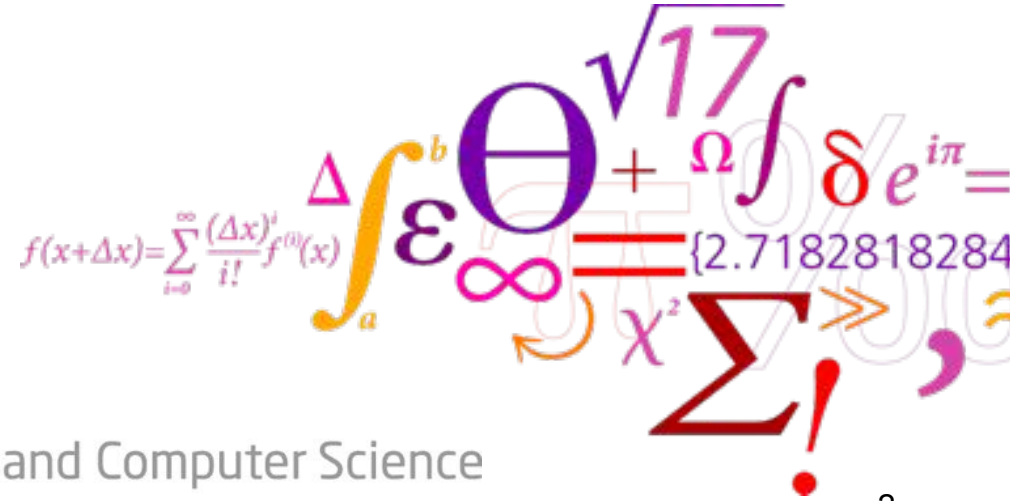



One kernel for CPU, Xeon Phi and GPU: is OpenMP 4.x the solution?

Bernd Dammann, Claudia Montoreano and Hans Henrik B. Sørensen
DTU Computing Center
<{beda,hhbs} (a) dtu.dk>

GPUlab
DTU Compute



DTU Compute
Department of Applied Mathematics and Computer Science

Who we are …

- DTU Computing Center
  - HPC operations
  - HPC Competence Center
    - advanced user support
    - exploring new technologies
- GPUlab
  - Scientific Computing on GPUs
    - founded in 2007, by B. Dammann, A.P. Engsig-Karup and others
    - funded by Danish Council for Independent Research FTP (2010-2013)
    - integrated part of DTU CC since 2013

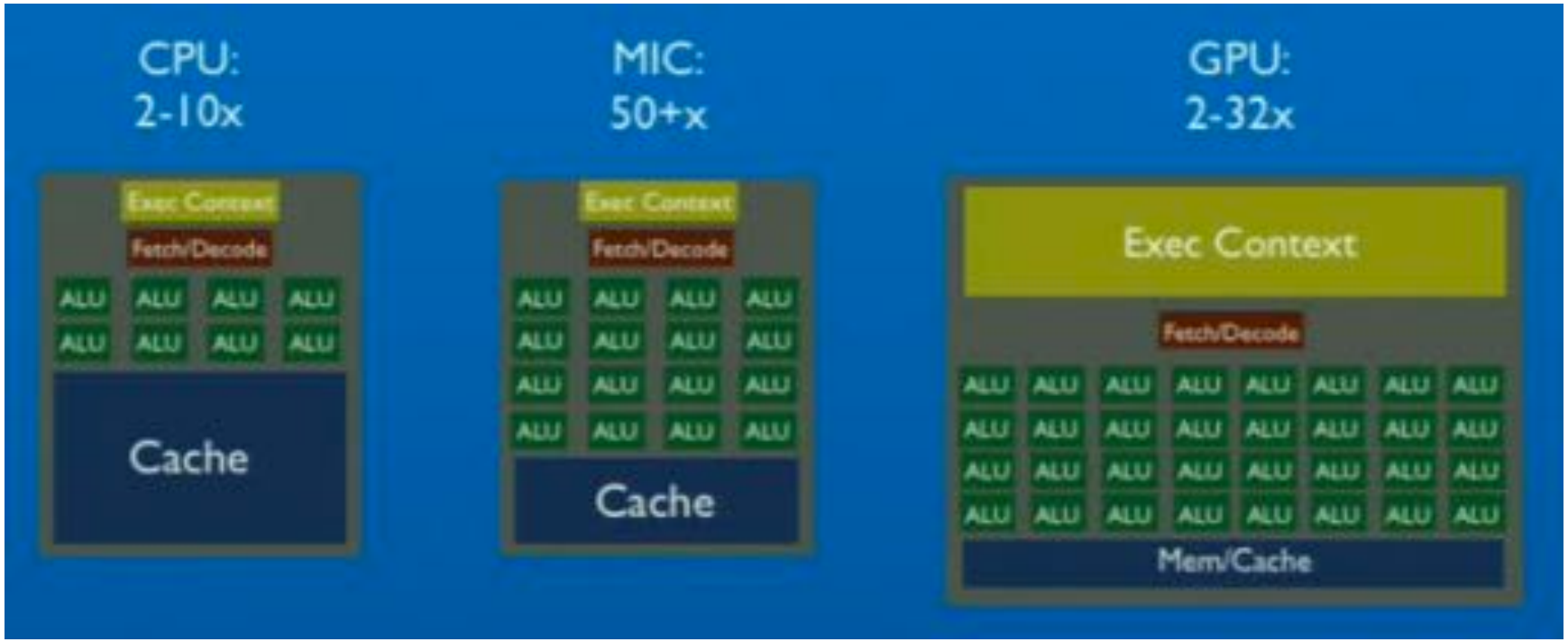
Outline

1. Motivation
2. Vectorization
   1. SIMD on CPU/Xeon Phi
   2. SIMT on GPU
3. CASE 1: Mandelbrot
4. CASE 2: Tomographic reconstruction
5. Conclusions – Future Work

- Note: this is work in progress!

Motivation: 3 “similar” architectures

- A “core” is a vector unit on modern processors



Running @2.6Ghz, @1.05Ghz, and @0.75Ghz, respectively.

Motivation: Getting performance

- High performance on modern vector units
  - Data parallelism
  - Sequential memory accessing
  - No branch divergence
- Intel's key sales point for Xeon Phi vs. GPU
  - Much easier – use by #pragma or compiler option only
  - No code transformation necessary (out-of-the-box)

However, our experience:

- Actual programming effort and approach needed to get the best performance is quite similar on all processors!

Questions to be asked

- can we make use of the CPU core vector units in a similar way as we do on a GPU?
- can we make use of our experiences with optimizing code for GPUs to optimize code on the CPU cores, using vectorization?
- can we write code, that will compile on all platforms (portability)?
- how well does this portable code perform?
- are OpenMP's 4.x 'SIMD' constructs the way to go?

Using vectorization on CPU and Xeon Phi

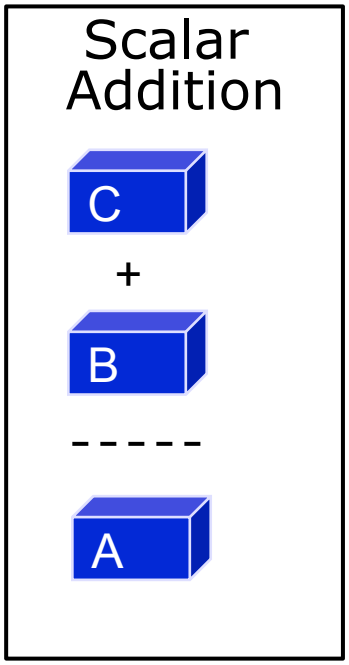
Vectorization

- Vectorization is the process of transforming a scalar operation that acts on single data elements at a time (Single Instruction Single Data – SISD), to an operation that that acts on multiple data elements at a time (Single Instruction Multiple Data – SIMD).

Source: Bob Chesebrough, “Performance Essentials 1: OpenMP 4 Vectorization Motivation“, Intel, 2013

Vectorization

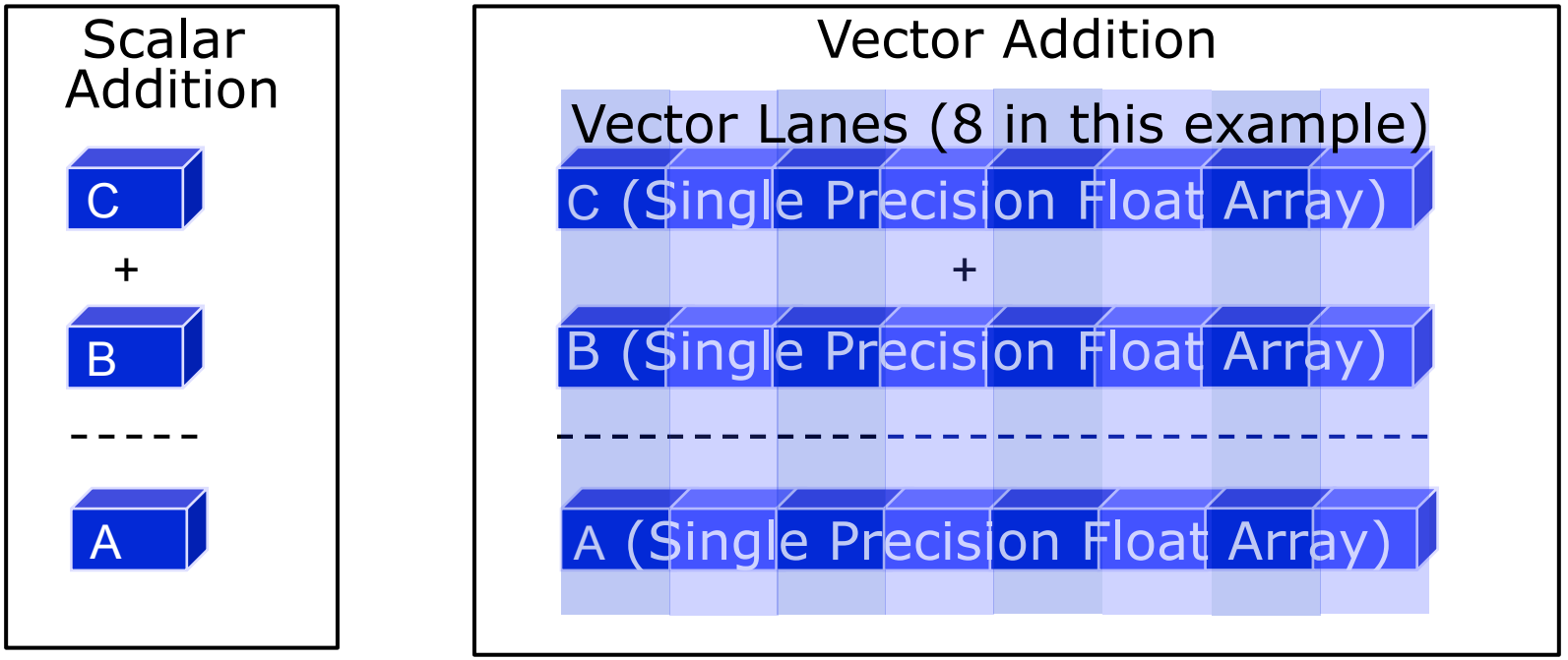
- Vectorization is the process of transforming a scalar operation that acts on single data elements at a time (Single Instruction Single Data – SISD), to an operation that that acts on multiple data elements at a time (Single Instruction Multiple Data – SIMD).



Source: Bob Chesebrough, “Performance Essentials 1: OpenMP 4 Vectorization Motivation“, Intel, 2013

Vectorization

- **Vectorization** is the process of transforming a scalar operation that acts on single data elements at a time (Single Instruction Single Data – SISD), to an operation that that acts on multiple data elements at a time (Single Instruction Multiple Data – SIMD).



Source: Bob Chesebrough, "Performance Essentials 1: OpenMP 4 Vectorization Motivation", Intel, 2013

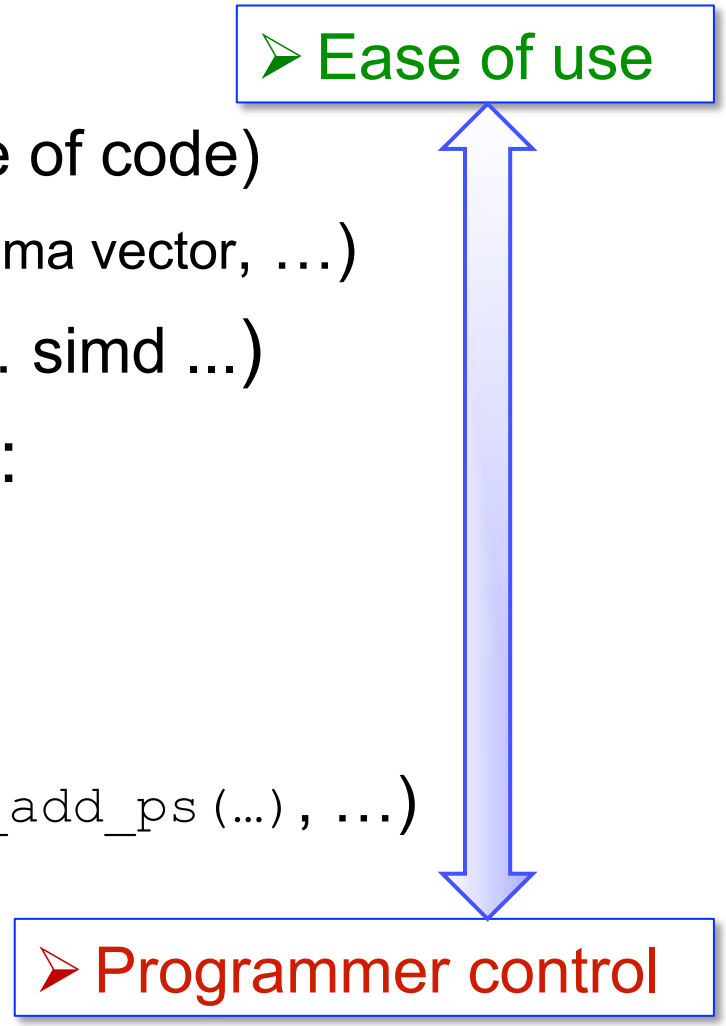
SIMD vector length

	Intel® Xeon® processor 64-bit	Intel® Xeon® processor 5100 series	Intel® Xeon® processor 5500 series	Intel® Xeon® processor 5600 series	Intel® Xeon® processor code-named Sandy Bridge	Intel® Xeon® processor code-named Ivy Bridge	Intel® Xeon® processor code-named Haswell	Intel® Xeon Phi™ coprocessor code-named Knights Corner
Core(s)	1	2	4	6	8			57-61
Threads	2	2	8	12	16			228-244
SIMD Width	128	128	128	128	256	256	256	512
	SSE2	**SSSE3**	**SSE4.2**	**SSE4.2**	**AVX**	**AVX**	**AVX2 FMA3**	**IMCI**

Source: J. Jeffers, “Intel Xeon Phi Co-processor”, 2013

Ways to vectorize for CPU/Phi

- Compiler:
  - Auto-vectorization (no change of code)
  - Auto-vectorization hints (#pragma vector, …)
- OpenMP 4.x (#pragma omp … simd ...)
- Explicit vector programming:
  - SIMD intrinsic class
    (e.g.: `F32vec`, `F64vec`, …)
  - Vector intrinsics
    (e.g.: `_mm_fmadd_pd(…)`, `_mm_add_ps(…)`, …)
  - Assembler code
    (e.g.: `[v]addps`, `[v]addss`, …)



Why always prefer the compiler?

- Easier and more readable code
- Portable across vendors and machines
  - … but compiler directives differ across compilers
  - OpenMP 4.x SIMD can (maybe?) solve this problem
- Better performance of the compiler generated code compared to explicit vector programmer
  - Compiler applies other transformations as well

However, we may need to help the compiler:

- Programmers may need to provide the necessary information (alignment, aliasing, inline)
- Programmers may need to transform the code

The OpenMP SIMD constructs

- The SIMD construct enables the execution of multiple iterations of the associated loops concurrently by means of SIMD instructions.

C/C++:
```
#pragma omp simd [clause(s)]
  for-loops
```

Fortran:
```
!$omp simd [clause(s)]
  do-loops
[!$omp end simd]
```

The OpenMP SIMD constructs

- The loop SIMD construct specifies a loop that can be executed in parallel by all threads and in SIMD fashion on each thread.

C/C++:
```
#pragma omp for simd [clause(s)]
  for-loops
```

Fortran:
```
!$omp do simd [clause(s)]
  do-loops
[!$omp end do simd [nowait]]
```

- Loop iterations are first distributed across threads, then each chunk is handled as a SIMD loop.

The OpenMP SIMD constructs

- Function calls in SIMD-loops can lead to bottle-necks, because functions need to be executed serially.
- “omp declare simd” allows the creation of multiple versions of a function or subroutine, where one or more versions can process multiple arguments using SIMD instructions.

C/C++:
```
#pragma omp declare simd [clause(s)]
[#pragma omp declare simd [clause(s)]]
  function definition / declaration
```

Fortran:
```
!$omp declare simd (proc_name)[clause(s)]
```

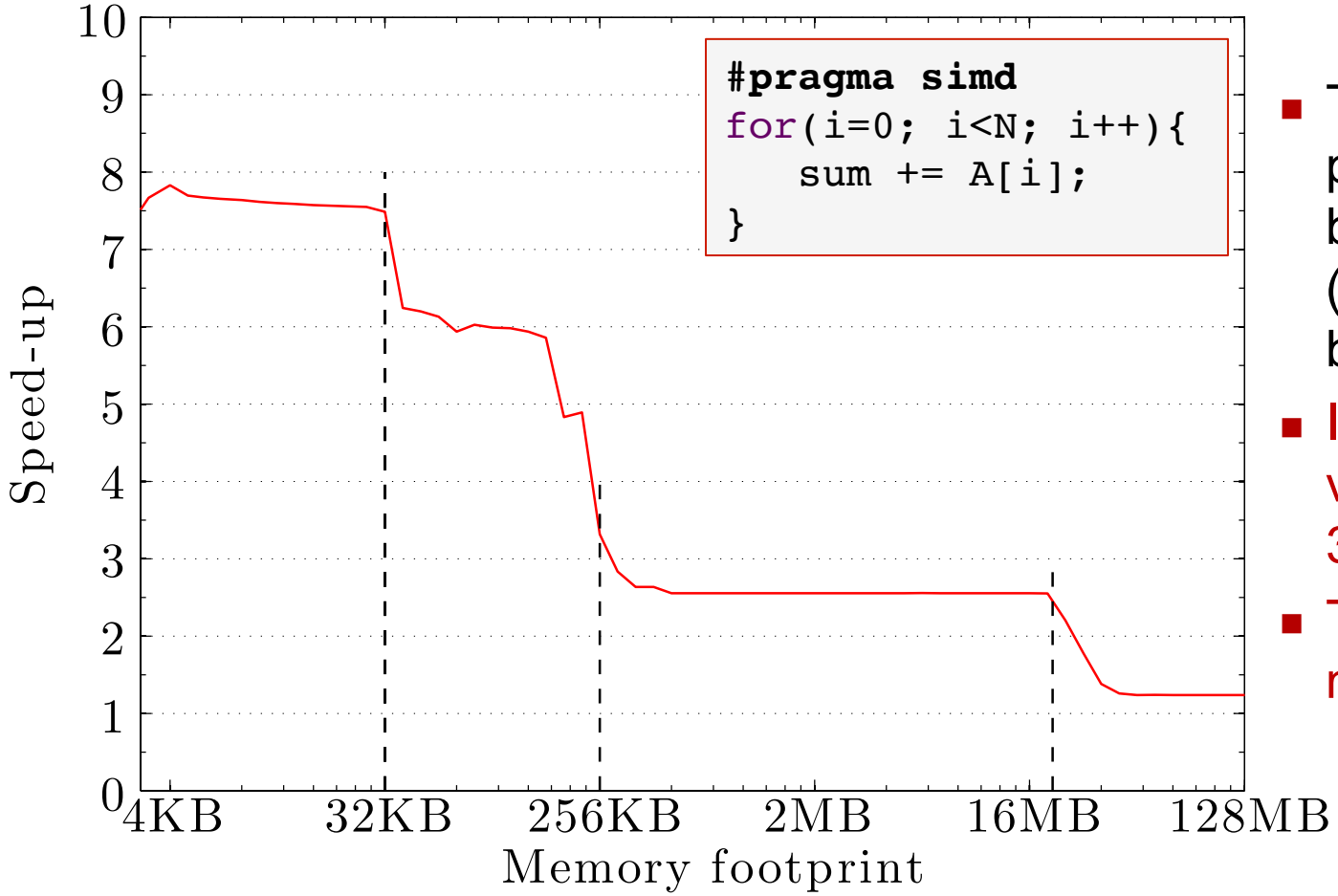
- “user defined” vectorized functions

How to assess the speed-up?

- Run application with full optimization options
- Compile with same optimizations, add options:
  - "`-no-vec -no-simd`" (and "`-no-openmp-simd`")
- Compare speed-up from vectorization
  - Speed-up(S) = Time(no-vec) / Time (vec)
  - Should be ≥ 1.0.
  - Ceilings for speed-up(S):
    - float: S ≤ 4 for SSE, S ≤ 8 for AVX, S ≤ 16 for MIC
    - double: S ≤ 2 for SSE, S ≤ 4 for AVX, S ≤ 8 for MIC
    - Higher is better, try to reach the ceiling

SIMD speed-up I

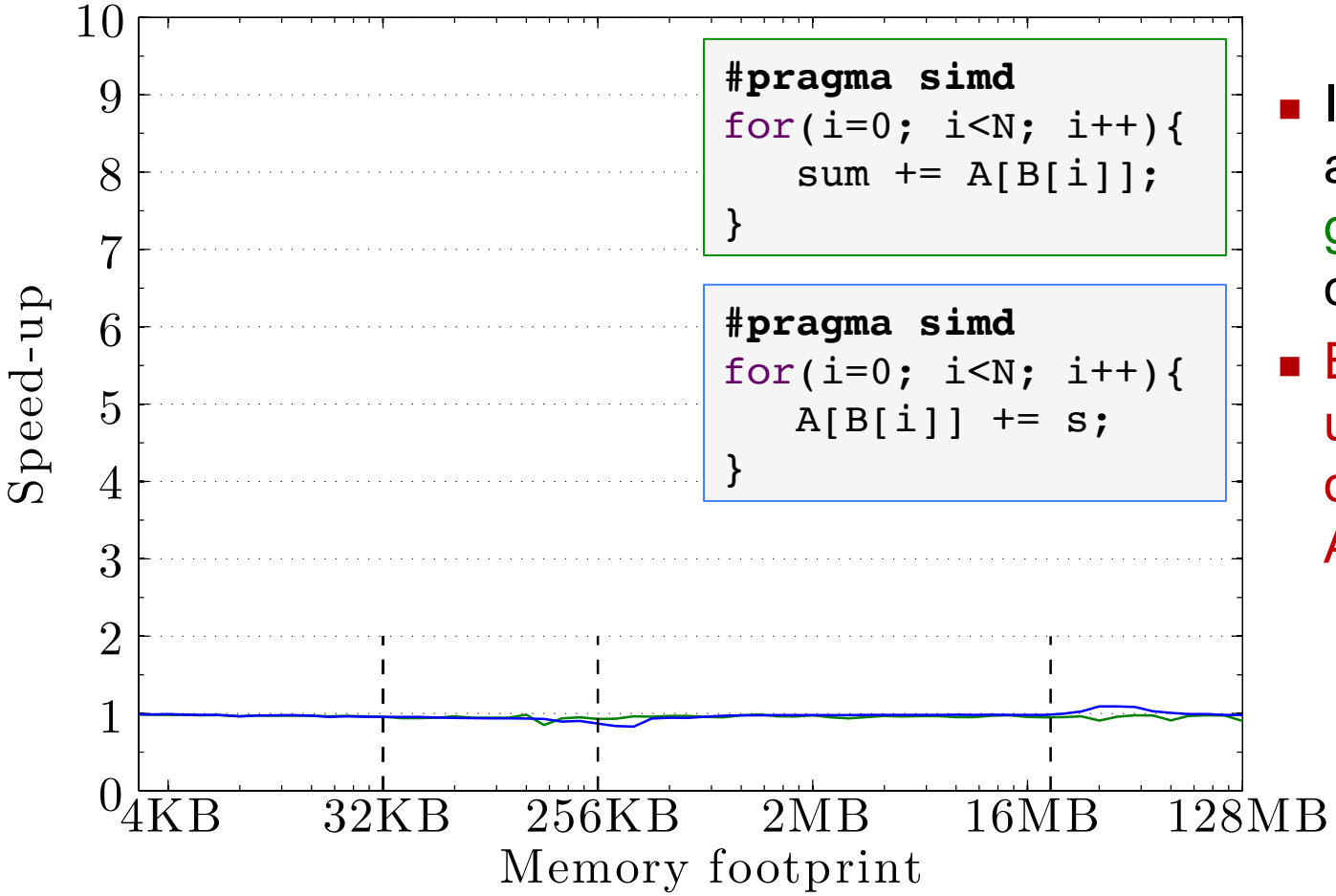
- SIMD impact decreases for data far away



- Typical speed-up profile for memory bound kernels (overhead has been deducted)
- Intel Xeon E5-2660 v3 @2.6GHz (turbo 3.3 GHz)
- Theoretical max memory 68 GB/s

SIMD speed-up II

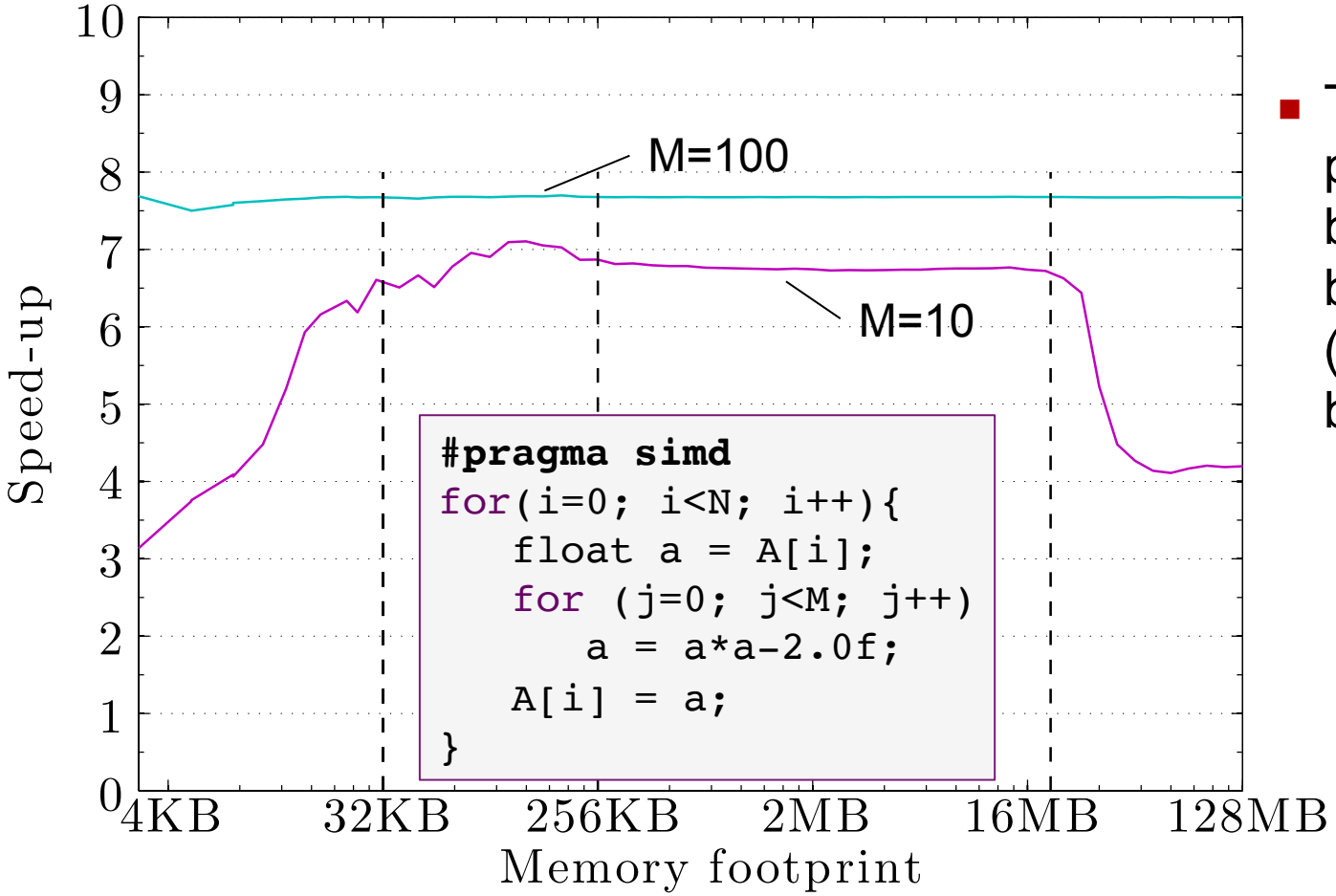
- Indirect memory accessing / “Data divergence”



- Indirect memory accesses produce gather and scatter operations
- Eliminates speed-up completely (also on Haswell using AVX2)

SIMD speed-up III

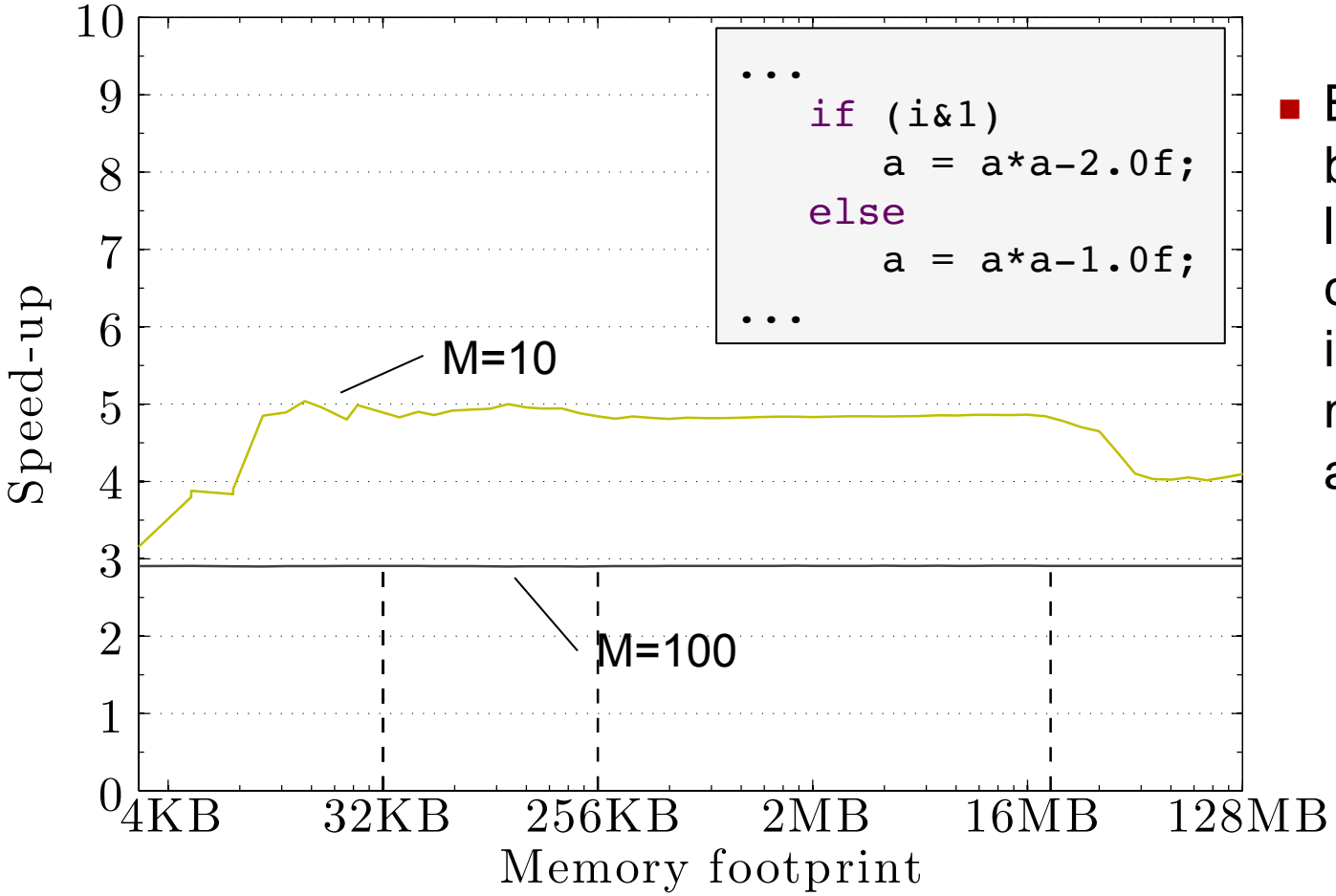
- Compute bound kernels can reach ceiling



- Typical speed-up profile for compute bound and close to balanced kernels (overhead has been deducted)

SIMD speed-up IV

- Branch divergence in the control flow



- Branch divergence between SIMD lanes results in costly blend instructions or masked memory accesses.

Vectorization on GPU

SIMT execution model

- Single Instruction Multiple Threads (SIMT)
  - Programmer writes code for a single thread
  - Each thread appears to have its own program counter
  - Branches look natural
- GPU hardware implementation of SIMT
  - Instructions are grouped to SIMD vector instructions
  - Memory accesses are dynamically coalesced from the individual loads and stores issued by the threads
  - Actually only one program counter per 32 threads
  - Simple branches implemented with vector masking

How to write SIMT code in CUDA

Compute bound loop

```
for(i=0; i<N; i++){
   float a = A[i];
   for (j=0; j<M; j++)
      a = a*a-2.0f;
   A[i] = a;
}
```

How to write SIMT code in CUDA

Compute bound loop

```
for(i=0; i<N; i++){
    float a = A[i];
    for (j=0; j<M; j++)
        a = a*a-2.0f;
    A[i] = a;
}
```

Simply peel off outer loop and compute `i` as the corresponding unique thread id

Kernel:

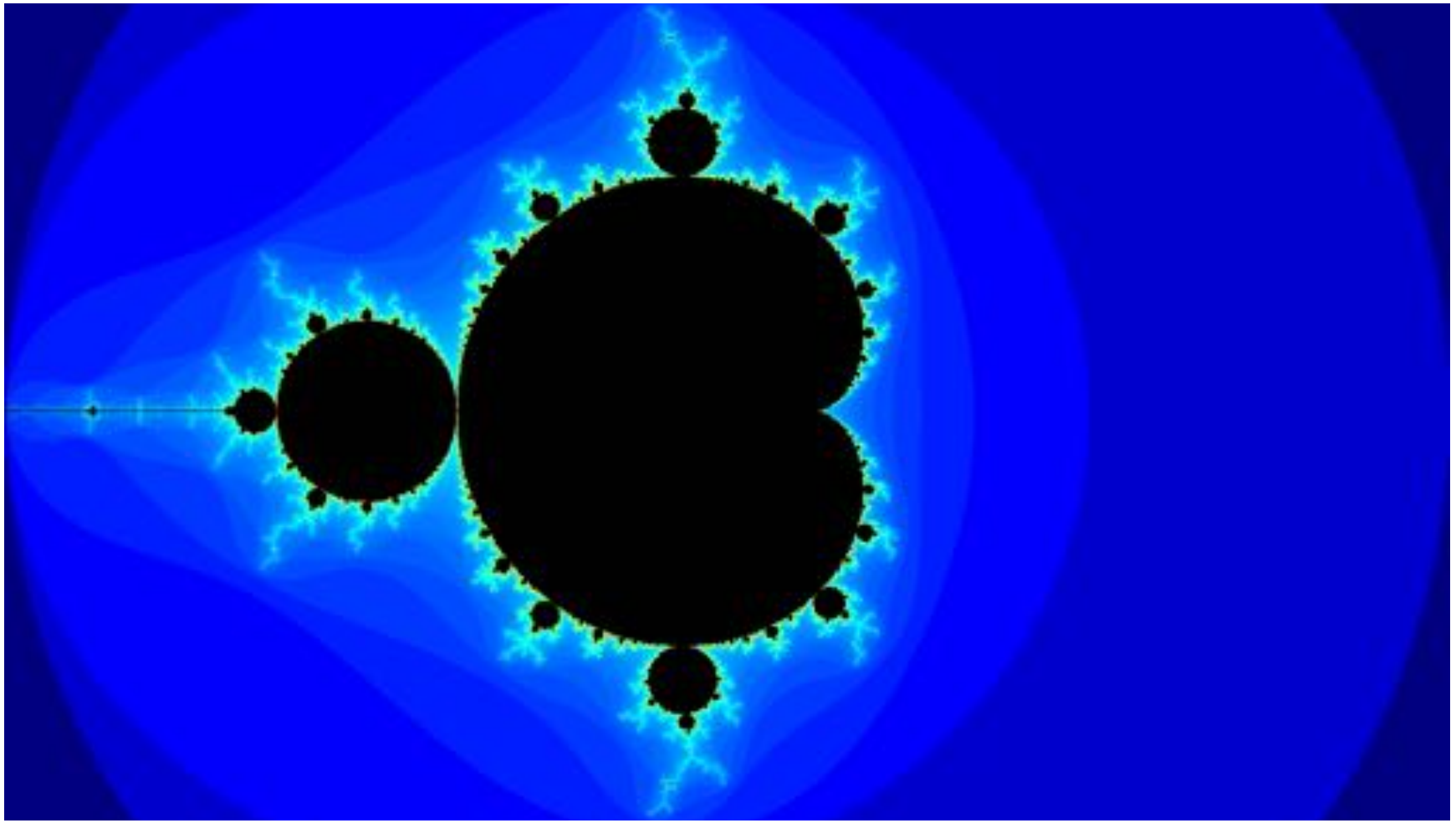
```
long i = threadIdx.x+blockIdx.x*blockDim.x;
float a = A[i];
for (j=0; j<M; j++)
    a = a*a-2.0f;
A[i] = a;
```

Inner code is unchanged!

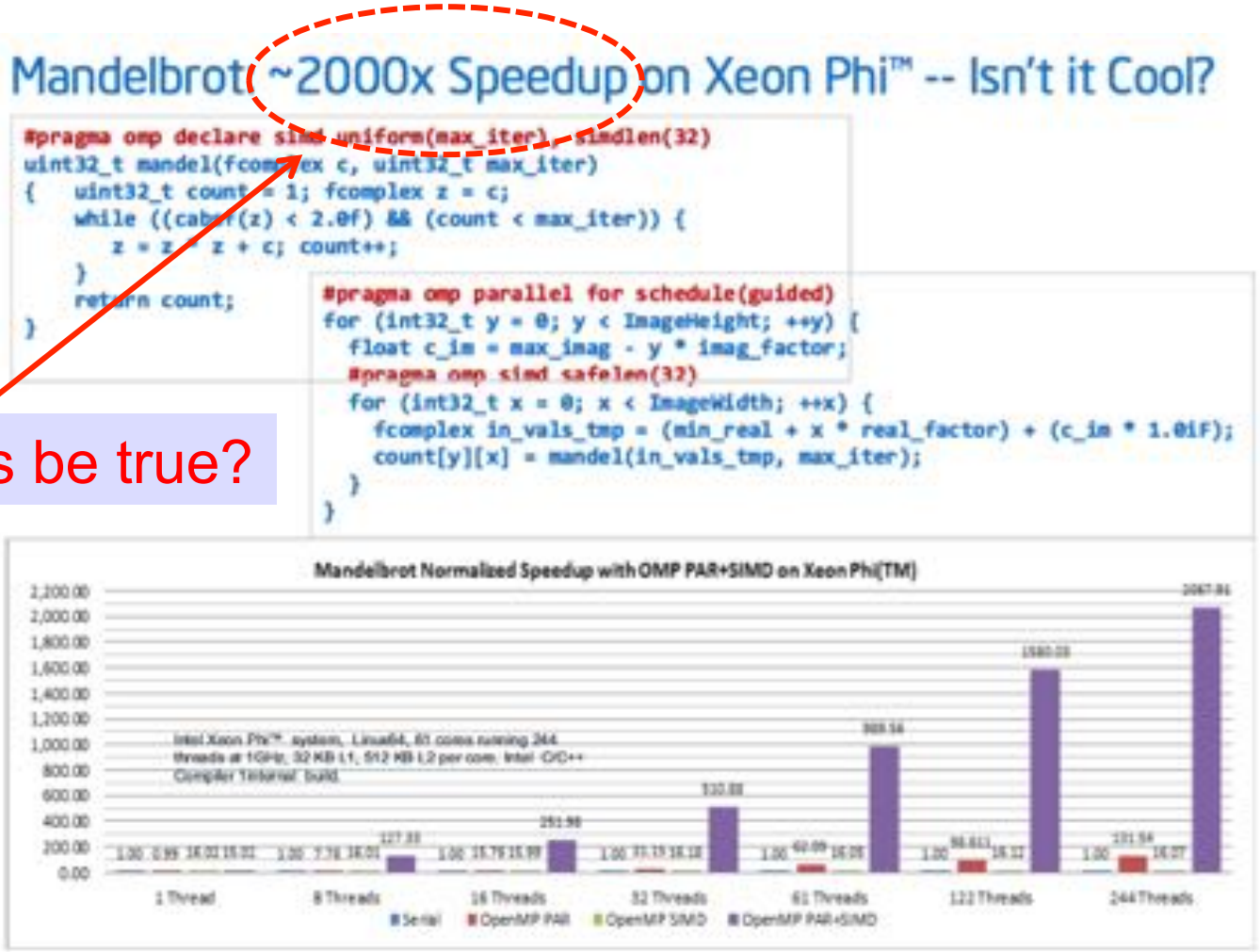
Launch:

```
kernel<<<blocks, threads>>>(A,M);
```

CASE: Mandelbrot



Motivation



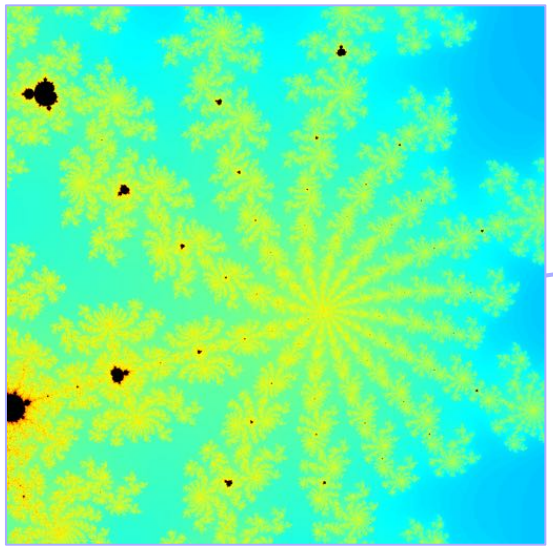
Can this be true?

courtesy: Xinmin Tian, Principal Engineer, Intel (LCPC 2014)

Runtime is position dependent

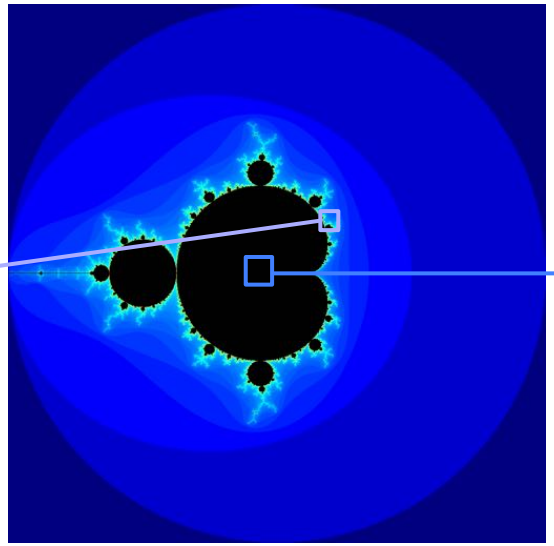
- Selective benchmark positions

Detailed



(0.34,0.38) – (0.35, 0.39)

Standard



(-2.0,-2.0) – (2.0, 2.0)

Black (Max Iter)

(-0.1,-0.1) – (0.1, 0.1)

1024 x 1024 pixels
10000 iterations

Also max iteration dependent

5-step optimization (GPU style)

- v1: Baseline (complex arithmetic)

```
#pragma omp declare simd \
  uniform(max_iter) simdlen(8)
int mandel1(float x, float y, int max_iter) {
   int iter = 1; float complex z = c;
   while (cabs(z) < 2.0f && iter < max_iter)
   {
      z = z * z + c;
      iter++;
   }
   return iter;
}
```

		v1
Detailed	Scalar	0.142
	SIMD	0.074
	Speed-up	**1.92**
Standard	Scalar	0.883
	SIMD	0.312
	Speed-up	**2.83**
Black	Scalar	9.488
	SIMD	3.136
	Speed-up	**3.02**

Runtime in seconds

Speed-up from vectorization

5-step optimization (GPU style)

- v2: Remove expensive complex function calls

```
#pragma omp declare simd \
  uniform(max_iter) simdlen(8)
int mandel2(float x, float y, int max_iter) {
   int iter = 1; float u = x, v = y;
   while (u * u + v * v < 4.0f && iter < max_iter)
   {
      const float v0 = 2.0f * v * u;
      u = u * u - v * v + x;
      v = v0 + y;
      iter++;
   }
   return iter;
}
```

Note that we can avoid a `sqrt()` by using `4.0f` in the condition

		v1	v2
Detailed	Scalar	0.142	0.028
	SIMD	0.074	0.011
	Speed -up	**1.92**	**2.49**
Standard	Scalar	0.883	0.172
	SIMD	0.312	0.047
	Speed -up	**2.83**	**3.66**
Black	Scalar	9.488	1.856
	SIMD	3.136	0.475
	Speed -up	**3.02**	**3.91**

6x faster

5-step optimization (GPU style)

- v3: Avoid “lane divergence” on arithmetic ops

```
#pragma omp declare simd \
  uniform(max_iter) simdlen(8)
int mandel3(float x, float y, int max_iter) {
   int iter = 1; float u = x, v = y;
   while (u * u + v * v < 4.0f && iter < max_iter)
   {
      const float v1 = 2.0f * v * u + y;
      const float u1 = u * u - v * v + x;
      iter++;
      u = u1; v = v1;
   }
   return iter;
}
```

Explicitly show the compiler that it can do the arithmetic for all lanes (as long as `u` and `v` are not updated for those lanes that would have exited the while loop)

		v1	v2	v3
Detailed	Scalar	0.142	0.028	0.024
	SIMD	0.074	0.011	0.009
	Speed -up	**1.92**	**2.49**	**2.71**
Standard	Scalar	0.883	0.172	0.149
	SIMD	0.312	0.047	0.037
	Speed -up	**2.83**	**3.66**	**4.03**
Black	Scalar	9.488	1.856	1.605
	SIMD	3.136	0.475	0.371
	Speed -up	**3.02**	**3.91**	**4.33**

5-step optimization (GPU style)

- v4: Unroll by hand (compiler cannot do this)

```
#pragma omp declare simd \
  uniform(max_iter) simdlen(8)
int mandel4(float x, float y, int max_iter) {
   int iter = 1; float u = x, v = y;
   while (u * u + v * v < 4.0f && iter < max_iter)
   {
      const float v1 = 2.0f * v * u + y;
      const float u1 = u * u - v * v + x;
      iter++;
      if (!(u1 * u1 + v1 * v1 < 4.0f)) break;
      const float v2 = 2.0f * v1 * u1 + y;
      const float u2 = u1 * u1 - v1 * v1 + x;
      iter++;
      if (!(u2 * u2 + v2 * v2 < 4.0f)) break;
      const float v3 = 2.0f * v2 * u2 + y;
      const float u3 = u2 * u2 - v2 * v2 + x;
      iter++;
      if (!(u3 * u3 + v3 * v3 < 4.0f)) break;
      const float v4 = 2.0f * v3 * u3 + y;
      const float u4 = u3 * u3 - v3 * v3 + x;
      iter++;
      u = u4; v = v4;
   }
   if (iter > max_iter) iter = max_iter;
   return iter;
}
```

		v1	v2	v3	v4
Detailed	Scalar	0.142	0.028	0.024	0.027
	SIMD	0.074	0.011	0.009	0.006
	Speed-up	**1.92**	**2.49**	**2.71**	**4.44**
Standard	Scalar	0.883	0.172	0.149	0.167
	SIMD	0.312	0.047	0.037	0.025
	Speed-up	**2.83**	**3.66**	**4.03**	**6.73**
Black	Scalar	9.488	1.856	1.605	1.796
	SIMD	3.136	0.475	0.371	0.249
	Speed-up	**3.02**	**3.91**	**4.33**	**7.21**

5-step optimization (GPU style)

- v5: Utilize more registers

```
#pragma omp declare simd \
  uniform(max_iter) simdlen(16)
int mandel5(float x, float y, int max_iter) {
   int iter = 1; float u = x, v = y;
   while (u * u + v * v < 4.0f && iter < max_iter)
   {
      const float v1 = 2.0f * v * u + y;
      const float u1 = u * u - v * v + x;
      iter++;
      if (!(u1 * u1 + v1 * v1 < 4.0f)) break;
      const float v2 = 2.0f * v1 * u1 + y;
      const float u2 = u1 * u1 - v1 * v1 + x;
      iter++;
      if (!(u2 * u2 + v2 * v2 < 4.0f)) break;
      const float v3 = 2.0f * v2 * u2 + y;
      const float u3 = u2 * u2 - v2 * v2 + x;
      iter++;
      if (!(u3 * u3 + v3 * v3 < 4.0f)) break;
      const float v4 = 2.0f * v3 * u3 + y;
      const float u4 = u3 * u3 - v3 * v3 + x;
      iter++;
      u = u4; v = v4;
   }
   if (iter > max_iter) iter = max_iter;
   return iter;
}
```

Breaking the speed-up barrier

		v1	v2	v3	v4	v5
Detailed	Scalar	0.142	0.028	0.024	0.027	0.026
	SIMD	0.074	0.011	0.009	0.006	0.005
	Speed-up	**1.92**	**2.49**	**2.71**	**4.44**	**5.11**
Standard	Scalar	0.883	0.172	0.149	0.167	0.166
	SIMD	0.312	0.047	0.037	0.025	0.018
	Speed-up	**2.83**	**3.66**	**4.03**	**6.73**	**9.21**
Black	Scalar	9.488	1.856	1.605	1.796	1.797
	SIMD	3.136	0.475	0.371	0.249	0.172
	Speed-up	**3.02**	**3.91**	**4.33**	**7.21**	**10.42**

Similar behavior for MIC and GPU

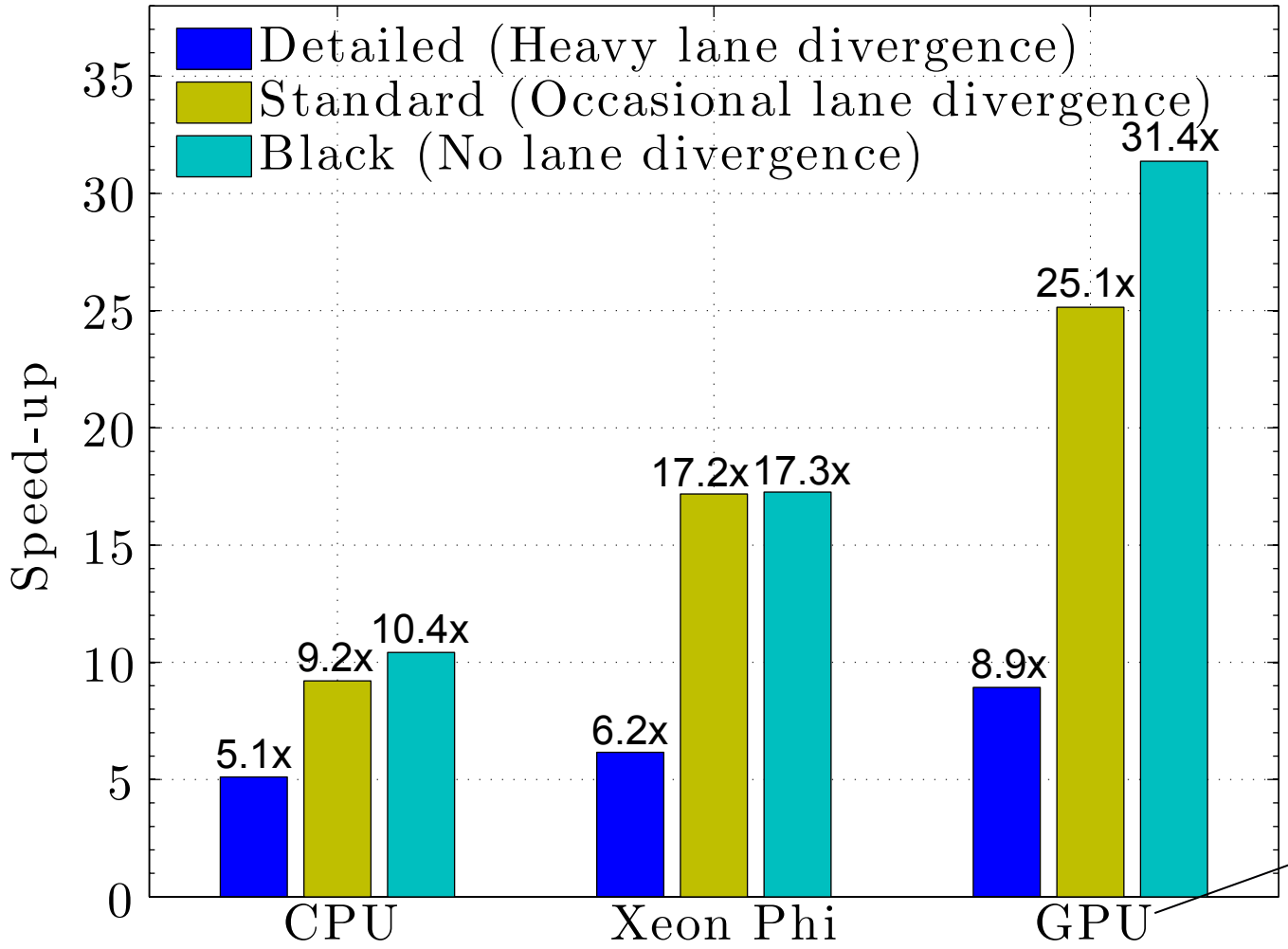
MIC

		v1	v2	v3	v4	v5
Detailed	Scalar	0.334	0.051	0.046	0.044	0.043
	SIMD	0.072	0.009	0.008	0.006	0.007
	Speed -up	**4.61**	**5.78**	**5.64**	**7.11**	**6.16**
Standard	Scalar	2.296	0.309	0.278	0.274	0.275
	SIMD	0.262	0.024	0.022	0.016	0.016
	Speed -up	**8.76**	**13.09**	**12.61**	**16.84**	**17.18**
Black	Scalar	16.50	2.592	2.388	2.296	2.284
	SIMD	2.106	0.211	0.199	0.143	0.132
	Speed -up	**7.83**	**12.31**	**11.98**	**16.06**	**17.26**

GPU

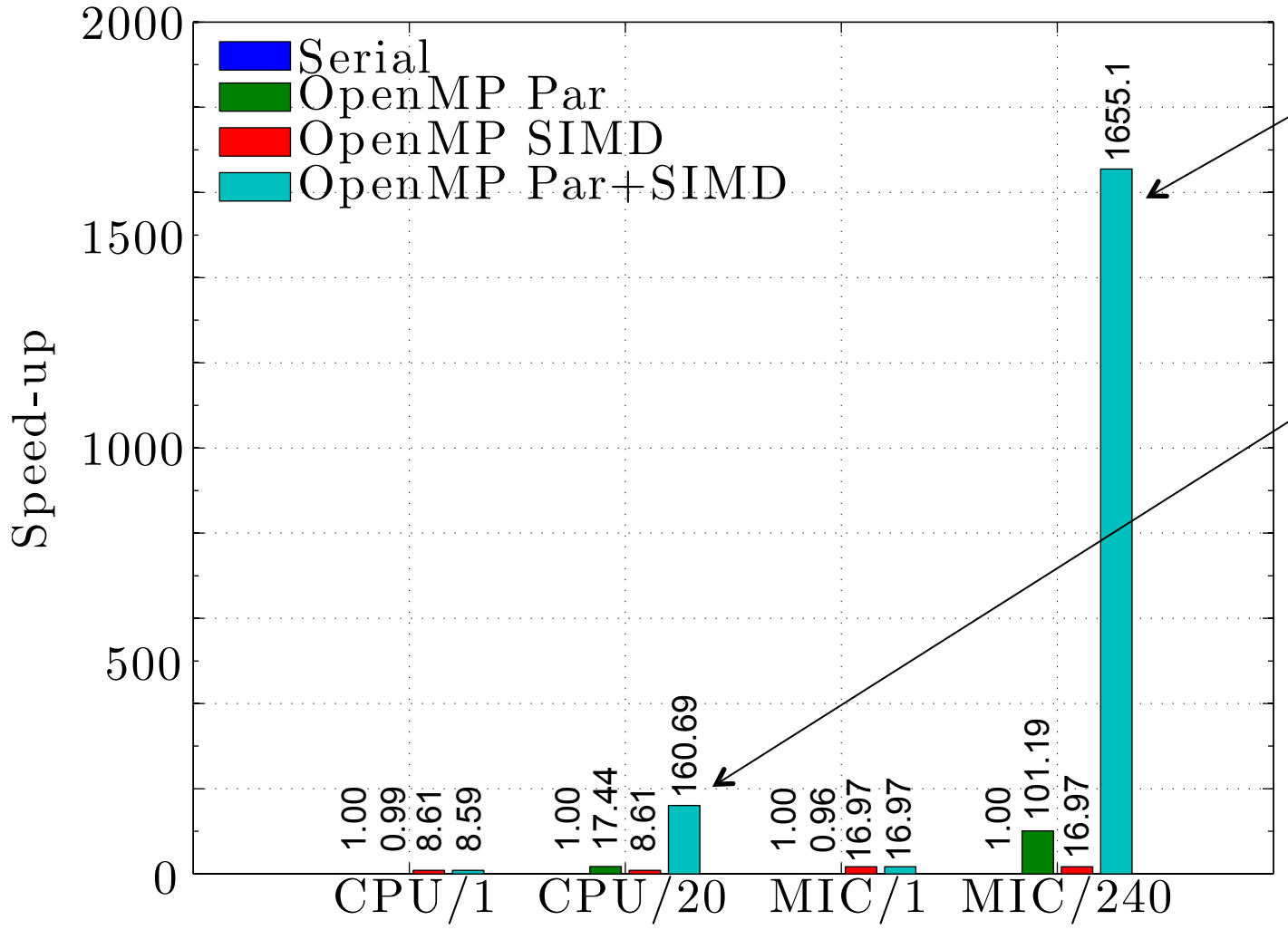
		v1	v2	v3	v4	v5
Detailed	Scalar	0.344	0.058	0.058	0.054	
	SIMD	0.039	0.006	0.006	0.006	
	Speed -up	**8.92**	**8.91**	**8.92**	**8.94**	
Standard	Scalar	1.148	0.218	0.218	0.214	
	SIMD	0.059	0.009	0.009	0.009	
	Speed -up	**24.13**	**24.66**	**24.63**	**25.01**	
Black	Scalar	13.04	2.046	2.051	2.044	
	SIMD	0.414	0.065	0.065	0.065	
	Speed -up	**31.46**	**31.41**	**31.45**	**31.45**	

Comparison of speed-up results



Speed-up is found by measuring time for 1 vs. 32 active threads per warp.

Speed-up on CPU & MIC



We get an equally good speed-up here

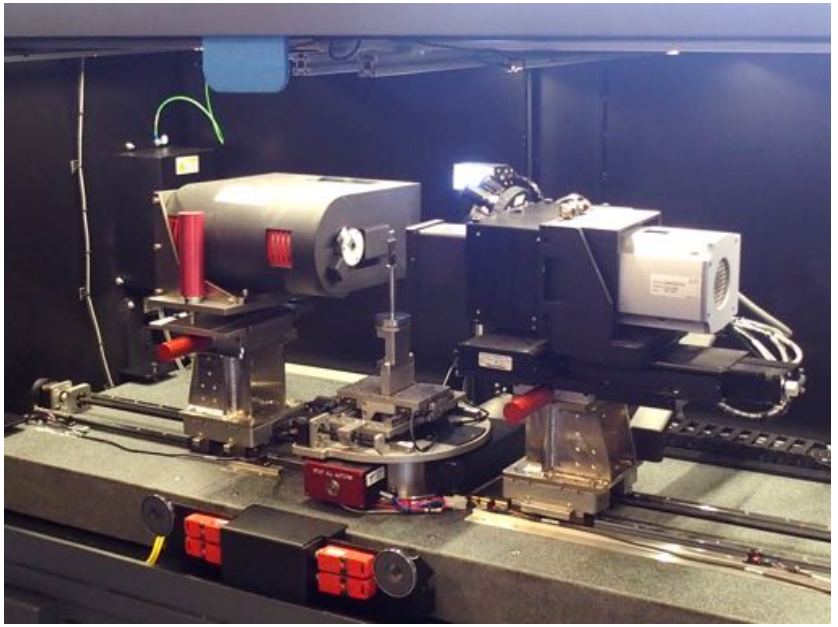
On the CPU system, we get full speed-up: 20 (cores) * 8 (vectorlength)

CPU: 2x Xeon E5-2660v3 @ 2.6GHz, 20 cores MIC: Xeon Phi 5110P @ 1.03GHz, 60 cores

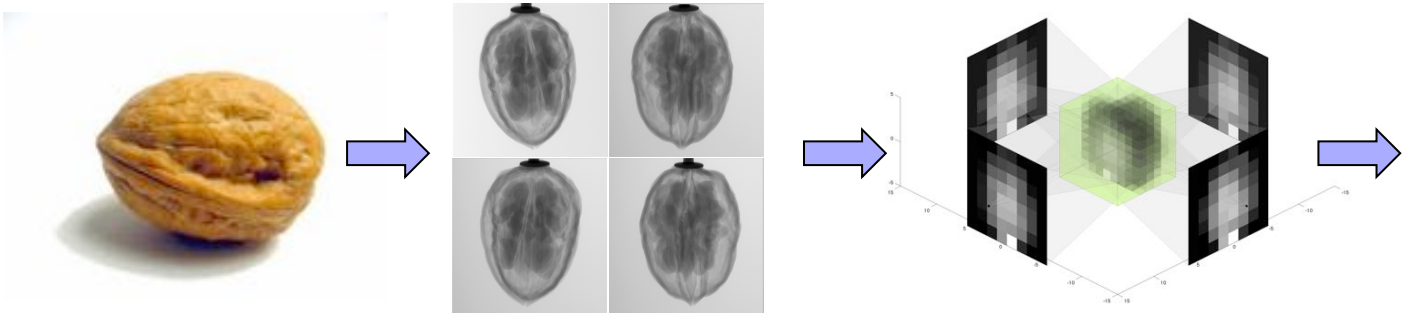
Mandelbrot study: conclusions

- we can speed-up the code by vectorizing the inner part of the Mandelbrot algorithm
- The new OpenMP 4.x SIMD constructs are useful here, especially the “omp declare simd” ➔ create a “kernel”, like in GPU computing
- applying optimization techniques we know from GPU computing help here as well
- passing in longer vectors than the vectorlength of the SIMD units is beneficial
- the speed-up on the CPU system is “optimal”

CASE: Tomographic reconstruction

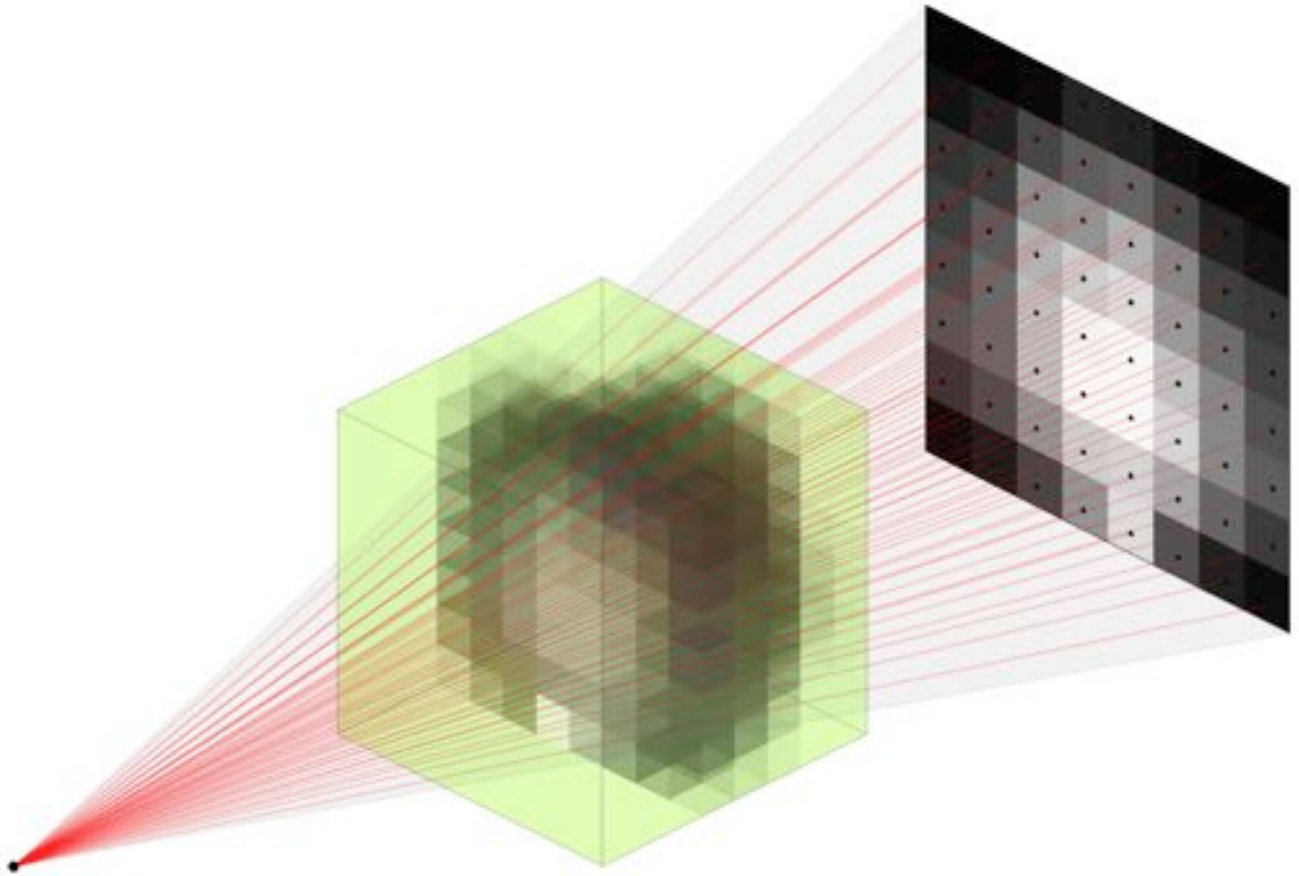


1600 projections (1024×1024)



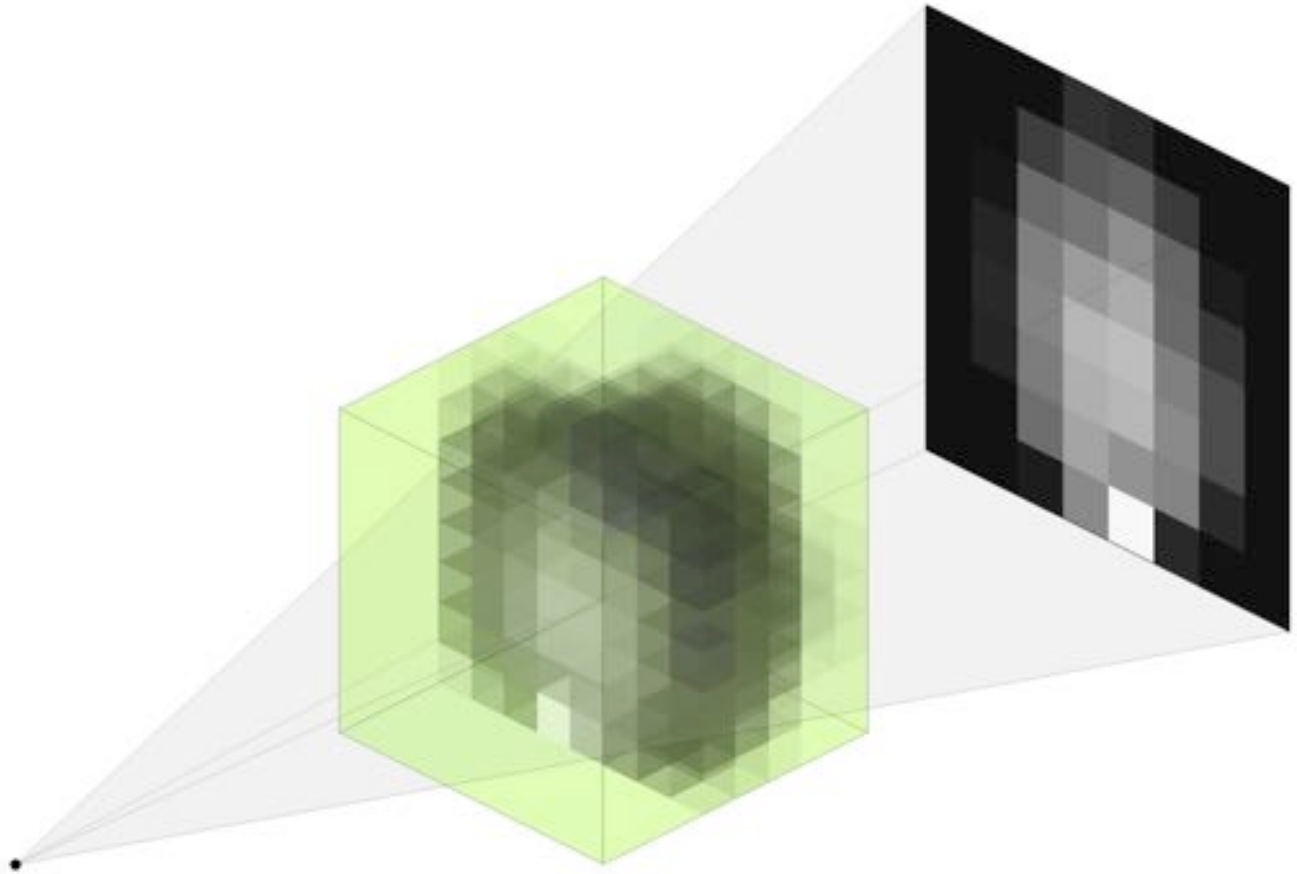
Step 1: Forward projection

- Raytrace through the solution object to calculate what the detector would show (=weighted sum)



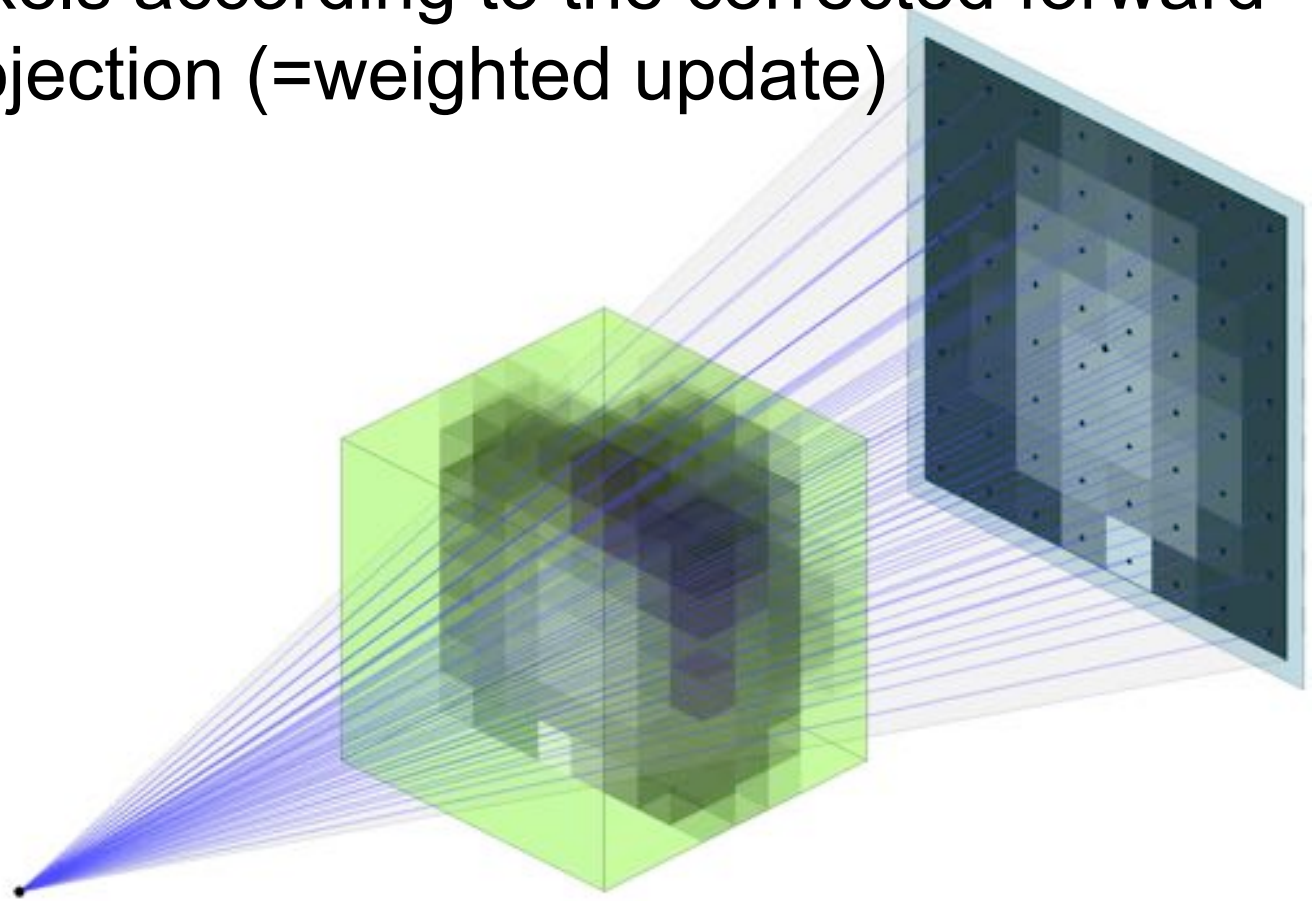
Step 2: Correction

- Subtract the measured detector image from the calculated forward projection (=vector add)



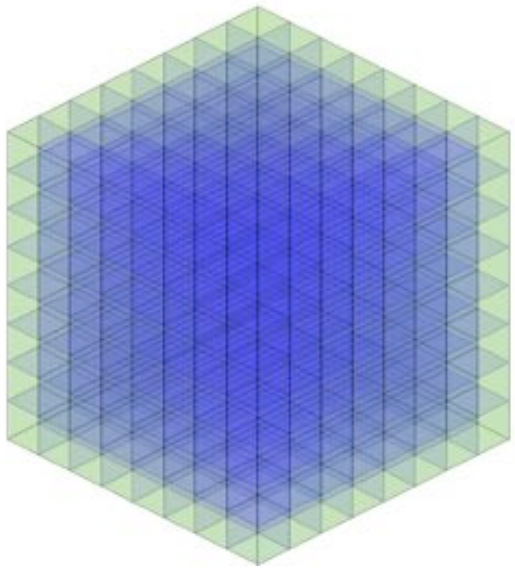
Step 3: Back projection

- Raytrace through the solution object and update voxels according to the corrected forward projection (=weighted update)



Blocking for caches I

- Standard HPC technique to utilize the memory hierarchy – works for CPU, Phi, and GPU!
- CASE: Tomographic reconstruction
  - We decompose the solution object into small cubes that fit the largest on-chip memory cache



Blocking for caches II

- For CPU we have:
  - Assume 20 hardware threads, 20 MB L3 cache
  - Every thread should have ~1 MB of data in order for all memory accesses to hit the L3 cache
  - Cubes of size: $N = (1024*1024/4)^{(1/3)} = 64$
- For Phi (4 threads / 512K L2) we have: $N = 32$
- For GPU (3 blocks / 48K L1) we have: $N = 16$

We use exactly the same data layout for all target processors – only the blocking size N varies

Kernel – original code

```
// Vectorization loop.
#pragma simd vectorlength(8)
for (int i = 0; i < 8; i++) {
    ...
    // Innermost step (original).
    if (tx[i] < ty[i])
        if (tx[i] < tz[i]) {
            ts[i] = tx[i]; tx[i] += sx[i]; voxel[i] += nx[i];
        } else {
            ts[i] = tz[i]; tz[i] += sz[i]; voxel[i] += nz[i];
        }
    else
        if (ty[i] < tz[i]) {
            ts[i] = ty[i]; ty[i] += sy[i]; voxel[i] += ny[i];
        } else {
            ts[i] = tz[i]; tz[i] += sz[i]; voxel[i] += nz[i];
        }
    ...
}
```

Kernel – portable among platforms

```
...
// Innermost step (portable).
float txy, nxy, dx, dy, dv;

txy = tx[i] < ty[i] ? tx[i] : ty[i];
nxy = tx[i] < ty[i] ? nx[i] : ny[i];
dx = tx[i] < ty[i] ? sx[i] : 0.0f;
dy = tx[i] < ty[i] ? 0.0f : sy[i];
ts[i] = txy < tz[i] ? txy : tz[i];
voxel[i] += txy < tz[i] ? nxy : nz[i];
tx[i] += txy < tz[i] ? dx : 0.0f;
ty[i] += txy < tz[i] ? dy : 0.0f;
tz[i] += txy < tz[i] ? 0.0f : sz[i];
...
```

Kernel – with AVX intrinsics

```
...
// Innermost step (using AVX intrinsics).
__m256 txy, nxy, dx, dy, dz, dv, cond, v0;

v0    = _mm256_xor_ps(v0, v0);
cond  = _mm256_cmp_ps(tx, ty, CMP_GE_OQ);
txy   = _mm256_blendv_ps(tx, ty, cond);
nxy   = _mm256_blendv_ps(nx, ny, cond);
dx    = _mm256_blendv_ps(sx, v0, cond);
dy    = _mm256_blendv_ps(v0, sy, cond);
cond  = _mm256_cmp_ps(txy, tz, CMP_GE_OQ);
dx    = _mm256_blendv_ps(dx, v0, cond);
dy    = _mm256_blendv_ps(dy, v0, cond);
dz    = _mm256_blendv_ps(v0, sz, cond);
ts    = _mm256_blendv_ps(txy, tz, cond);
dv    = _mm256_blendv_ps(nxy, nz, cond);
voxel = _mm256_add_ps(voxel, dv);
...
```

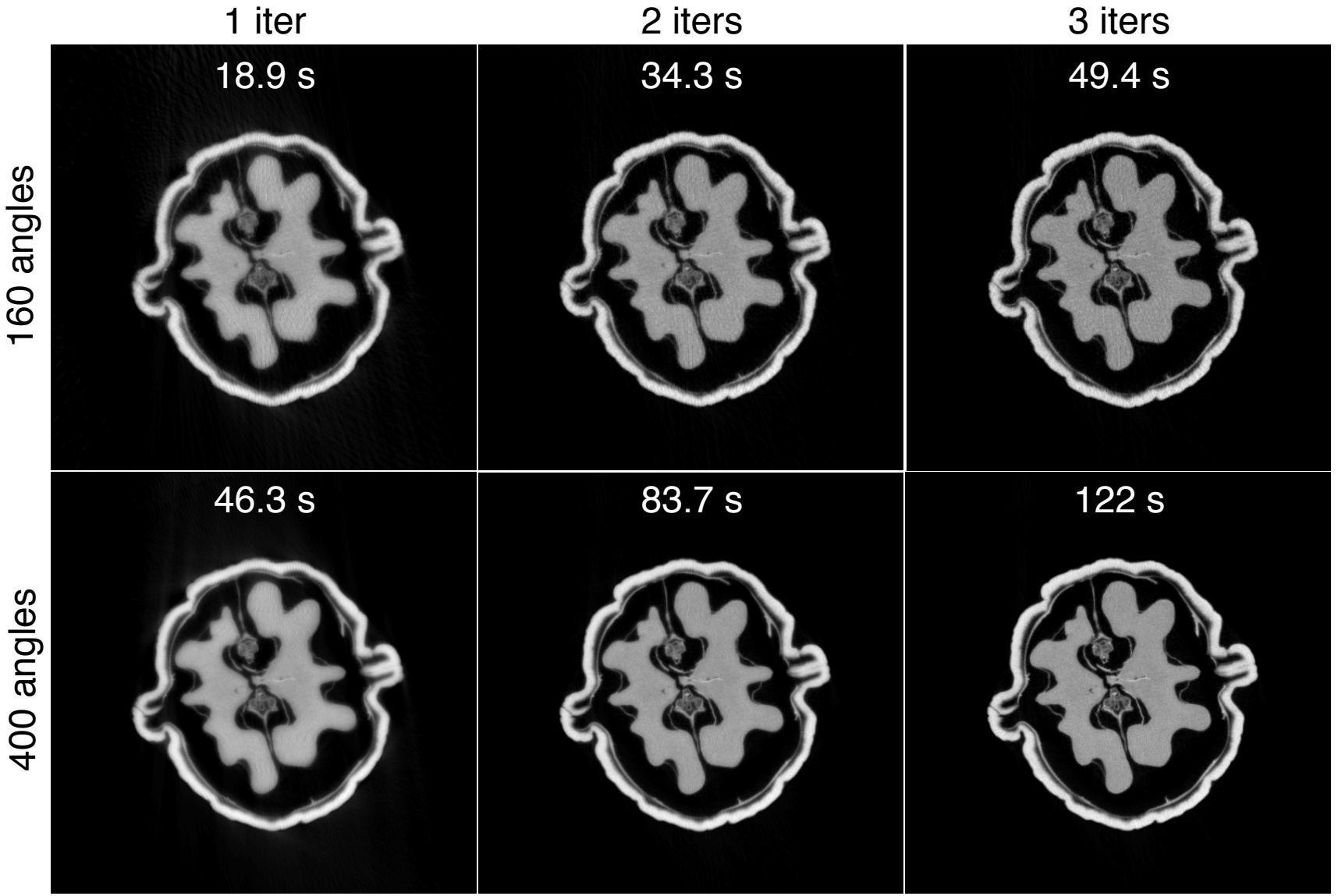
Kernel performance

- Tested on Intel Xeon E5-2665 (SandyBridge)
- Reference: code using AVX intrinsics

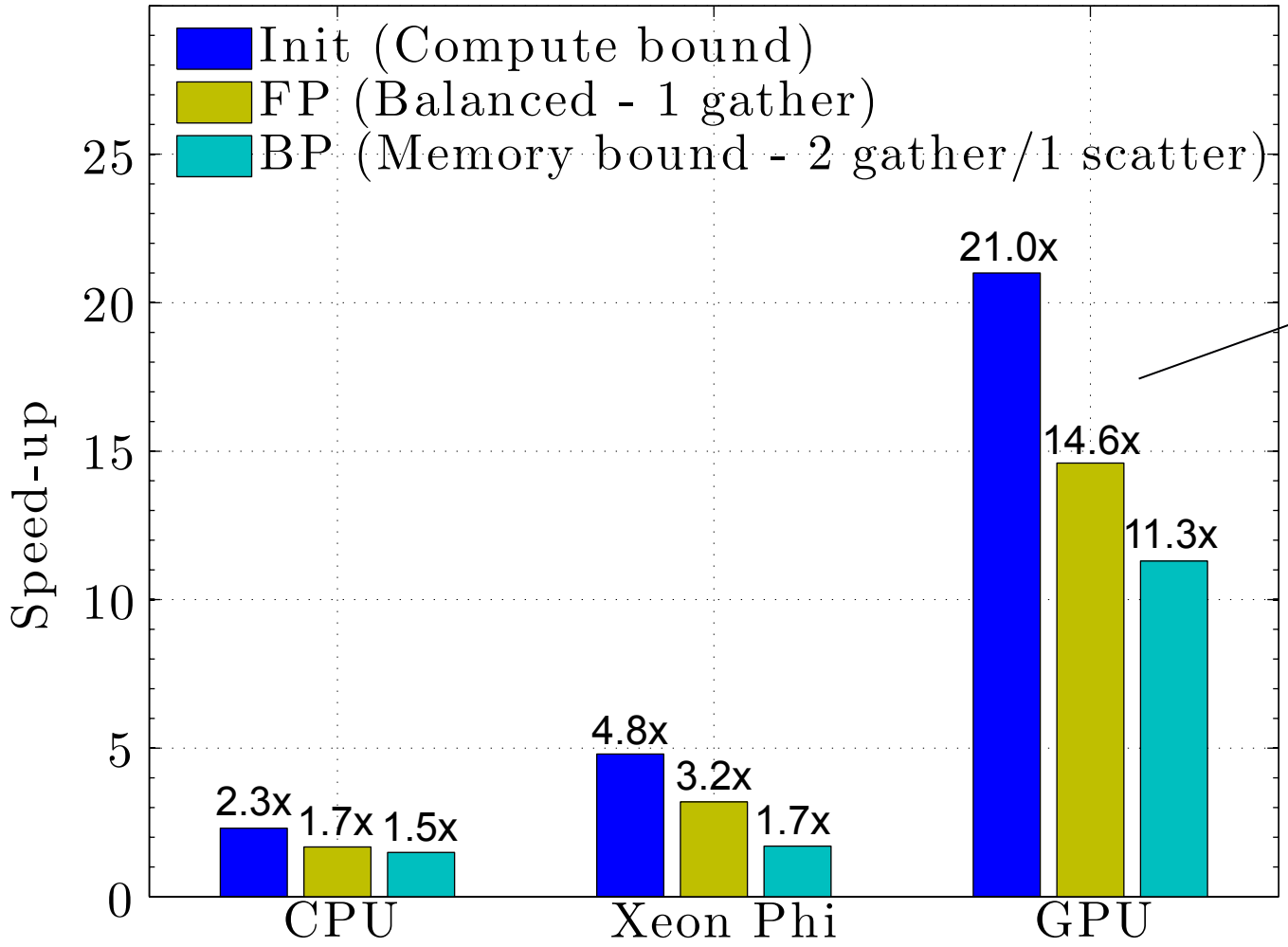
- Performance original code: ~31%
- Performance portable code: ~77%

- OK to sacrifice ~20% for portability!

Results of reconstruction (K40)

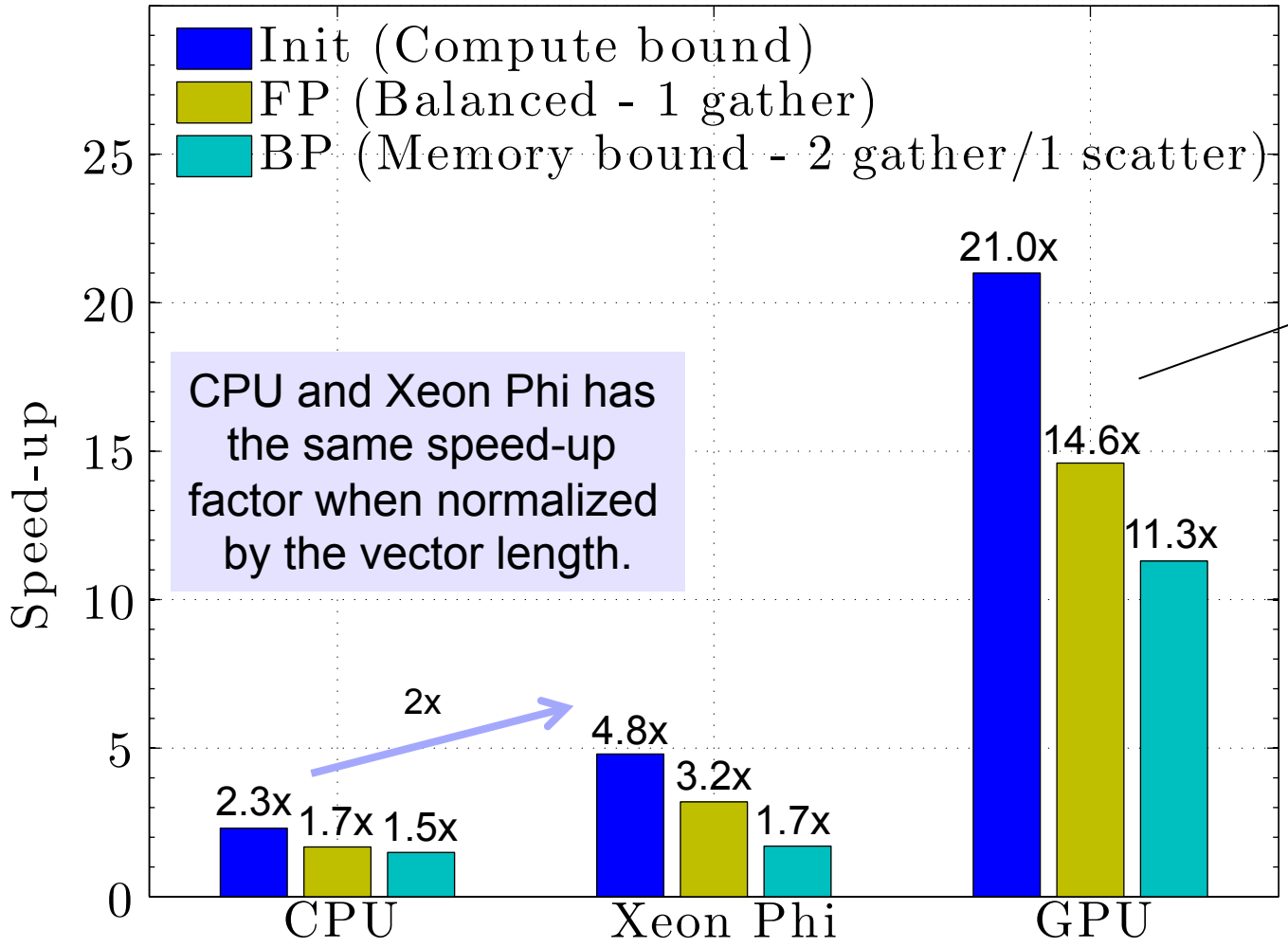


SIMD/SIMT speed-up results



Speed-up is found by measuring time for 1 vs. 32 active threads per warp.

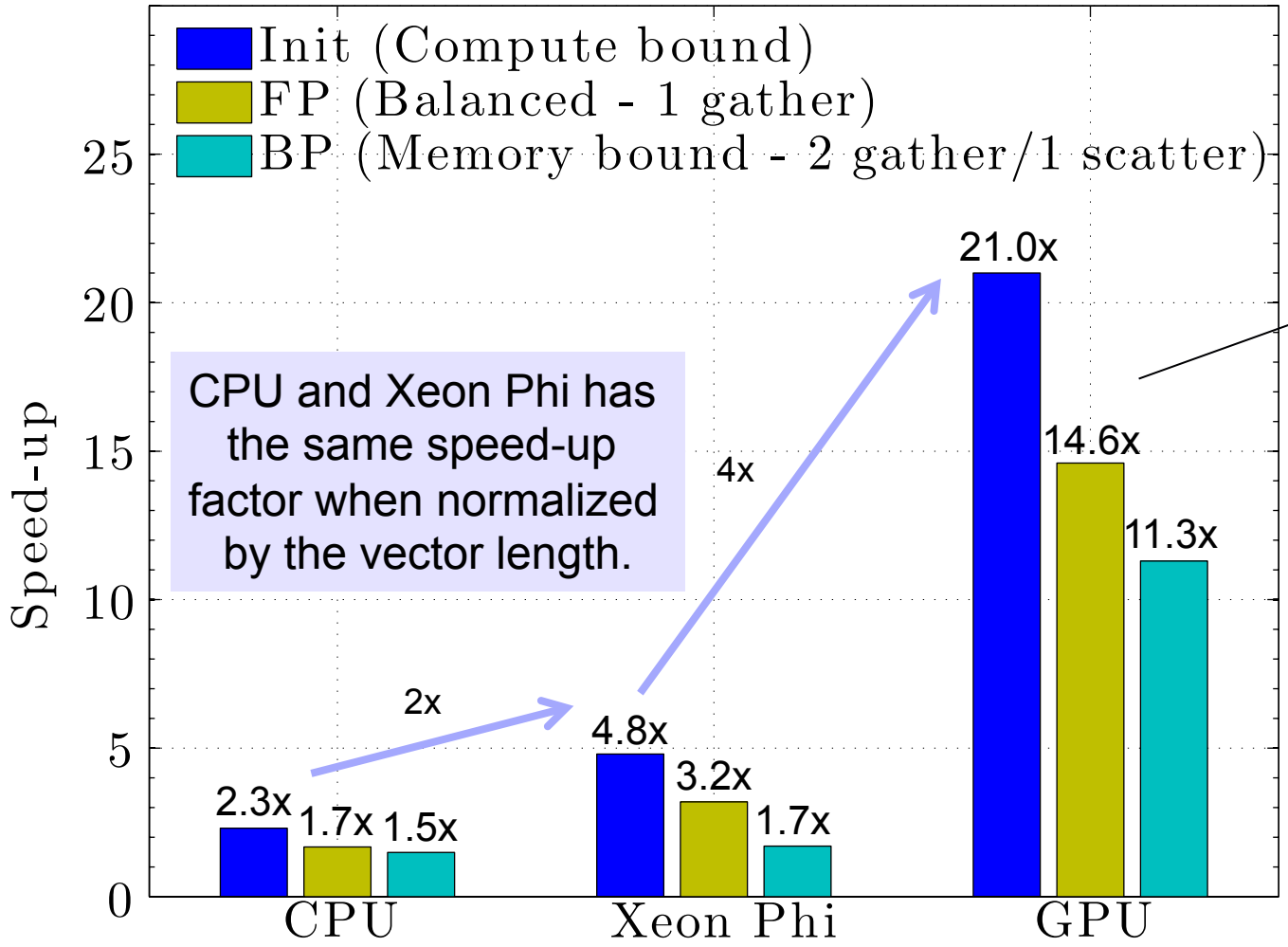
SIMD/SIMT speed-up results



CPU and Xeon Phi has the same speed-up factor when normalized by the vector length.

Speed-up is found by measuring time for 1 vs. 32 active threads per warp.

SIMD/SIMT speed-up results



Speed-up is found by measuring time for 1 vs. 32 active threads per warp.

CPU and Xeon Phi has the same speed-up factor when normalized by the vector length.

GPU gets and additional speed-up by a factor of 2 for its vector length

It seems a SIMT model running on SIMD hardware is a good approach

Runtime results I

- Total runtime in [s] minus I/O

GPU fastest, and Xeon Phi is not doing so well!

Voxels	128^3	256^3	512^3	1024^3	2048^3
Xeon E5-2660v3	0.783	5.54	38.7	316	2380
Xeon Phi 5110P	15.5	47.4	155	678	out of memory
GPU (K40)	0.837	4.35	31.3	242	

Runtime results II

- Total runtime in [s] minus I/O

GPU fastest, and Xeon Phi is not doing so well!

Voxels	128^3	256^3	512^3	1024^3	2048^3
Xeon E5-2660v3	0.783	5.54	38.7	316	2380
Xeon Phi 5110P	15.5	47.4	155	678	out of memory
GPU (K40)	0.837	4.35	31.3	242	

Well known from Top500: CPUs are generally utilizing resources better than accelerators!

	Stream BW
Xeon E5-2660	122 GB/s
Xeon Phi 5110P	170 GB/s
GPU (K40)	202 GB/s

- Normalized by stream BWs / CPU

Voxels	128^3	256^3	512^3	1024^3	2048^3
Xeon E5-2660v3	1	1	1	1	1
Xeon Phi 5110P	0.04	0.09	0.18	0.33	out of memory
GPU (K40)	0.57	0.77	0.75	0.78	

Conclusions – I

- Modern architectures are all based on SIMD units
  - Common issues that limit SIMD speed-up
    - Decreases with distance to data
    - Indirect memory accessing
    - Branch divergence within SIMD lanes
- Getting high performance
  - General approach is similar on all processors
  - Specific optimization is equally time consuming
- Writing good code
  - A well written SIMD code for CPU / Xeon Phi should also run well on GPUs – and vice versa – effort is the same.

Conclusions – II

- OpenMP simd constructs can be useful
  - portable code
  - create vectorized functions
- … but compilers need to improve
  - we needed to apply code transformations to improve the performance
  - the compilers for GPUs know, how to apply those transformations

Future work

- re-do the “Tomographic reconstruction” case with OpenMP 4.x constructs
- re-do our baseline studies with OpenMP 4.x constructs
  - will show the quality of the OpenMP implementations
- test different compilers, that support OpenMP 4.x
- can OpenMP 4.x used on GPUs as well?
  - in principle, yes
  - … but what about the quality of the compiler/runtime implementations?

DTU

Thank you for your attention!