

# Intel Xeon Phi Knights Landing

Jan H. Meinke

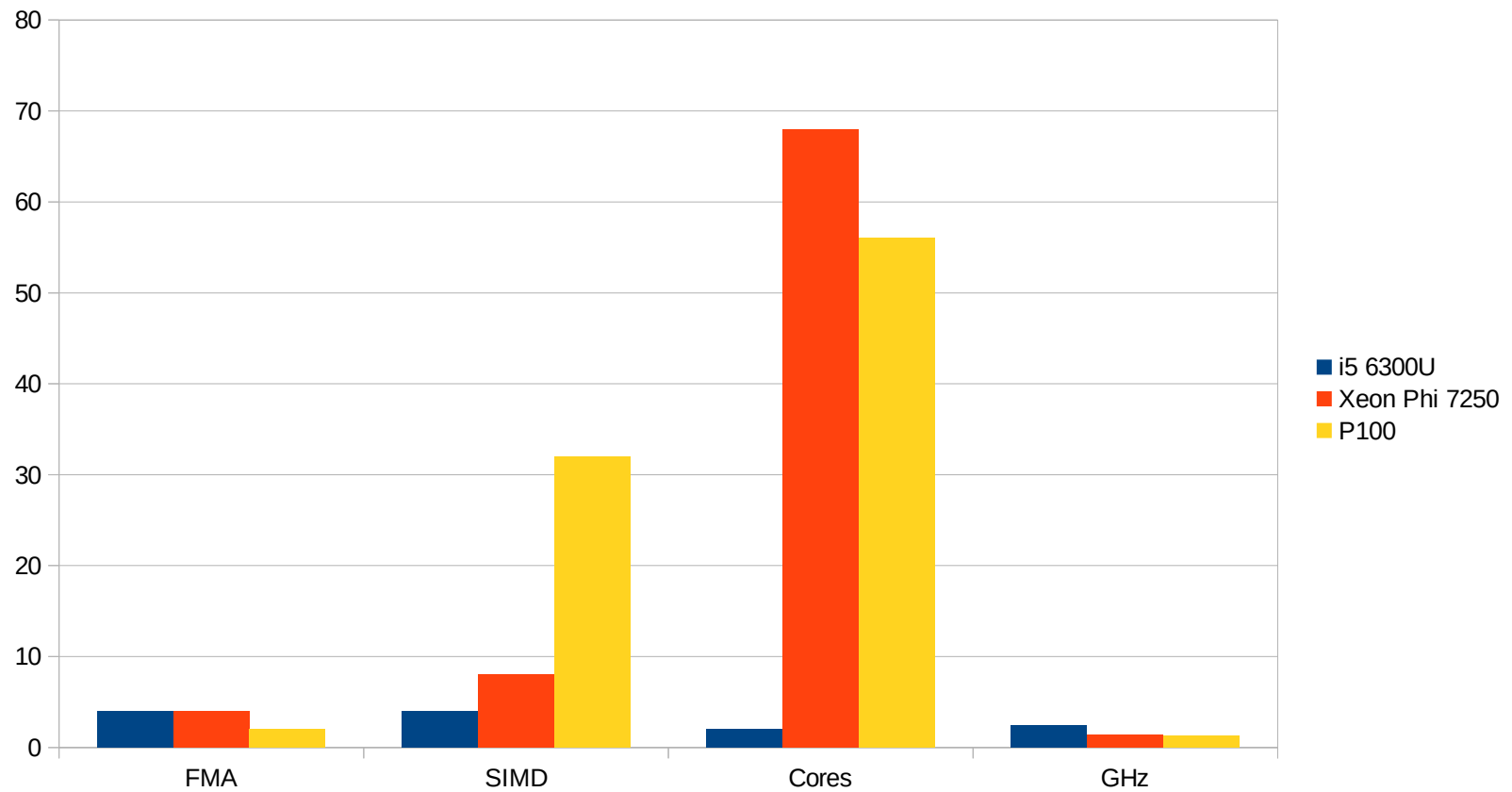
Your free Lunch will soon be over!

Herb Sutter

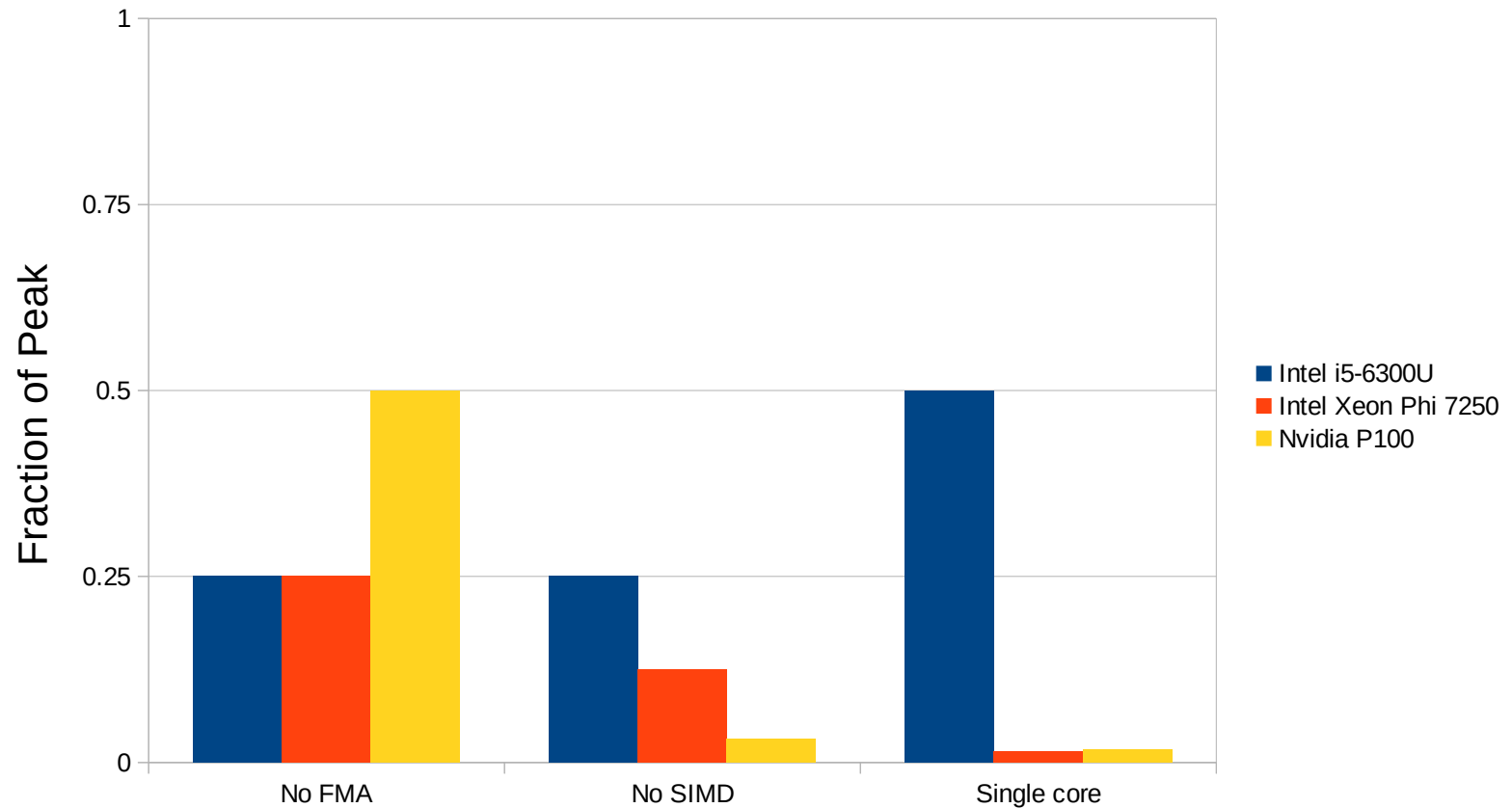
# Lots of Parallelism

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

# Processor comparison



# Floating Point Performance Contributions

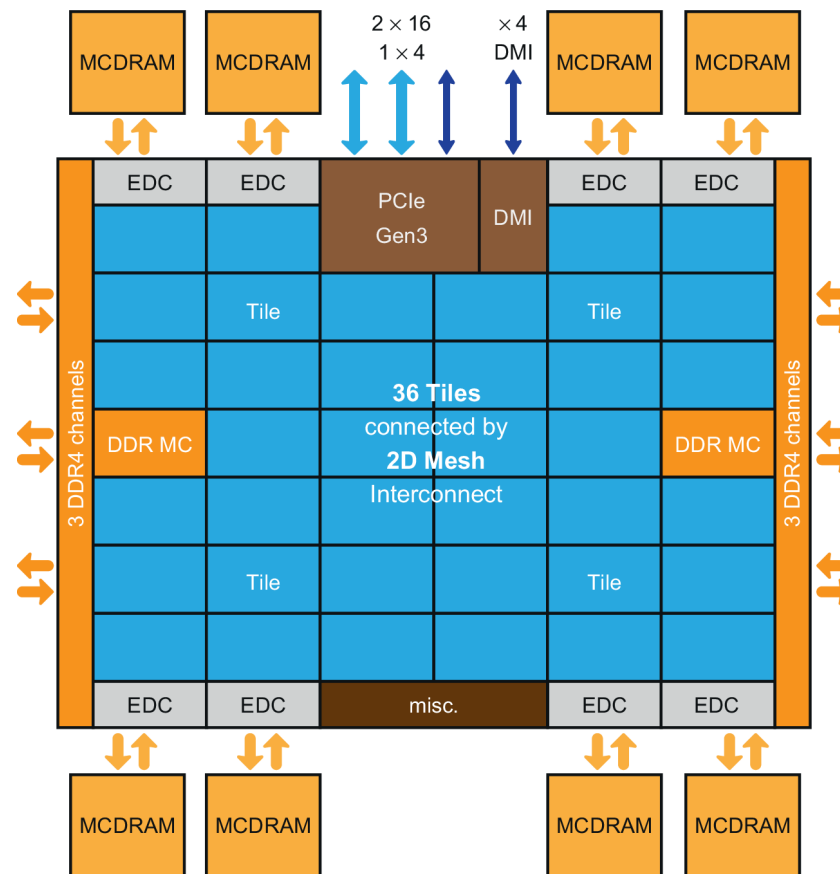


## What Could This Mean for You?

- 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

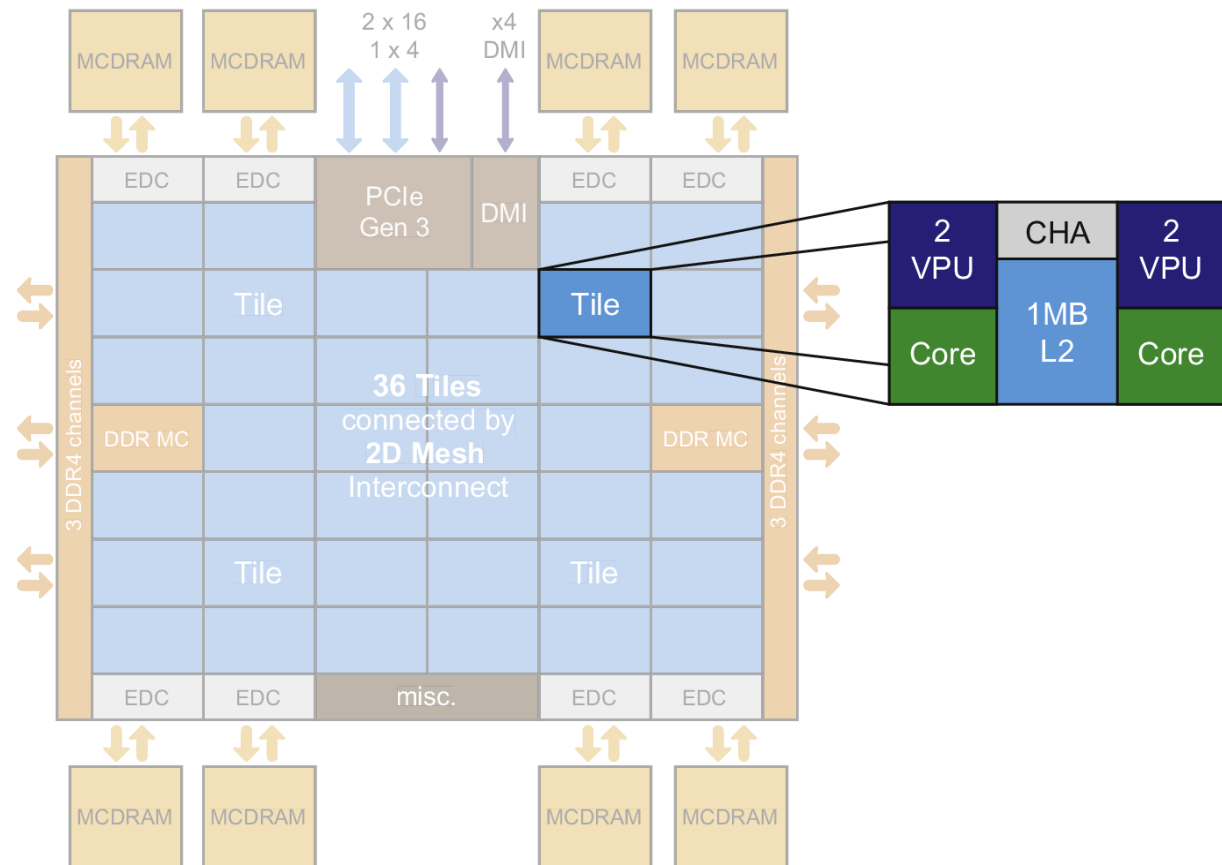
# KNL Architecture

# Intel Xeon Phi – Architecture

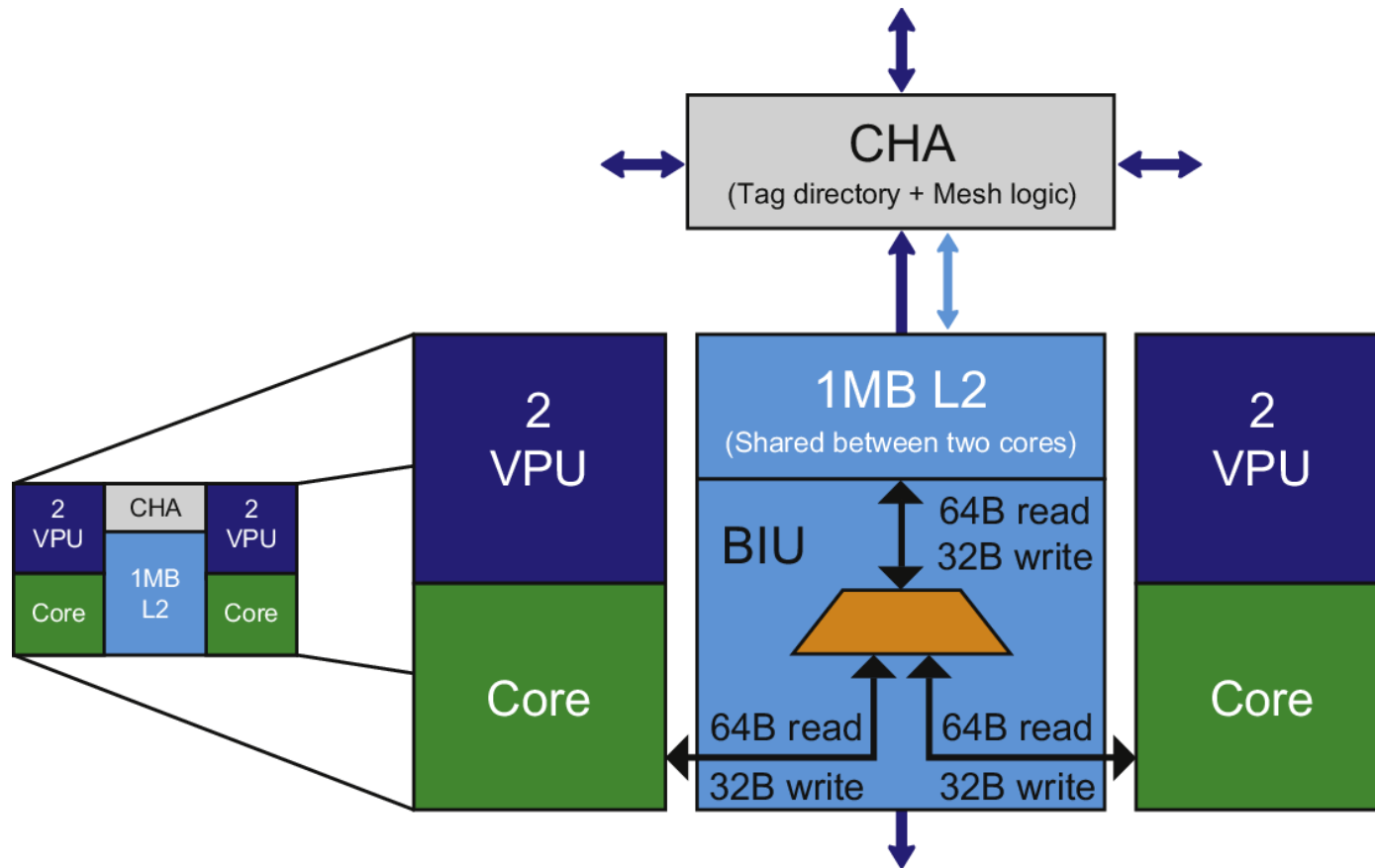




# Intel Xeon Phi – Tile

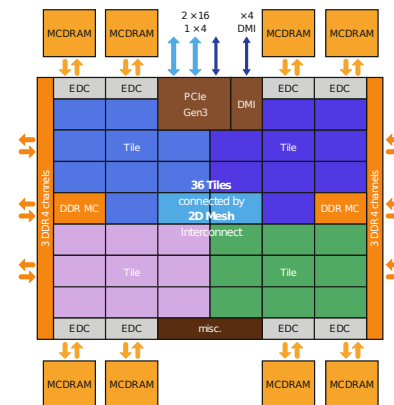
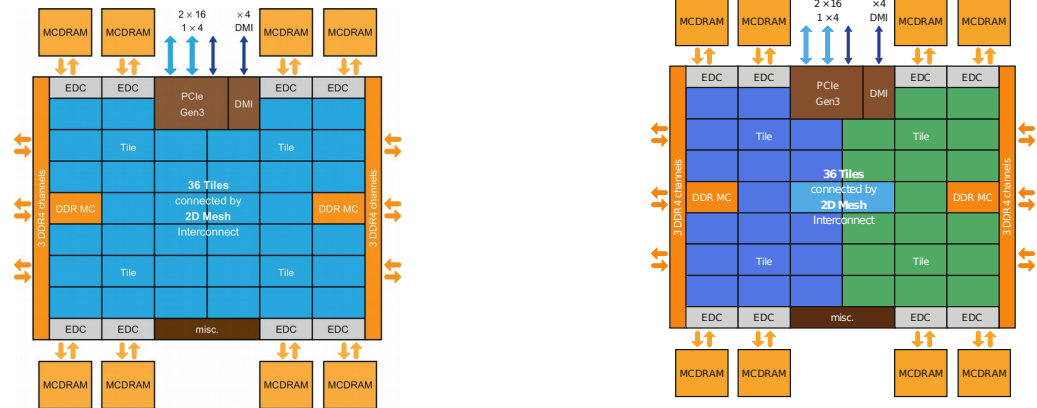


# Intel Xeon Phi – Tile



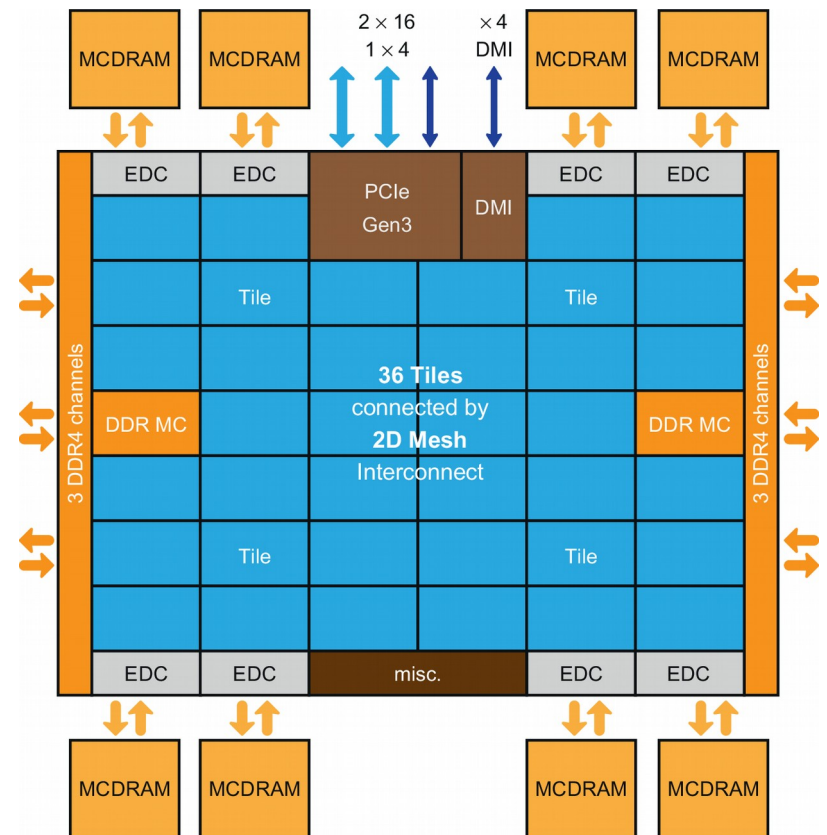
# Cluster Modes

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



# Cluster Modes: All-to-All

- Any tile
- Any CHA
- Any memory location



# Cluster Modes: Quadrant/SNC-4

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

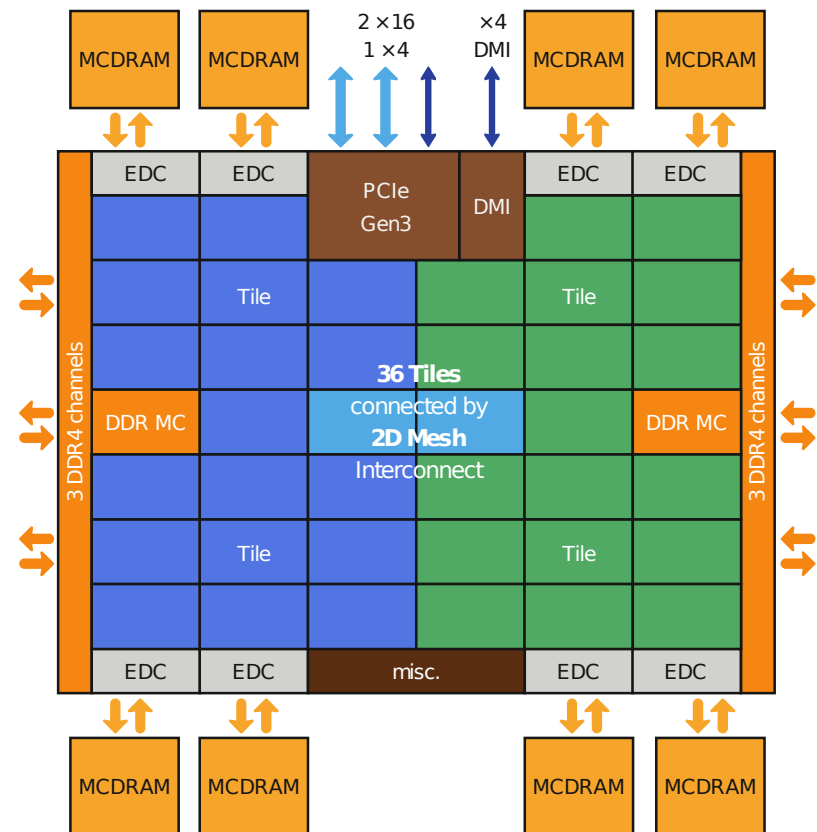
Sub-NUMA-Cluster (SNC): Divide chip into NUMA domains, c.f., multi-socket system. SNC-4 corresponds to a 4-socket node.



# Cluster Modes: Hemisphere/SNC-2

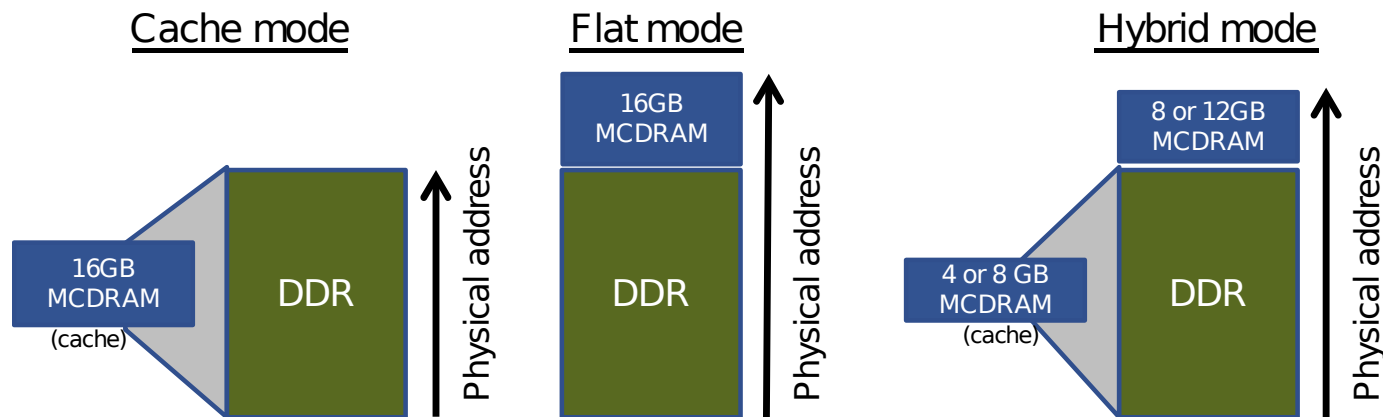
- Any tile
- CHA of the hemisphere of the memory location
- Memory in same hemisphere as CHA

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



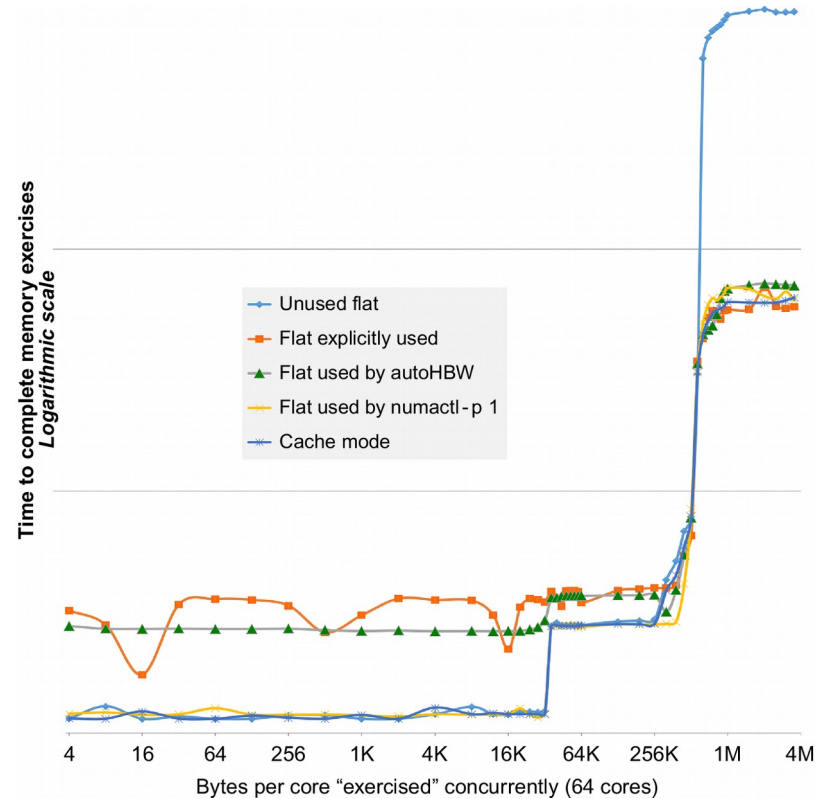
# MCDRAM

- 8x2 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

# Vectorization


zmm0 

+

zmm1 

k1 

=

zmm2 

Mask to treat if statement

# 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

# 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`

## 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);
```

```
for(int i = 0; i < N; ++i){  
    a[i] *= 3.14;  
}
```



```
for(auto& x : a){  
    x = 3.14;  
}
```



## 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**:

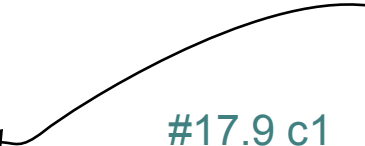
```
icpc -std=c++14 -O2 -xhost -ansi-alias -fargument-noalias -c order.cpp -S
```

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

```

..B1.24:                                # Preds ..B1.24 ..B1.23
                                           # Execution count [1.02e+03]
Packed   vaddpd  8(%rdx,%rcx,8), %zmm5, %zmm5  #17.9 c1
double   vaddpd  72(%rdx,%rcx,8), %zmm8, %zmm8 #17.9 c1
         vaddpd  136(%rdx,%rcx,8), %zmm7, %zmm7 #17.9 c7 stall 2
         vaddpd  200(%rdx,%rcx,8), %zmm6, %zmm6 #17.9 c7
         vaddpd  264(%rdx,%rcx,8), %zmm4, %zmm4 #17.9 c13 stall 2
         vaddpd  328(%rdx,%rcx,8), %zmm3, %zmm3 #17.9 c13
         vaddpd  392(%rdx,%rcx,8), %zmm2, %zmm2 #17.9 c19 stall 2
         vaddpd  456(%rdx,%rcx,8), %zmm1, %zmm1 #17.9 c19
         addq    $64, %rcx                       #16.5 c19
         cmpq    %rax, %rcx                      #16.5 c21
         jb     ..B1.24   # Prob 99%             #16.5 c23

```



## 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)	$A[\text{column}[i]]$
Gather (scatter) 16 elements	20 (25)	$A[\text{column}[i]]$
Horizontal reduction	30	$\text{sum} += A[i]$
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.

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] * (x[j]-x[i]);
        Fy[i] += G*m[i]*m[j] * (y[j]-y[i]);
        Fz[i] += G*m[i]*m[j] * (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)	private(list)
simdlen(length)	lastprivate(list)
linear(list[:linear step])	reduction(type : list)
aligned(list[:alignment in bytes])	collapse(n)

# Vectorized User Functions

```
double distance(const double x0, const double x1){
    return sqrt((x1 - x0) * (x1 - x0));
}
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";
}
```





# 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);
  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 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

```

#include "distance.h"
int main(int argc, char** argv)
{
    constexpr size_t N = 1024;
    std::vector<double> x(N, (double) 1.0);
    for(size_t i; i < N; i++)
        x[i] = (double) i;

    for(size_t i = 0; i < N; i++)
    {
        double sum = 0.0;
        for(size_t j = 0; j < N; j++)
            if(i != j) sum += distance(x[i], x[j]);
        f[i] = sum;
    }
    auto sum = std::accumulate(f.begin(), f.end(), double(0.0));
    std::cout << "Average vector 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

# 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;
        #pragma omp for
        for(size_t j = 0; j < N; ++j)
            if(i != j) sum += 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`

# 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`

## 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

# (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:  
 $i = 0: f(i) \parallel i = 1: f(i) \parallel i = 2: f(i) \parallel \dots \parallel i = n - 1: f(i)$

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:  
 $i = 0: f(i) \parallel i = 1: f(i) \parallel i = 2: f(i) \parallel \dots \parallel i = n - 1: f(i)$

Correctness is responsibility of the programmer.

Use *-parallel* with ifort to parallelize do concurrent

## 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>

# Intel Threading Building Blocks (TBB)

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



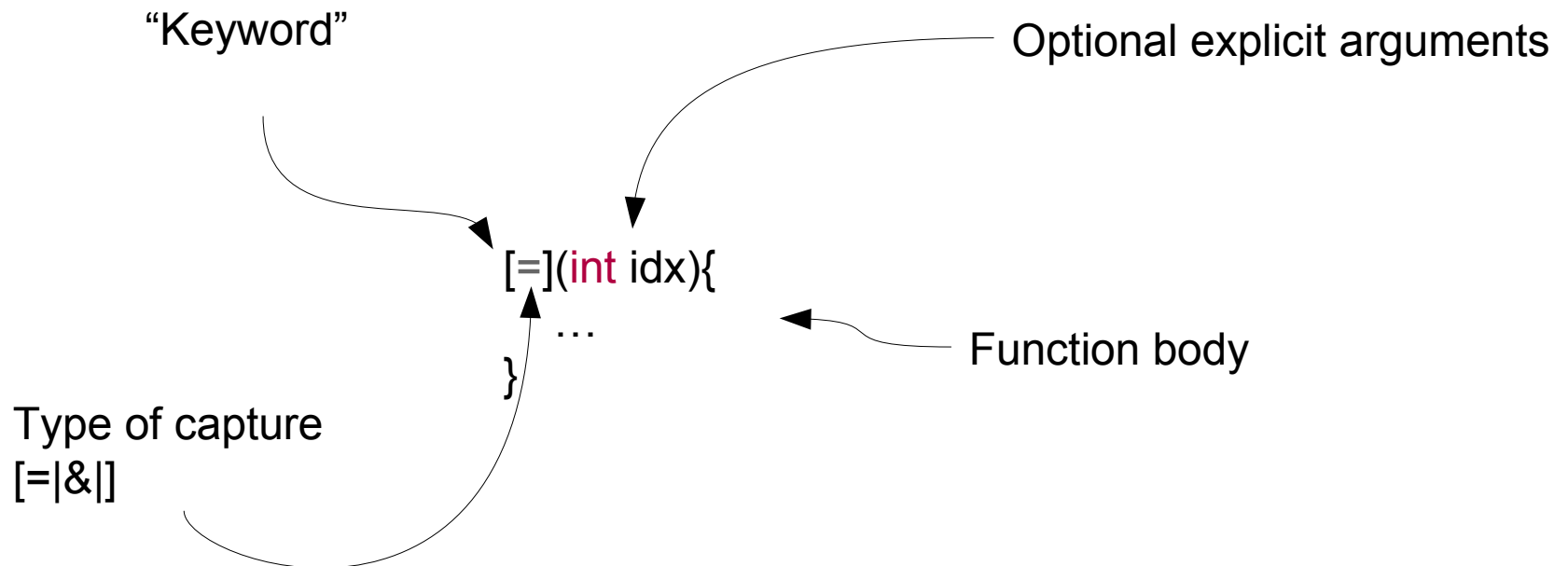
## tbb::parallel\_for

```
#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;

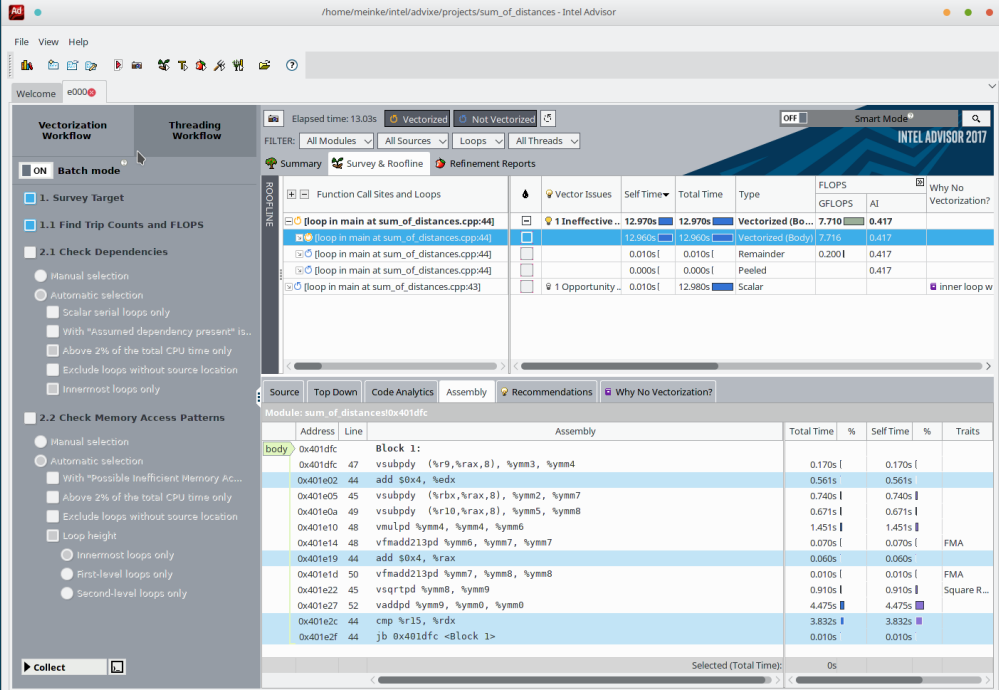
start_time = std::chrono::high_resolution_clock::now();
sum = std::accumulate(coordinates.begin(), coordinates.end(), 0.0);
stop_time = std::chrono::high_resolution_clock::now();
dt = stop_time - start_time;
std::cout << "#Serial average is " << (sum / n) << " and took " <<
    dt.count() << " s" <<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

# Intel Advisor



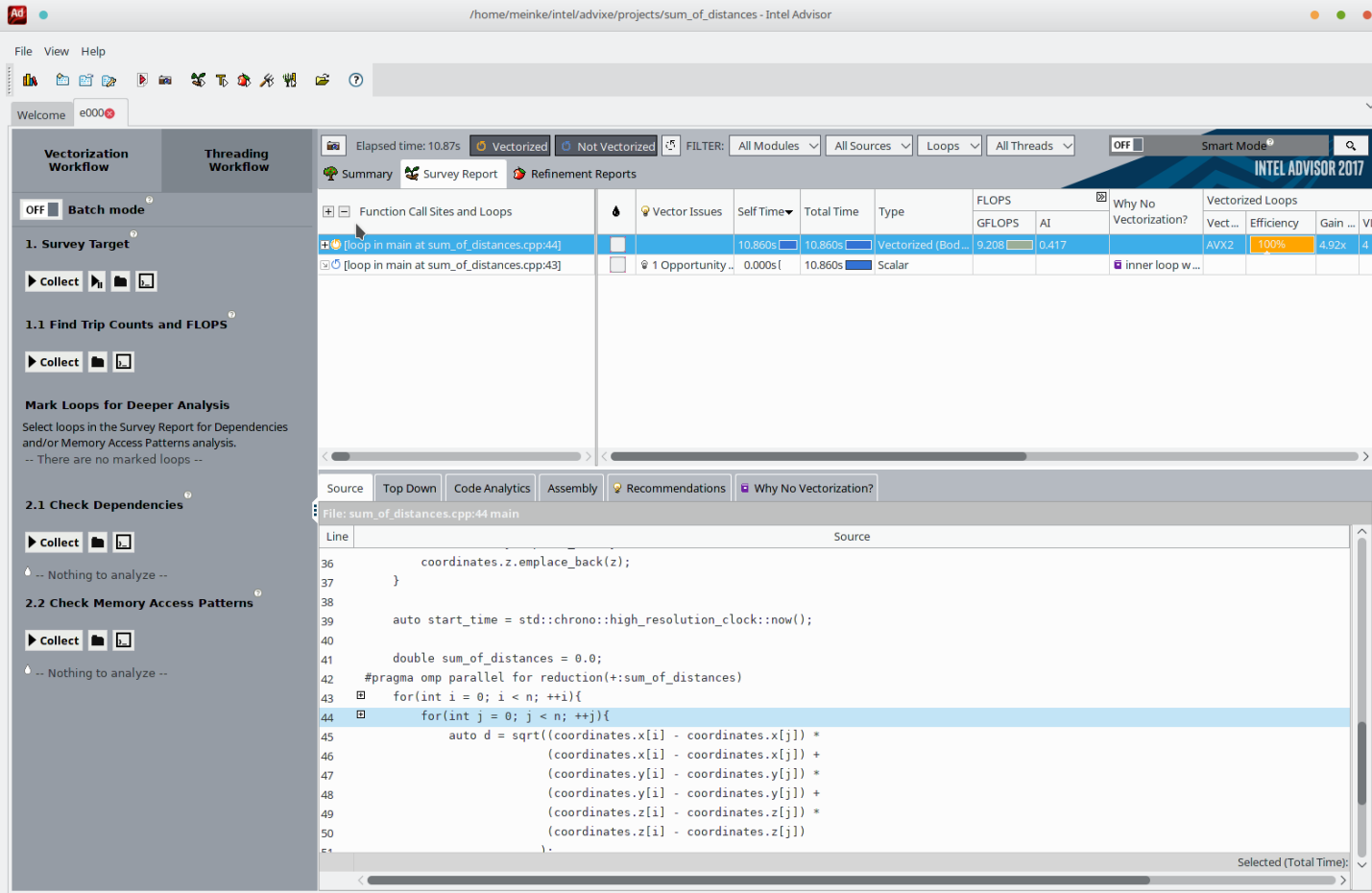
The screenshot displays the Intel Advisor interface for a project named 'sum\_of\_distances'. The left sidebar shows the 'Vectorization Workflow' with 'Batch mode' selected. The main window shows a summary table of analysis results.

Function Call Sites and Loops	Vector Issues	Self Time	Total Time	Type	FLOPS	AI	Why No Vectorization?
[x] loop in main at sum_of_distances.cpp:44	1 Ineffective ...	12.970s	12.970s	Vectorized (Bo...	7.710	0.417	
[x] loop in main at sum_of_distances.cpp:44		12.960s	12.960s	Vectorized (Body)	7.716	0.417	
[x] loop in main at sum_of_distances.cpp:44		0.010s	0.010s	Remainder	0.2001	0.417	
[x] loop in main at sum_of_distances.cpp:44		0.000s	0.000s	Peeled		0.417	
[x] loop in main at sum_of_distances.cpp:43	1 Opportunity ...	0.010s	12.980s	Scalar			inner loop w

Below the summary table, the 'Assembly' view shows the code for the selected loop:

Address	Line	Assembly	Total Time	%	Self Time	%	Traits
body	0x401dfc	BLOCK 1:					
	0x401dfc	vsbpdq (%r9,%rax,%), %ymm3, %ymm4	0.170s		0.170s		
	0x401e02	add \$0x4, %edx	0.561s		0.561s		
	0x401e05	vsbpdq (%rbx,%rax,%), %ymm2, %ymm7	0.740s		0.740s		
	0x401e0a	vsbpdq (%r10,%rax,%), %ymm5, %ymm8	0.671s		0.671s		
	0x401e10	vmlpd %ymm4, %ymm4, %ymm6	1.451s		1.451s		
	0x401e14	vfadd213pd %ymm6, %ymm7, %ymm7	0.070s		0.070s		FMA
	0x401e19	add \$0x4, %rax	0.060s		0.060s		
	0x401e1d	vfadd213pd %ymm7, %ymm8, %ymm8	0.010s		0.010s		FMA
	0x401e22	vsqrtpd %ymm8, %ymm9	0.910s		0.910s		Square R...
	0x401e27	vaddpd %ymm9, %ymm9, %ymm9	4.475s		4.475s		
	0x401e2c	cmp %r15, %rdx	3.832s		3.832s		
	0x401e2f	jb 0x401dfc <BLOCK 1>	0.010s		0.010s		

# Intel Advisor



The screenshot shows the Intel Advisor interface for a project named 'sum\_of\_distances'. The main window displays a summary table with the following data:

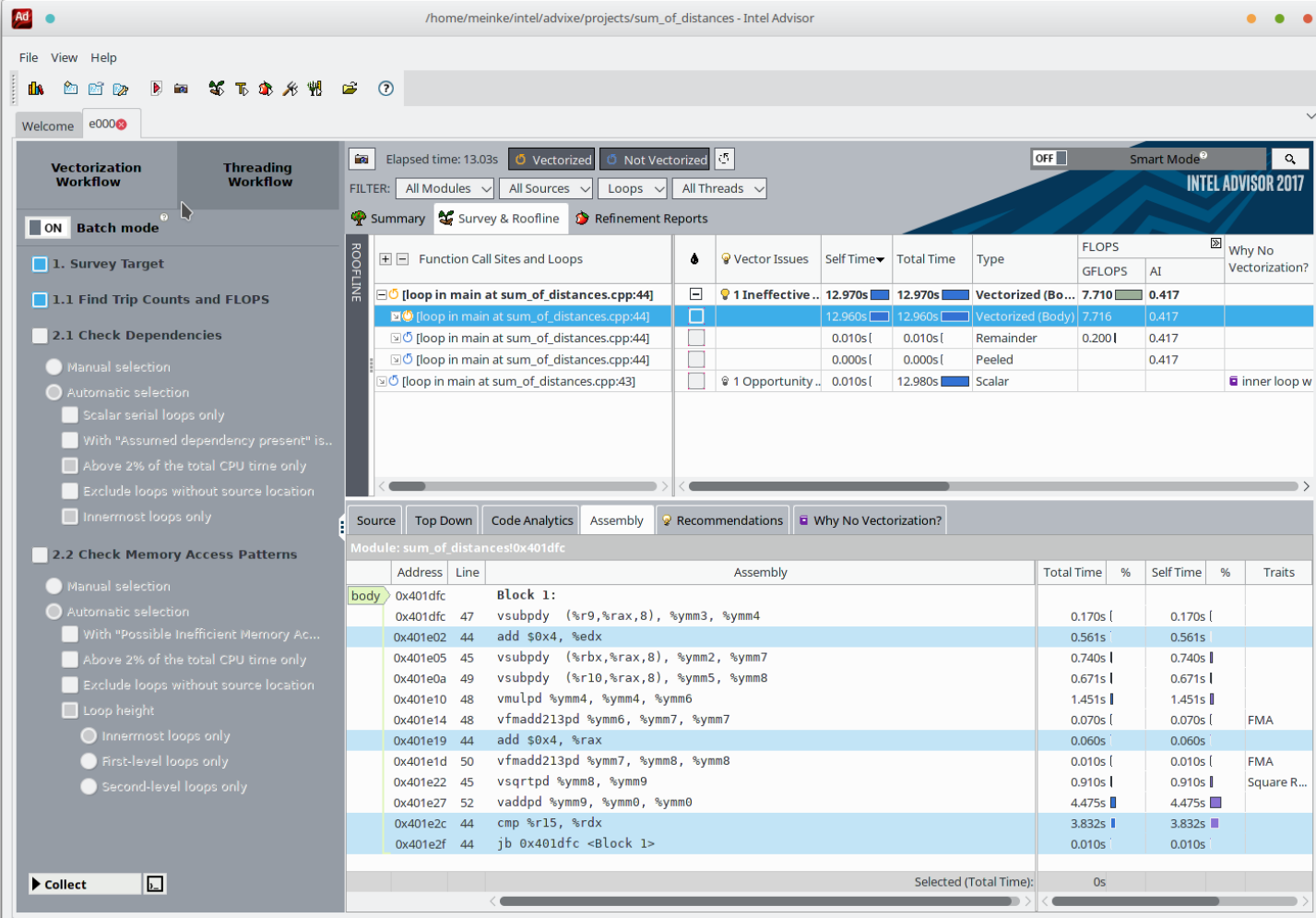
Function Call Sites and Loops	Vector Issues	Self Time	Total Time	Type	FLOPS	Why No Vectorization?	Vectorized Loops
					GFLOPS	AI	Vect... Efficiency Gain ... VI
[loop in main at sum_of_distances.cpp:44]		10.860s	10.860s	Vectorized (Bod...)	9.206	0.417	AVX2 100% 4.92x 4
[loop in main at sum_of_distances.cpp:43]	1 Opportunity..	0.000s	10.860s	Scalar			inner loop w...

The source code view shows the following code snippet:

```

36     coordinates.z.emplace_back(z);
37   }
38
39   auto start_time = std::chrono::high_resolution_clock::now();
40
41   double sum_of_distances = 0.0;
42   #pragma omp parallel for reduction(+:sum_of_distances)
43   for(int i = 0; i < n; ++i){
44     for(int j = 0; j < n; ++j){
45       auto d = sqrt((coordinates.x[i] - coordinates.x[j]) *
46                   (coordinates.x[i] - coordinates.x[j]) +
47                   (coordinates.y[i] - coordinates.y[j]) *
48                   (coordinates.y[i] - coordinates.y[j]) +
49                   (coordinates.z[i] - coordinates.z[j]) *
50                   (coordinates.z[i] - coordinates.z[j])
51     );
  
```

# Look at the Assembler—the Easier Way



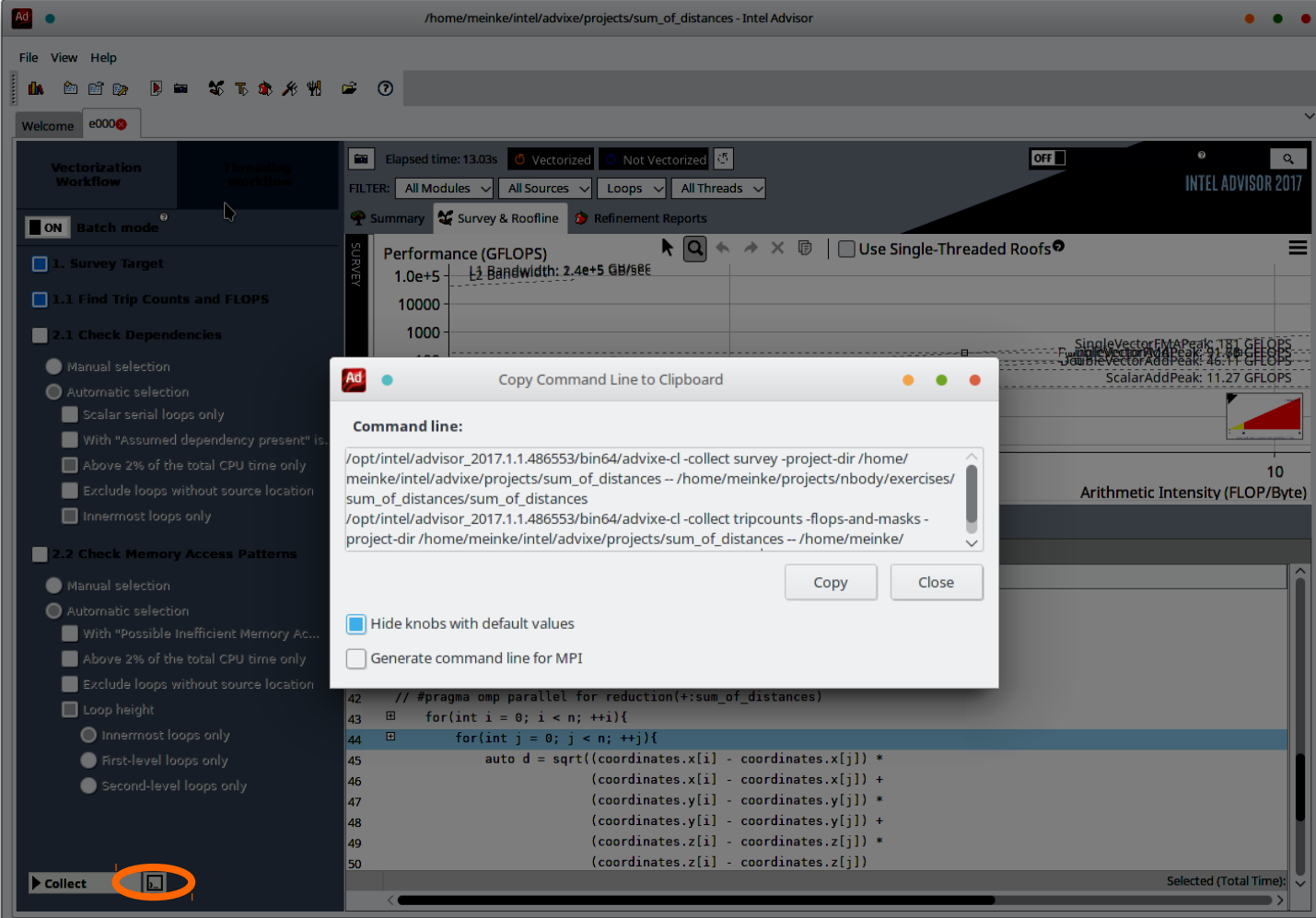
The screenshot shows the Intel Advisor 2017 interface. On the left, the 'Vectorization Workflow' is displayed with 'Batch mode' selected. The workflow includes '1. Survey Target' and '2.2 Check Memory Access Patterns'. The main window shows a summary table of vectorization results for a loop in 'sum\_of\_distances.cpp:44'. Below this, the 'Assembly' view is open, showing the assembly code for 'Block 1' starting at address 0x401dfc.

Function Call Sites and Loops	Vector Issues	Self Time	Total Time	Type	FLOPS		Why No Vectorization?
					GFLOPS	AI	
[loop in main at sum_of_distances.cpp:44]	1 Ineffective...	12.970s	12.970s	Vectorized (Bo...	7.710	0.417	
[loop in main at sum_of_distances.cpp:44]		12.960s	12.960s	Vectorized (Body)	7.716	0.417	
[loop in main at sum_of_distances.cpp:44]		0.010s	0.010s	Remainder	0.200	0.417	
[loop in main at sum_of_distances.cpp:44]		0.000s	0.000s	Peeled		0.417	
[loop in main at sum_of_distances.cpp:43]	1 Opportunity...	0.010s	12.980s	Scalar			inner loop w

Address	Line	Assembly	Total Time	%	Self Time	%	Traits
body	0x401dfc	Block 1:					
	0x401dfc 47	vsubpd (%r9,%rax,8), %ymm3, %ymm4	0.170s		0.170s		
	0x401e02 44	add \$0x4, %edx	0.561s		0.561s		
	0x401e05 45	vsubpd (%rbx,%rax,8), %ymm2, %ymm7	0.740s		0.740s		
	0x401e0a 49	vsubpd (%r10,%rax,8), %ymm5, %ymm8	0.671s		0.671s		
	0x401e10 48	vmulpd %ymm4, %ymm4, %ymm6	1.451s		1.451s		
	0x401e14 48	vfmadd213pd %ymm6, %ymm7, %ymm7	0.070s		0.070s		FMA
	0x401e19 44	add \$0x4, %rax	0.060s		0.060s		
	0x401e1d 50	vfmadd213pd %ymm7, %ymm8, %ymm8	0.010s		0.010s		FMA
	0x401e22 45	vsqrtpd %ymm8, %ymm9	0.910s		0.910s		Square R...
	0x401e27 52	vaddpd %ymm9, %ymm0, %ymm0	4.475s		4.475s		
	0x401e2c 44	cmp %r15, %rdx	3.832s		3.832s		
	0x401e2f 44	jb 0x401dfc <Block 1>	0.010s		0.010s		

# Intel Advisor



The screenshot shows the Intel Advisor interface for a project named 'sum\_of\_distances'. The main window displays a performance graph with a peak of 1.0e+5 GFLOPS and a bandwidth of 2.4e+5 GB/s. A dialog box titled 'Copy Command Line to Clipboard' is open, showing the following command line:

```

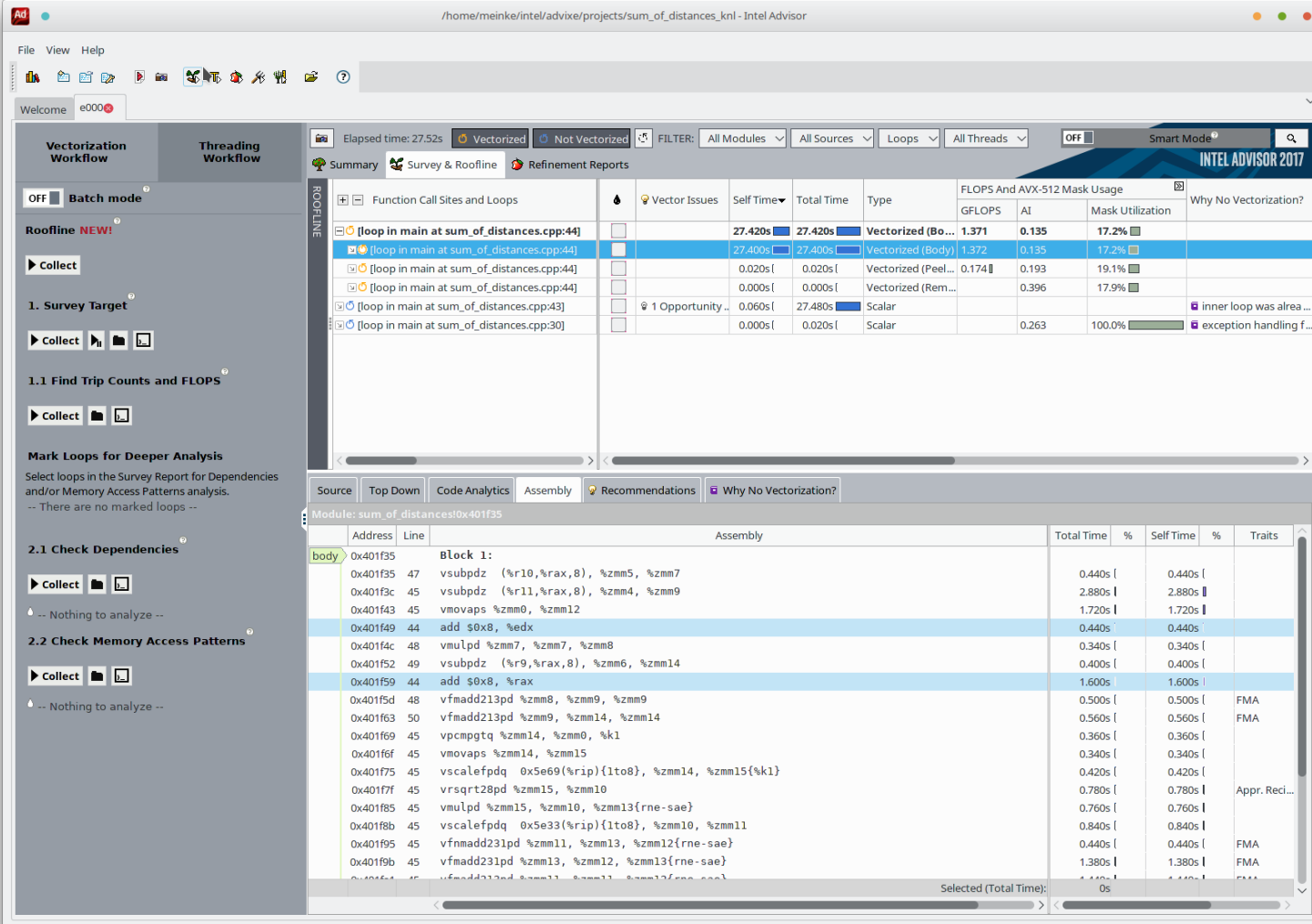
/opt/intel/advisor_2017.1.1.486553/bin64/advixe-cl-collect survey -project-dir /home/
meinke/intel/advixe/projects/sum_of_distances -- /home/meinke/projects/nbody/exercises/
sum_of_distances/sum_of_distances
/opt/intel/advisor_2017.1.1.486553/bin64/advixe-cl-collect tripcounts -flops-and-masks -
project-dir /home/meinke/intel/advixe/projects/sum_of_distances -- /home/meinke/
  
```

The dialog box also includes options for 'Hide knobs with default values' and 'Generate command line for MPI'. The background interface shows a code editor with the following code snippet:

```

42 // #pragma omp parallel for reduction(+:sum_of_distances)
43 for(int i = 0; i < n; ++i){
44     for(int j = 0; j < n; ++j){
45         auto d = sqrt((coordinates.x[i] - coordinates.x[j]) *
46                     (coordinates.x[i] - coordinates.x[j]) +
47                     (coordinates.y[i] - coordinates.y[j]) *
48                     (coordinates.y[i] - coordinates.y[j]) +
49                     (coordinates.z[i] - coordinates.z[j]) *
50                     (coordinates.z[i] - coordinates.z[j]))
  
```

# Look at the Assembler—the Easier Way



The screenshot displays the Intel Advisor 2017 interface. The top navigation bar includes 'Vectorization Workflow' and 'Threading Workflow'. The main window shows a 'Roofline' view with a table of function call sites and loops. The table columns include 'Function Call Sites and Loops', 'Vector Issues', 'Self Time', 'Total Time', 'Type', and 'FLOPS And AVX-512 Mask Usage'. The table lists several loops, with some marked as 'Vectorized (Body)' and others as 'Scalar'. The bottom pane shows the assembly code for the selected loop, with columns for 'Address', 'Line', 'Assembly', 'Total Time', '%', 'Self Time', '%', and 'Traits'.

Function Call Sites and Loops	Vector Issues	Self Time	Total Time	Type	FLOPS And AVX-512 Mask Usage	Why No Vectorization?
					GFLOPS AI Mask Utilization	
[loop in main at sum_of_distances.cpp:44]		27.420s	27.420s	Vectorized (Bo...	1.371 0.135 17.2%	
[loop in main at sum_of_distances.cpp:44]		27.400s	27.400s	Vectorized (Body)	1.372 0.135 17.2%	
[loop in main at sum_of_distances.cpp:44]		0.020s	0.020s	Vectorized (Peel...	0.174 0.193 19.1%	
[loop in main at sum_of_distances.cpp:44]		0.000s	0.000s	Vectorized (Rem...	0.396 17.9%	
[loop in main at sum_of_distances.cpp:43]	1 Opportunity ...	0.060s	27.480s	Scalar		inner loop was alrea...
[loop in main at sum_of_distances.cpp:30]		0.000s	0.020s	Scalar	0.263 100.0%	exception handling f...

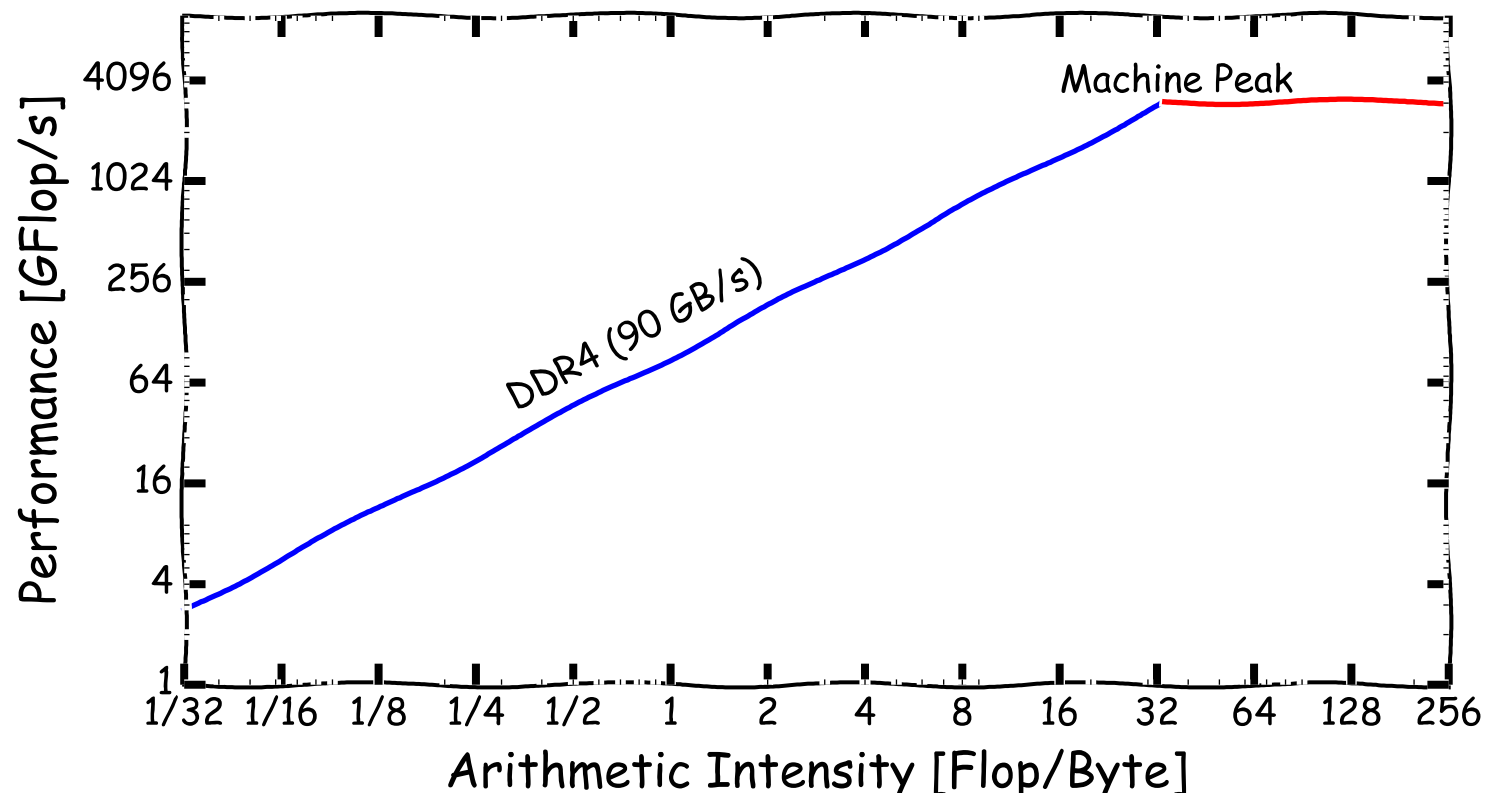
  

Address	Line	Assembly	Total Time	%	Self Time	%	Traits
0x401f35	47	Block 1: vsubpdz (%r10,%rax,8), %zmm5, %zmm7	0.440s		0.440s		
0x401f3c	45	vsubpdz (%r11,%rax,8), %zmm4, %zmm9	2.880s		2.880s		FMA
0x401f43	45	vmovaps %zmm0, %zmm12	1.720s		1.720s		
0x401f49	44	add \$0x8, %edx	0.440s		0.440s		
0x401f4c	48	vmulpd %zmm7, %zmm7, %zmm8	0.340s		0.340s		
0x401f52	49	vsubpdz (%r9,%rax,8), %zmm6, %zmm14	0.400s		0.400s		
0x401f59	44	add \$0x8, %rax	1.600s		1.600s		
0x401f5d	48	vmadd213pd %zmm8, %zmm9, %zmm9	0.500s		0.500s		FMA
0x401f63	50	vmadd213pd %zmm9, %zmm14, %zmm14	0.560s		0.560s		FMA
0x401f69	45	vpcmpgtq %zmm14, %zmm0, %k1	0.360s		0.360s		
0x401f6f	45	vmovaps %zmm14, %zmm15	0.340s		0.340s		
0x401f75	45	vscalefpdq 0x5e69(%rip){1to8}, %zmm14, %zmm15{k1}	0.420s		0.420s		
0x401f7f	45	vrsqrt28pd %zmm15, %zmm10	0.780s		0.780s		Appr. Rec...
0x401f85	45	vmulpd %zmm15, %zmm10, %zmm13{rne-sae}	0.760s		0.760s		
0x401f8b	45	vscalefpdq 0x5e33(%rip){1to8}, %zmm10, %zmm11	0.840s		0.840s		
0x401f95	45	vmadd231pd %zmm11, %zmm13, %zmm12{rne-sae}	0.440s		0.440s		FMA
0x401f9b	45	vmadd231pd %zmm13, %zmm12, %zmm13{rne-sae}	1.380s		1.380s		FMA
0x401fa1	45	vmadd231pd %zmm11, %zmm11, %zmm13{rne-sae}	1.140s		1.140s		FMA

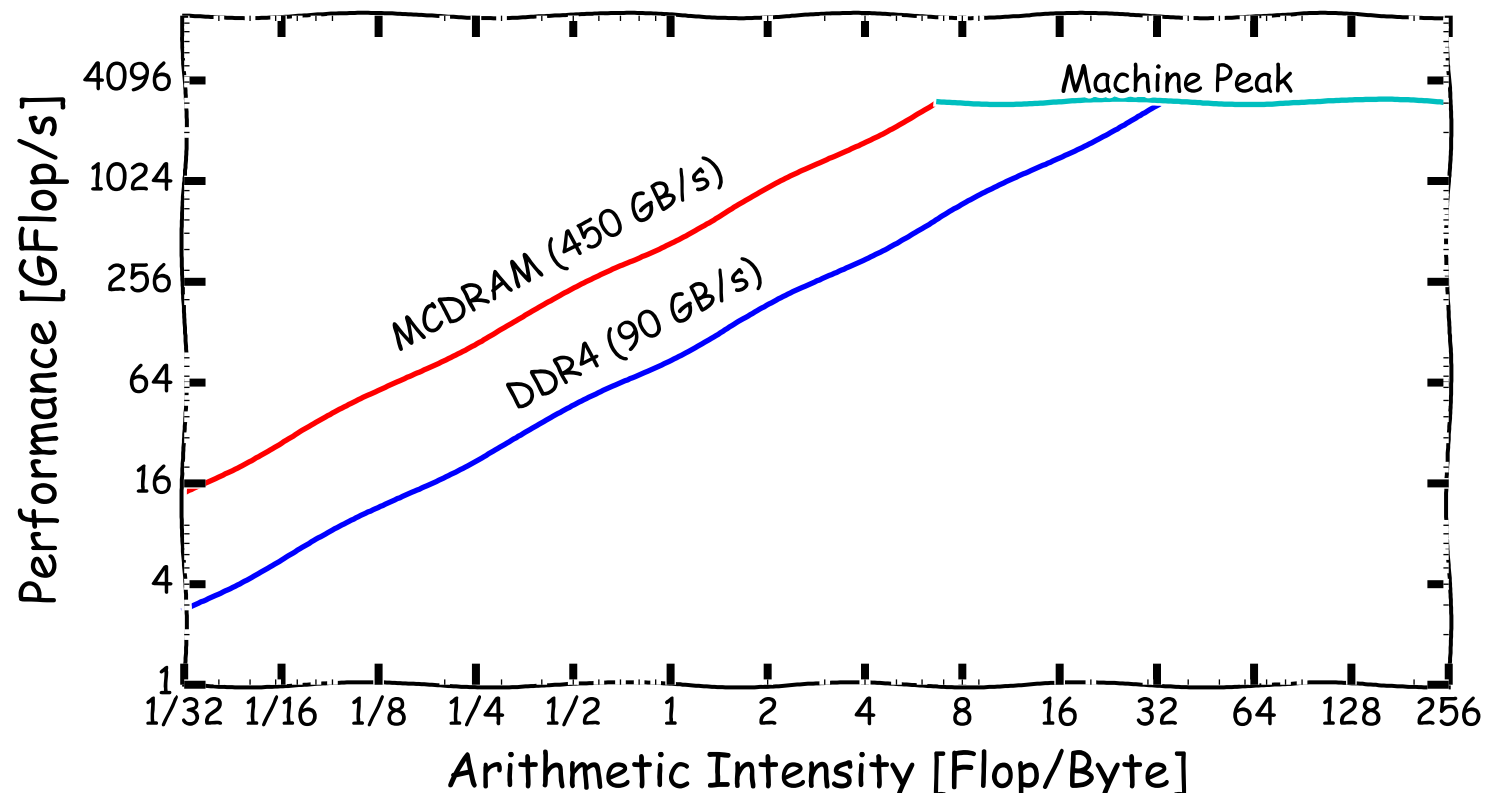


# Performance Modeling

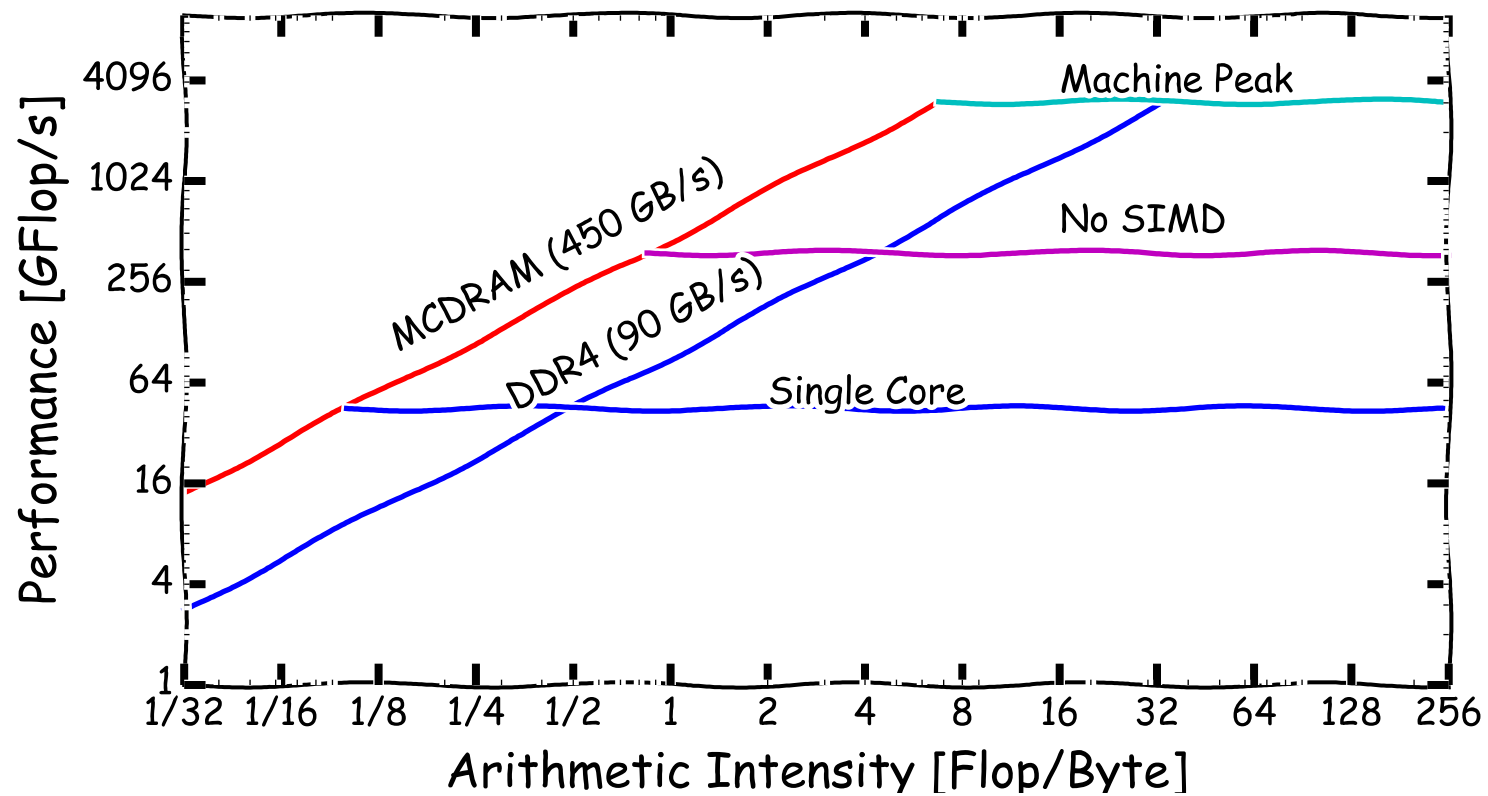
# Roofline model



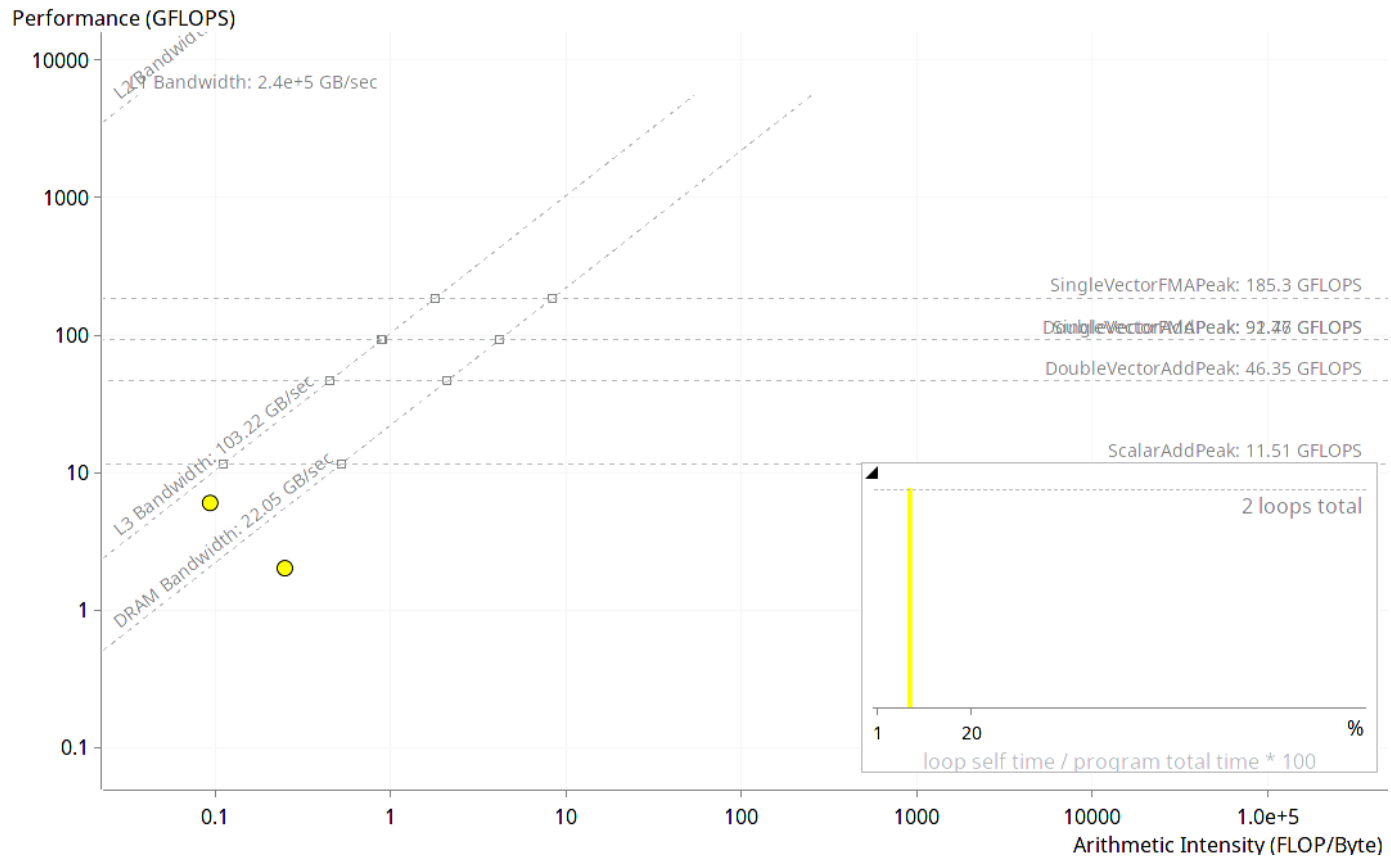
# Roofline model



# Roofline model

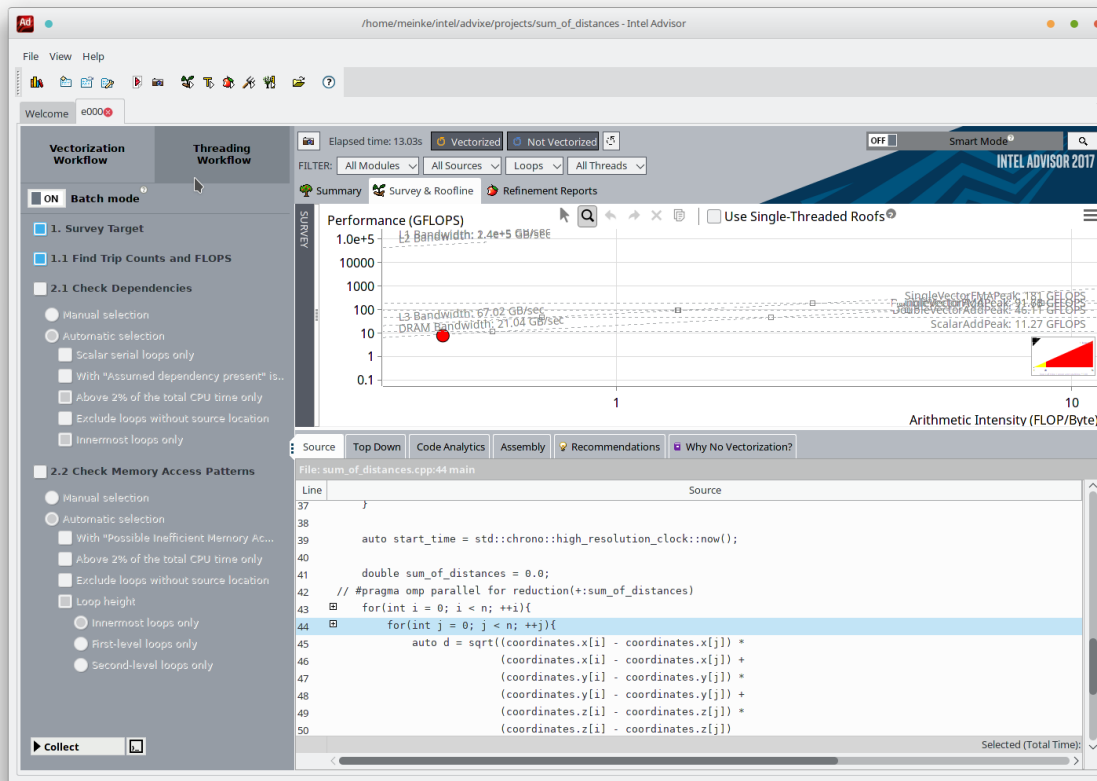


# Roofline Model with Intel Advisor



# Roofline Model with Intel Advisor

Experimental



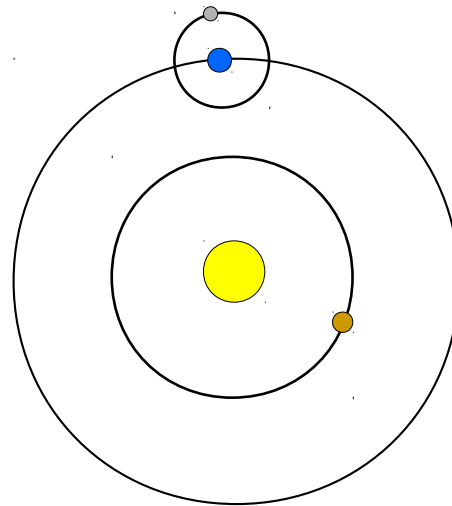
export ADVIXE\_EXPERIMENTAL=roofline

## Demo: Using MCDRAM

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

# The N-Body Problem

N bodies



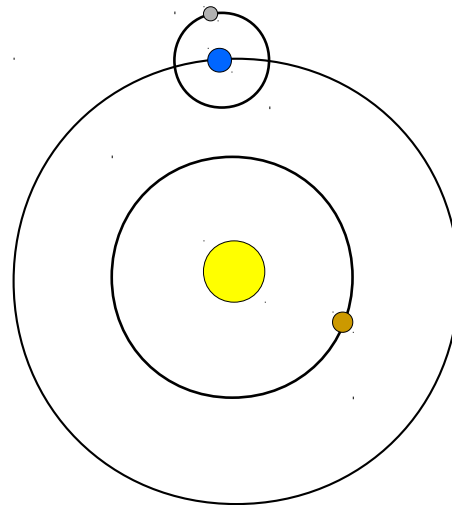
$$V(p_0, p_1) = -GM_1 \frac{M_2}{r_{12}}$$



# Simulating the N-Body Problem

$$\mathbf{F}(p_0, p_1) = GM_1 \frac{M_2}{r_{12}^2} \hat{\mathbf{r}}$$

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



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

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

## Estimating Peak Performance for N-Body

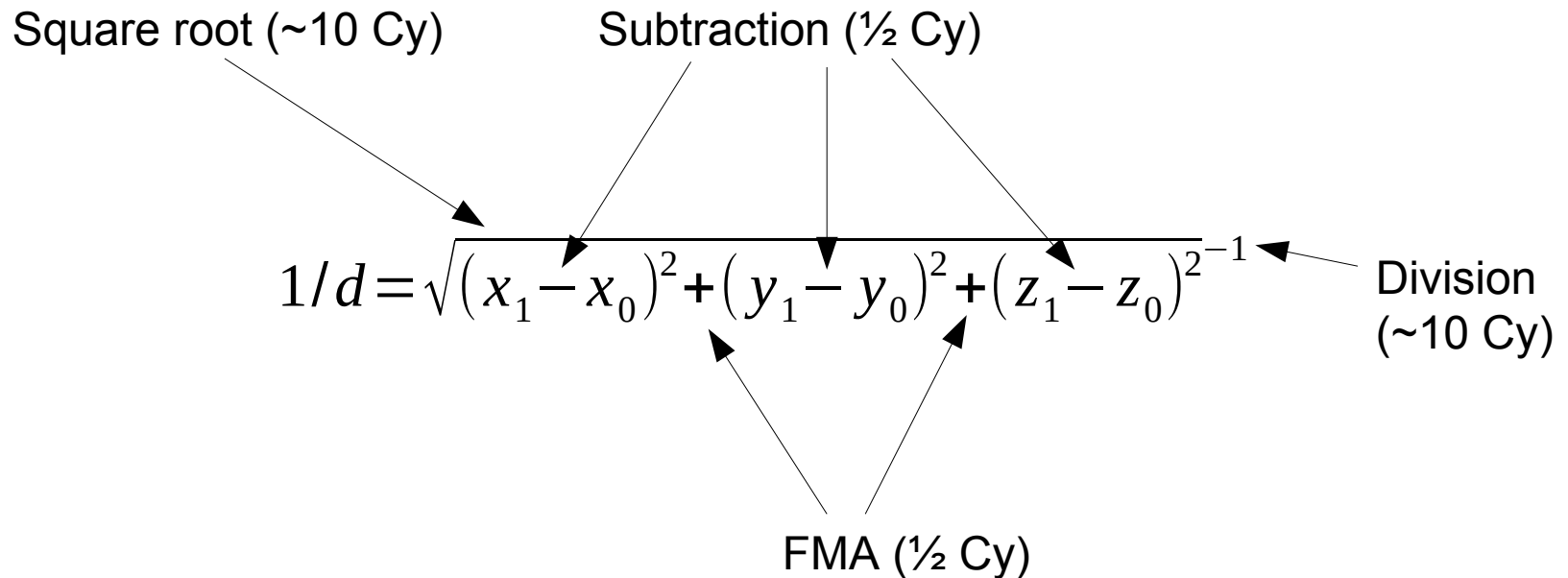
Calculation is dominated by sum of forces.  
The sum of forces is dominated by the reciprocal distance:

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

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

# 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).

