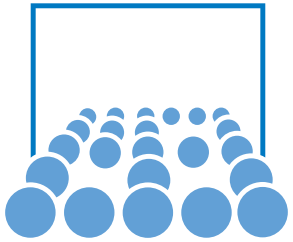
Vectorization

CSCS-FoMICS-USI Summer School on Computer Simulations in Science and Engineering

Michael Bader, Alexander Heinecke

July 8–19, 2013



Part I

Instruction-Level Parallelism

Levels of Parallelism

Classify parallel applications according to granularity of parallelism.
Different kinds of hardware support different levels of parallelism.

Application level:

- Workstation-Cluster (SuperMUC)
- Grid-Computer
- ...

Process level:

- Distributed-Memory-Machine (Cray-T3E, IBM-SP)
- ...

Thread level:

- Shared-Memory-Machine (e.g. SUN-UltraSparc, SGI-Altix)
- ...

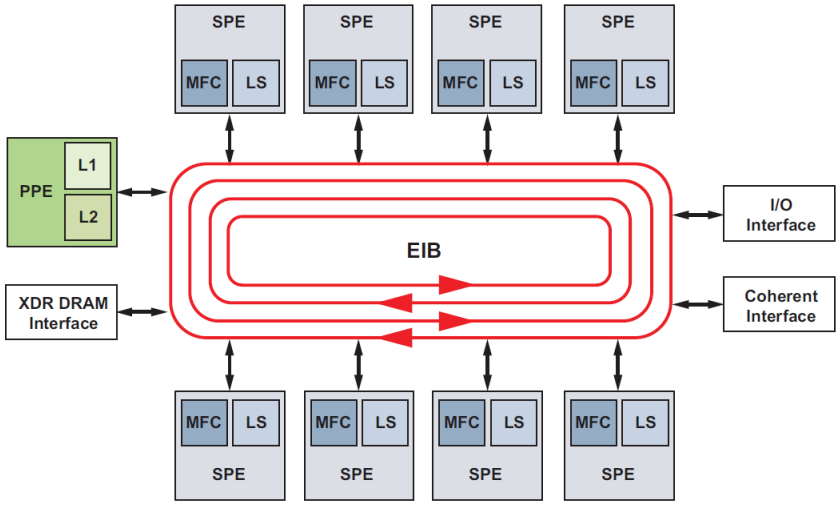
Levels of Parallelism (2)

Instruction level

- super-scalar architecture (e.g. Workstation/PC)
- VLIW architecture (very long instruktion word) (e.g. Itanium, AMD GPUs)
- pipelining
- . . .

Intra-instruction level:

- vector-computer (e.g. NEC-SX, Cray-SV)
- SSE, AVX, Intel MIC
- Cell, GPUs
- . . .



both: ILP – Instruction Level Parallelism

Flynn's Taxonomy

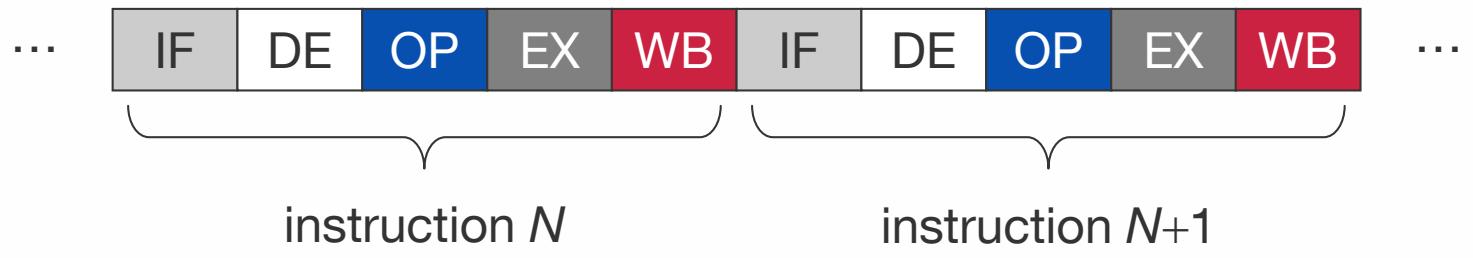
- **instruction flow** and / vs. **data flow**
- handling single/multiple instructions simultaneously
- handling single/multiple data elements simultaneously

SISD scalar machine	**SIMD** vector machine MMX, SSE, AVX, ... GPGPU (VLIW)
MISD	**MIMD** GPGPU (SIMT) MPI (SPMD)

VLIW = "very large instruction word"
SIMT = "single instruction, multiple thread"
SPMD = "single program, multiple data"

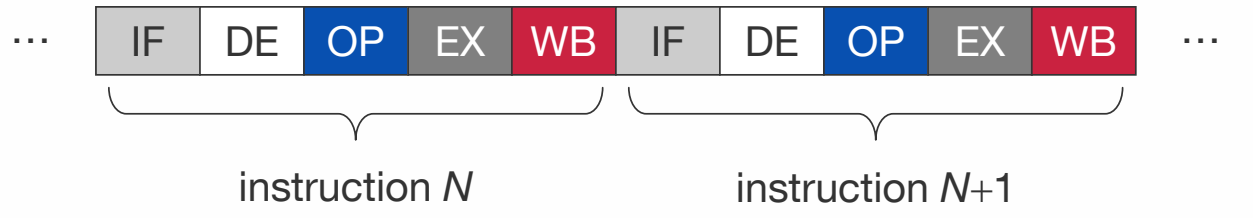
Stages of an Instruction Cycle

Example:

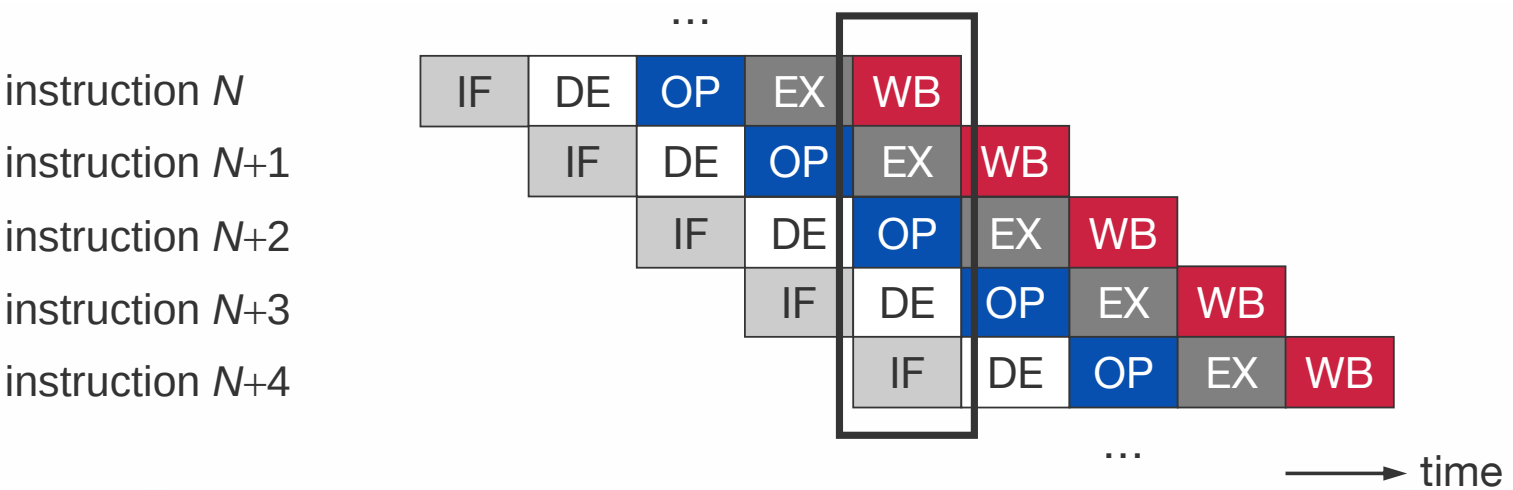


IF: instruction fetch
DEC: instruction decode, register fetch
EXEC: execution, effective address, branch output
MEM: memory access, branch completion
WB: write back

Pipelining



- instruction pipelining: overlapping of independent instructions
- increasing instruction throughout
- a pipeline with *k* stages retires one instruction per cycle!
  (after a certain **preload-time**, starting with cycle *k*)

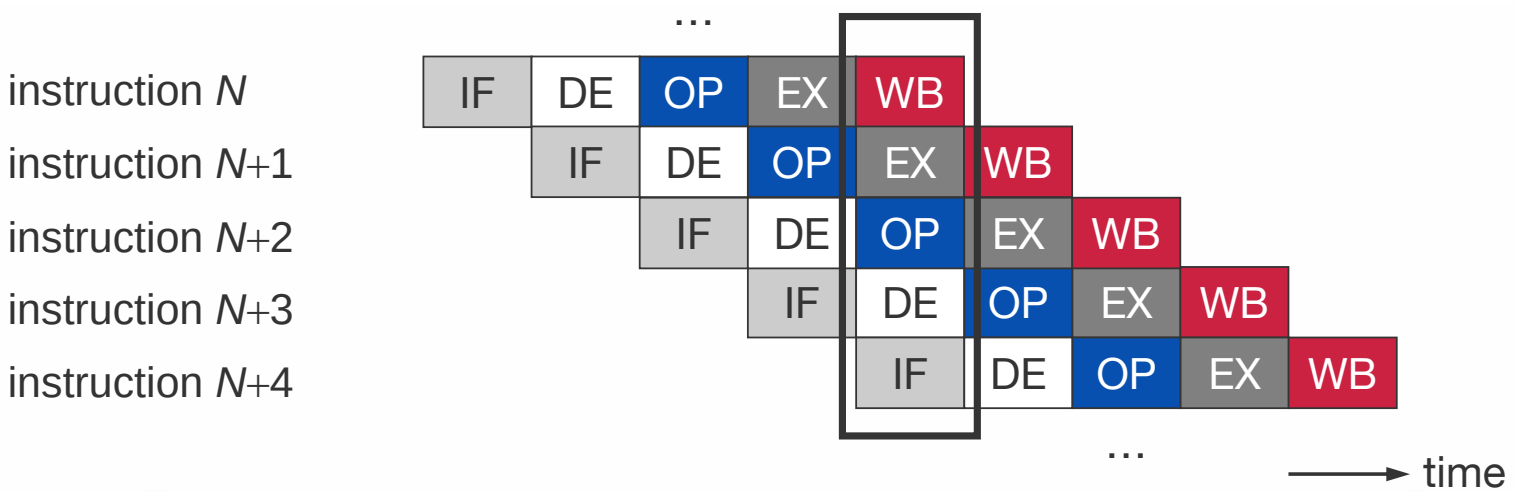


Arithmetic Pipelining

Floating-point addition in four steps

- load operands
- compare exponents, shift mantissa
- add
- normalization and write back to registers

⇒ extends regular pipeline:



Pipelining Optimizations

Loop unrolling:

```
for(i=0;i<3;i++) {
  for(k=0;k<m;k++) {
     c[i,k]=a[i,k]*b[i,k];
  }
}
```

```
for(k=0;k<m;k++) {
    c[0,k]=a[0,k]*b[0,k];
    c[1,k]=a[1,k]*b[1,k];
    c[2,k]=a[2,k]*b[2,k];
}
```

Loop fusion:

```
for(i=0;i<n;i++) {
    d[i]=d[i-1]*e[i];
}

for(i=0;i<n;i++) {
    f[i]=f[i]+c[i-1];
}
```

```
for(i=0;i<n;i++) {
    d[i]=d[i-1]*e[i];
    f[i]=f[i]+c[i-1];
}
```

→ for further details, see **Intel C++ Compiler User Guide**

Pipeline-Conflicts

- **structure-conflict:** parallel execution is prohibited due to hardware limitations (execution units, memory ports)
- **branch-conflict:** jumps
  $\Rightarrow$ branch-prediction, speculative execution
- **data-conflict:** conflicting data-accesses ($c = a + b, d = c + e$)

$\Rightarrow$ compilers try to avoid conflicts as best as possible

$\Rightarrow$ hardware plays several tricks in order to avoid conflicts at runtime!

Dependency Analysis

S1: A = C − A
S2: A = B − C
S3: B = A + C

True dependence (RAW) (e.g. **S3** to **S2** w.r.t. **A**)

- **S3** gives a **true dependence** to **S2**: **S3** δ **S2**
- "read after write": **OUT(S2)** $\cap$ **IN(S3)** $\neq \emptyset$

$\Rightarrow$ **S3** uses result of **S2**

Anti dependence (WAR) (e.g. **S3** to **S2** w.r.t. **B**)

- **S3** gives an **anti dependence** to **S2**: **S3** δ^{-1} **S2**
- "write after read": **IN(S2)** $\cap$ **OUT(S3)** $\neq \emptyset$

$\Rightarrow$ exchanging **S3** and **S2** would use results of **S3** for **S2**

Dependency Analysis (2)

S1: A = C − A
S2: A = B − C
S3: B = A + C

Output dependence (WAW) (e.g. **S2** to **S1** w.r.t. **A**)

- **S2** and **S1** are an **output dependence**: **S2** δ^{o} **S1**
- "write after write": **OUT(S1)** $\cap$ **OUT(S2)** $\neq \emptyset$

$\Rightarrow$ exchanging **S2** and **S1** would result in wrong values

Dependency Analysis checks whether pipelining or vectorization is possible!

Vector Computing

Vector Computer: (70ies–90ies)

- **pipelined** execution units operating on **arrays** of float-point numbers
- several generations of supercomputing architectures by Cray, Fujitsu, Hitachi
- ended with change towards “massively parallel computing”

Vector Registers (SIMD, VLIW, . . .):

- execution of entire **vector-instructions** on **vector-registers**
- started out as “multimedia extensions” (MMX)
- **SSE** (Streaming SIMD Extensions), **AVX**: Advanced Vector Extensions, . . .

SSE-Registers

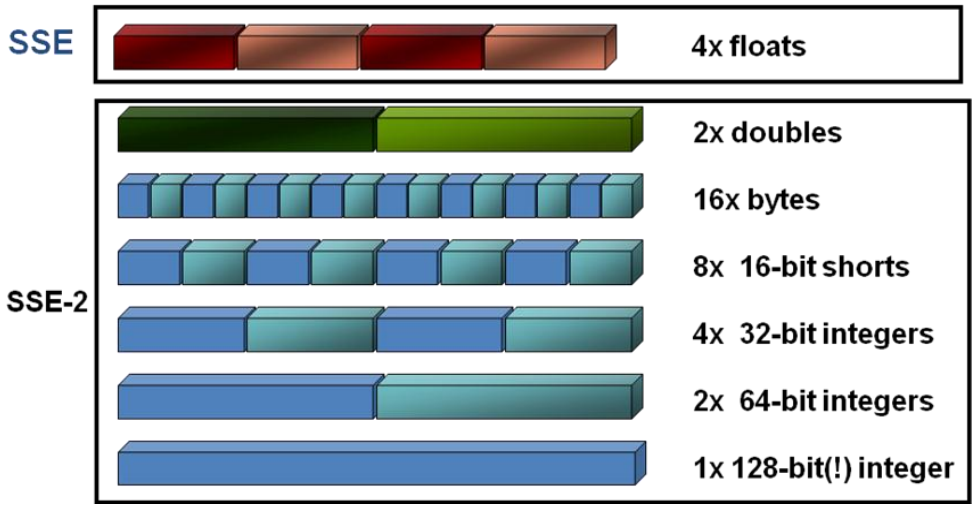


Figure : Streaming SIMD Extensions data types (source: Intel)

Streaming SIMD Extension (SSE)

- 16 registers (xmm0 to xmm15, each 128 bit wide)
- 2 double-precision floating point values (each 64 bit)
- 4 single-precision floating point values (each 32 bit)
- from 2 64-bit integer values up to 16 8-bit integer values

AVX-Register

Advanced Vector Extensions (AVX)

- Since Intel Sandy Bridge or AMD Bulldozer
- 16 Register (ymm0 to ymm15, je 256 bit wide)
- 2 double-precision floating point values (each 64 bit)
- 4 single-precision floating point values (each 32 bit)
- currently no integer support, will be added with AVX2 (Intel Haswell)
- full masking support

Part II

Vectorization

source: *A Guide to Vectorization with Intel® C++ Compilers*

Auto-Vectorization

"auto-vectorization" = unrolling of a loop combined with the generation of packed SIMD instructions by the compiler

Compiler-based vectorization:

- compiler identifies and optimizes suitable loops on its own ...
- ...but can be supported by programmer

Intel compiler:

- attempts vectorization from optimzation level −O2
- reports on vectorized loops: −vec−report
- see: *Intel Compiler Autovectorization Guide*

Vectorization of Loops – Restrictions

Countable:

- loop trip count must be known: at entry to the loop, at runtime
- excludes most while-type loops

Single entry and single exit:

- no data-dependent exit from the loop allowed
- no jumps to labels within the loop

Straight-line code:

- no branches, so switch-statements: identical control flow for all loop iterations necessary
- however: if-statements allowed if they can be implemented as “masked assignments”

Vectorization of Loops – Restrictions (2)

Innermost loop of a nest:

- outer loops are not vectorized
  (unless they can be optimized into innermost loops)

No function calls:

- already a `printf()` can inhbit vectorization
- exception: loops that can be inlined
- exception: vectorized intrinsic math functions
  (e.g.: sin, cos, sqrt, log, fmin, fmax, ...)

Further obstacles:

- data dependencies
- **non-continuous memory access**

Dependency Analysis and Vectorization

- Loop I

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

  no dependency across iterations
  $\Rightarrow$ can be optimally vectorized and pipelined

- Loop II

```
for(i=1;i<n-1;i++) {
      c[i]=c[i]*c[i-1];
}
```

  Recursion! c_{i-1} is still processed when calculating c_i
  $\Rightarrow$ no vectorization is possible

Non-Contiguous Memory Access

- loading strided data elements is inefficient
  → “vectorization possible but seems inefficient”
- non-unit-stride access to an array:

```
for (int i=0; i<SIZE; i+=2) b[i] += a[i] * x[i];
```

- stride access in inner loop:

```
for (int j=0; j<SIZE; j++)
  for (int i=0; i<SIZE; i++) b[i] += a[i][j] * x[j];
```

- indirect addressing:

```
for (int i=0; i<SIZE; i++) b[i] += a[i] * x[index[i]];
```

  → “Existence of vector dependence”

#pragma-Support for Auto-Vectorization

```
#pragma vector always
#pragma ivdep
for(i=1;i<n;i++) {
   c[i]=s*c[i];
}
```

Intel Compiler Pragmas for Intel64 Vectorization:

- **#pragma vector always**
  ⇒ vectorize even if considered inefficient by compiler
- **#pragma ivdep**
  ⇒ vectorize despite of potential vector dependencies
  ⇒ vectorize despite check for exception
- **#pragma simd**
  ⇒ aggressive vectorization, might result in wrong code!

Vectorization and Aliasing

- why is the following for-loop not vectorized?

```
void add(float *a, float *b, float *c) {
    for (int i=0; i<SIZE; i++) c[i] += a[i] + b[i];
}
```

- **aliasing**: potential overlap of arrays a, b, and c in memory!
- option #1: restrict

```
void add(float* restrict a, float* restrict b, float* restrict c) {
    for (int i=0; i<SIZE; i++) c[i] += a[i] + b[i];
}
```

- option #2: **#pragma** ivdep

```
void add(float *a, float *b, float *c) {
    #pragma ivdep
    for (int i=0; i<SIZE; i++) c[i] += a[i] + b[i];
}
```

SSE/AVX Intrinsic Functions

- register variables SSE: __m128i; __m128d;
  register variables AVX: __m256i; __m256d;
- instructions:
  `__m128d _mm_`*instr*`_pd(__m128d a, __m128d b)`
  `__m256d _mm_`*instr*`_pd(__m256d a, __m256d b)`
- with suffixes:
  - **ps**,**pd** → packed single- und double-precision floating point functions
  - **ss**,**sd** → scalar single- und double-precision floating point functions
  - ...
- documentation of all functions in chapter “Intel C++ Intrinsics Reference” of **Intel C++ Compiler User’s Guide**

Example: SSE Intrinsic Functions

```
#include <immintrin.h>

int sse_add(int length, double* a, double* b, double* c) {
    int i;
    __m128d t0,t1;

    for (i = 0; i < length; i +=2) {
        // loading 2 64-bit-double values
        t0 = _mm_load_pd(a[i]);
        t1 = _mm_load_pd(b[i]);
        // adding 2 64-bit-double values
        t0 = _mm_add_pd(t0,t1);
        // storing 2 64-bit-double values
        _mm_store_pd(c[i],t0);
    }
}
```

Example: AVX Intrinsic Functions

```
#include <immintrin.h>

int avx_add(int length, double* a, double* b, double* c) {
    int i;
    __m256d t0,t1;

    for (i = 0; i < length; i +=4) {
        // loading 4 64-bit-double values
        t0 = _mm256_load_pd(a[i]);
        t1 = _mm256_load_pd(b[i]);
        // adding 4 64-bit-double values
        t0 = _mm256_add_pd(t0,t1);
        // storing 4 64-bit-double values
        _m256m_store_pd(c[i],t0);
    }
}
```

Part III

SWE and Vectorization

Array-of-Struct vs. Struct-of-Array

Question: how to choose basic data structure?

- **Array-of-Struct** $\rightarrow$ store *h*, *hu*, *hv*, and *b* for each element:

```
class SWE_Element {
    double h;   // water height
    double hu;  // momentum in x-direction
    double hv;  // momentum in y-direction
    double b;   // bathymetry
}
```

- each SWE_Block then holds an array of SWE_Element
- advantages for software design:
  - \+ Riemann solver takes two variables of SWE_Element
  - \+ would even allow to encapsulate physical model in an SWE_Element (consider derived classes, e.g.)

Array-of-Struct vs. Struct-of-Array (2)

Question: how to choose basic data structure?

- **Struct-of-Array** $\rightarrow$ use separate arrays for *h*, *hu*, *hv*, and *b*:

```
class SWE_Block { /* ... */
    double** h;  // water height
    double** hu; // momentum in x-direction
    double** hv; // momentum in y-direction
    double** b;  // bathymetry
/* ... */ }
```

- advantages for vectorization and optimization:
  - + cache line holds several elements of *h* instead one SWE_Element struct
  - + access to neighbor variables with unit stride
  - + supports vectorization: operations on short vectors of *h*, *hu*, etc.

Vectorization of Riemann Solvers in SWE

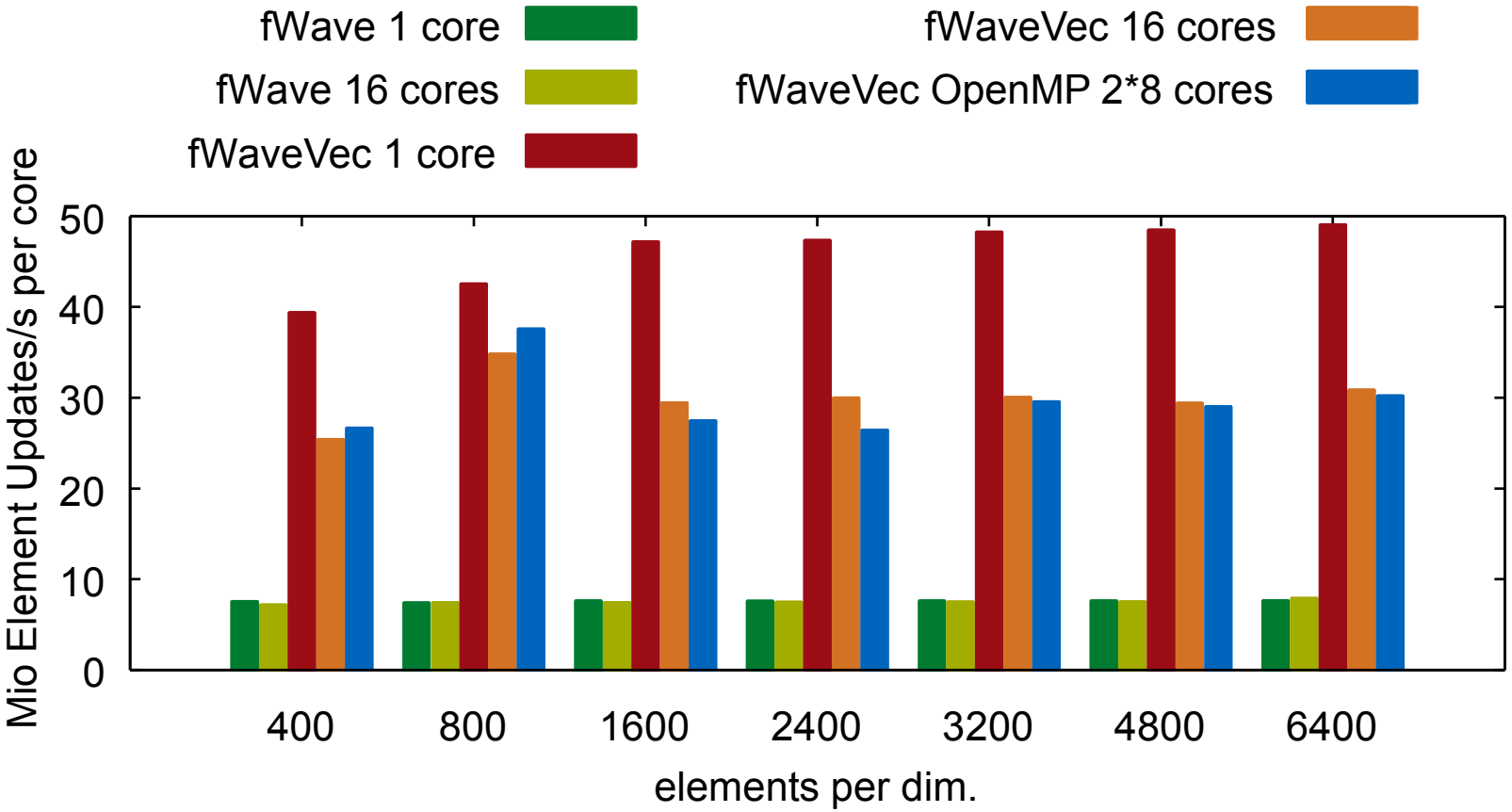
fWave solver:

- vectorization by compiler using **#pragma simd**, etc.
- requires some changes to the implementation (see afternoon workshop)

Augmented Riemann solver:

- complicated Riemann solver, beyond auto-vectorization
- required implementation of vector-valued function using intrinsics
- implemented as part of a Bachelor Thesis (Wolfgang Hölzl)

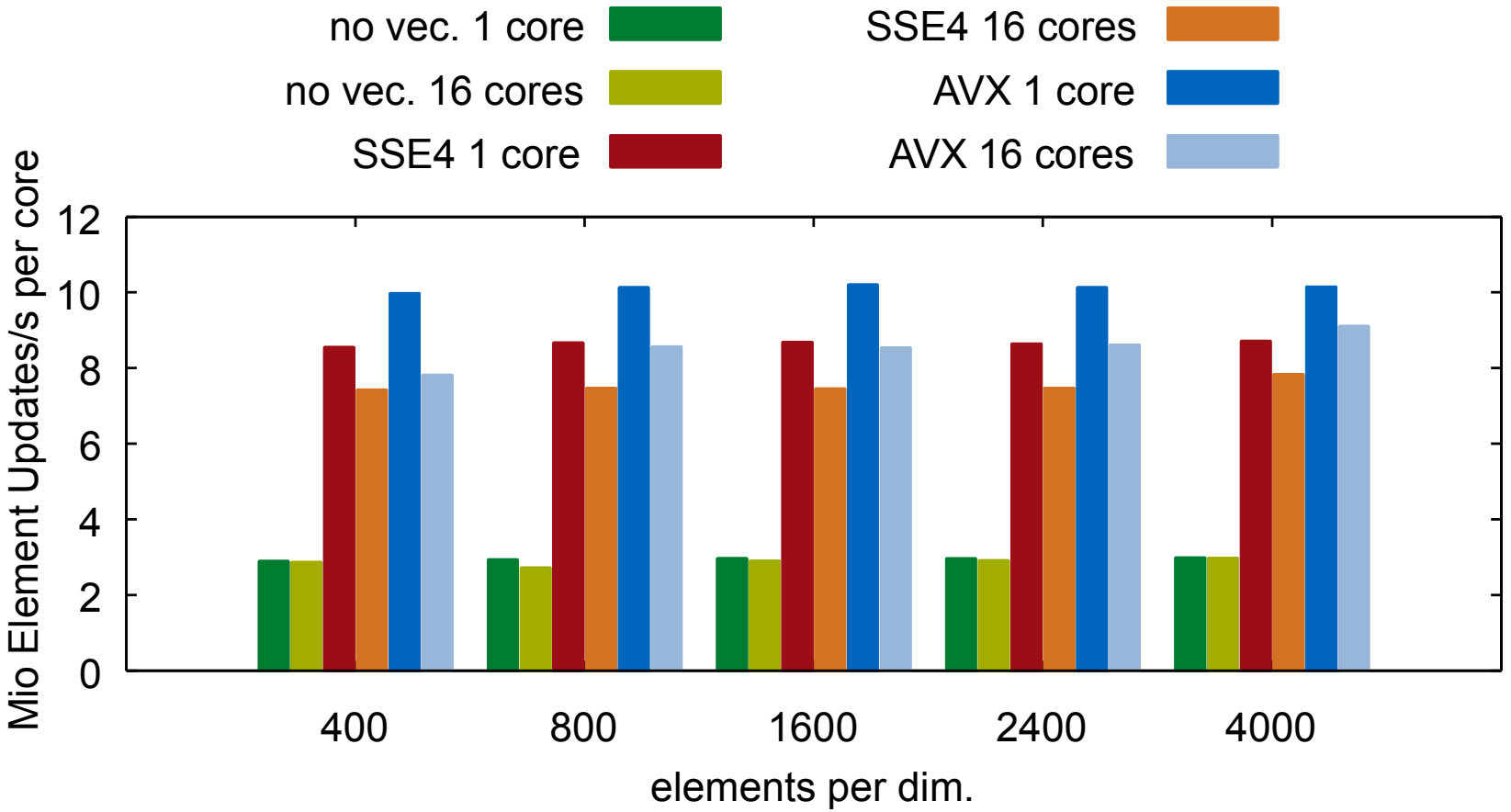
Vectorization of fWave Solver in SWE



Dual-socket Intel Xeon E5-2670 (8 cores, 2.4 GHz)

Vectorization of an Augmented Riemann Solver

Bachelor Thesis Wolfgang Hölzl



Dual-socket Intel Xeon E5-2670 (8 cores, 2.4 GHz)

Part IV

Workshop – SWE and Vectorization

MPI Communication Between Patches

Examine vectorization of SWE:

- 1st step: use option “guided auto-vectorization” of Intel compiler
  $\rightarrow$ check vectorization report for loops
- compare solvers **fWave** vs. **fWaveVec**
- ToDo: support compiler by vectorization pragmas for all performance-critical loops
- test performance with and without vectorization
- test performance implications on serial *and* parallel code!

Possible extensions: (for the ambitious . . .)

- determine GFlop/s for SWE
  (ToDo: count arithmetic operations)
- implement intrinsics-based vectorization in SWE