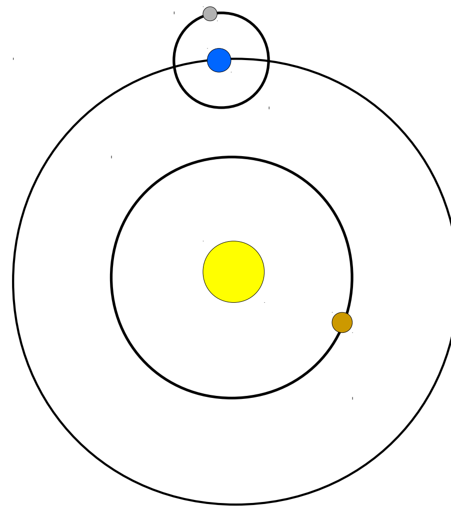
JÜLICH
FORSCHUNGSZENTRUM

Demo: Using MCDRAM

- How to determine modes?
- Using MCDRAM as cache
- numactl -m 1 executable

The N-Body Problem

N bodies

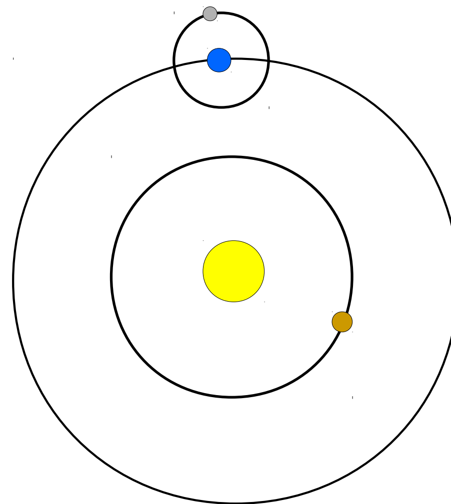


$$V(p_0, p_1) = -GM_1 \frac{M_2}{r_{12}}$$

Simulating the N-Body Problem

$$\boldsymbol{F}(p_{0,}p_1)=GM_1\frac{M_2}{r_{12}^2}\hat{\boldsymbol{r}}$$

$$\boldsymbol{F}=\boldsymbol{a}m$$



$$\boldsymbol{r}(t+dt)=\boldsymbol{r}(t)+\boldsymbol{v}(t)dt+\frac{1}{2}\boldsymbol{a}dt^2$$

$$\boldsymbol{v}(t+dt)=v(t)+a\,dt$$

JÜLICH FORSCHUNGSZENTRUM

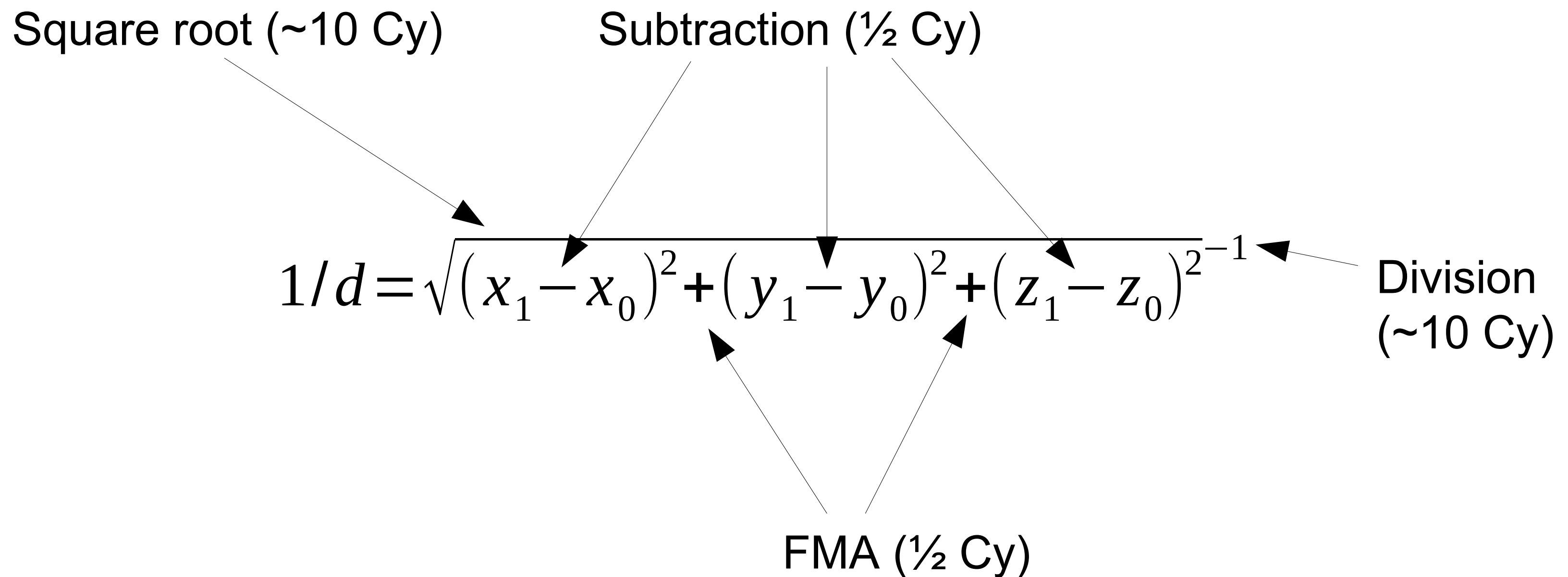
Estimating Peak Performance for N-Body

Calculation is dominated by sum of forces.
The sum of forces is dominated by the reciprocal distance:

$$1/d=\sqrt{(x_1-x_0)^2+(y_1-y_0)^2+(z_1-z_0)^2}^{-1}$$

The Intel Optimization Reference Manual has lots of information about the cost of operations:
http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html

Estimating Peak Performance for N-Body



Exercise: Optimizing the N-Body Code

Look at exercises/nbody and follow the instructions in README.rst

JÜLICH
FORSCHUNGSZENTRUM

References

[1] J. Jeffers, J. Reinders, and A. Sodani, Intel Xeon Phi Processor High Performance Programming, 2nd Edition, (Morgan Kaufmann, 2016).

[2] M. McCool, J. Reinders, and A. Robison, Structured Parallel Programming: Patterns for Efficient Computation, 1st Edition, (Morgan Kaufmann, 2012).