TECHNISCHE UNIVERSITÄT DRESDEN

Center for Information Services and High Performance Computing (ZIH)

An Introduction to Scout, a Vectorizing Source-to-Source Transformator

ACCU 2012, Oxford, UK

Zellescher Weg 12

Willers-Bau A 105

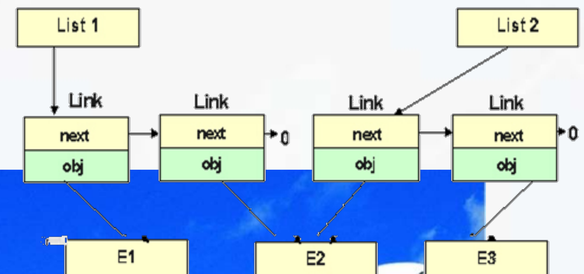
Tel. +49 351 - 463 - 32442

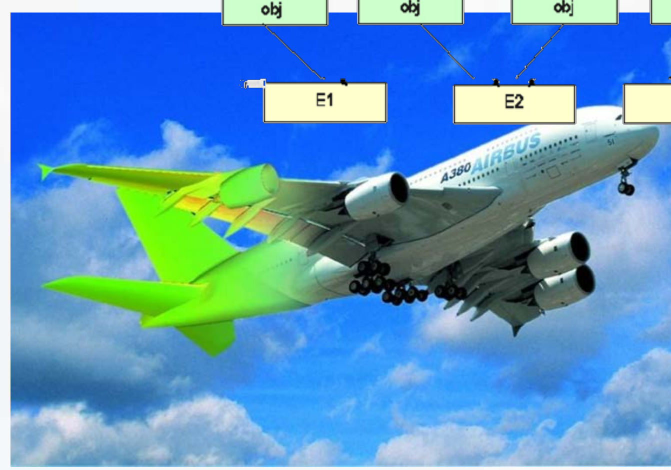
Olaf Krzikalla (olaf.krzikalla@tu-dresden.de)

Center for Information Services & High Performance Computing

Introducing myself

- software engineer
- over 10 years in the "software industry"
- author of the initial version of boost::intrusive
  - Kudos to Ion Gaztañaga!
- back to the university in 2009
  - focus of ZIH on HPC research





Motivation

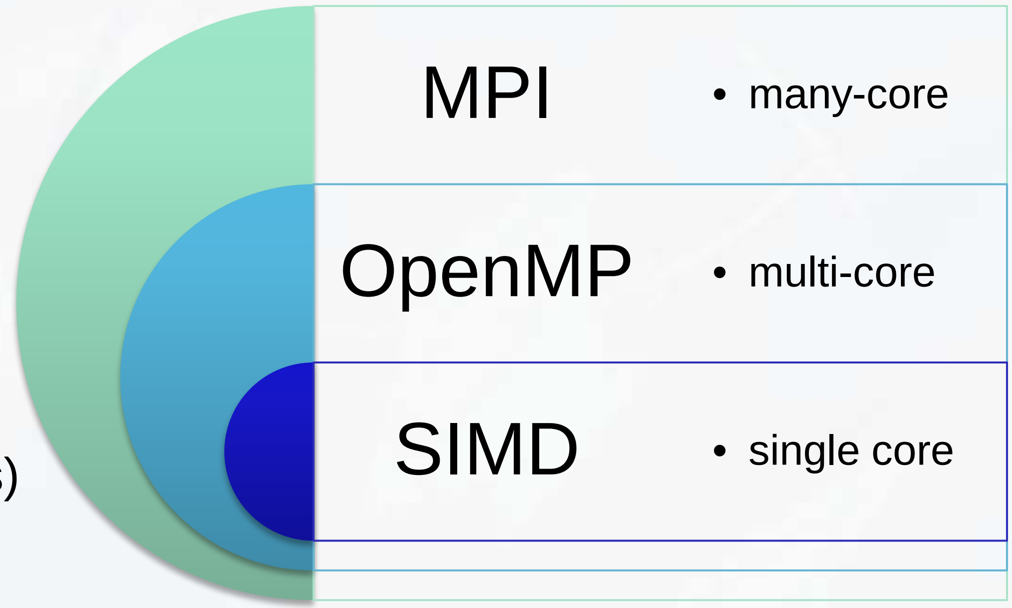
Project HiCFD

Highly efficient Implementation of CFD-Codes for Many-Core-Architectures

Improve the runtime and parallel efficiency of computational fluid dynamics codes on HPC-Many-Core architectures by exploiting all levels of parallelism.

→ "all levels" includes the SIMD features of modern CPUs

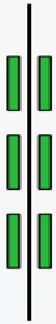
- starting point 2009: **3300 h** for a typical flight maneuver of one minute (40 million points, time resolution 0,01 sec, 10.000 CPUs)



Agenda today

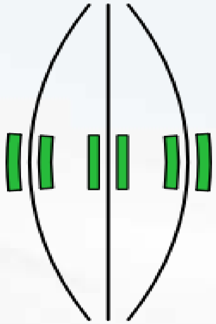
1. Motivation
   - where we come from: the HiCFD project
   - **state of the art of (auto) vectorization**
2. Introducing Scout
   - overview
   - background: using the clang framework for source-to-source transformation
   - features, capabilites, configuration
3. Applying Scout
   - measurements of two real-world CFD codes
4. Advanced vectorization techniques
   - Register blocking
   - Gather and scatter operations
5. Discussion

SIMD Vectorization: The Task



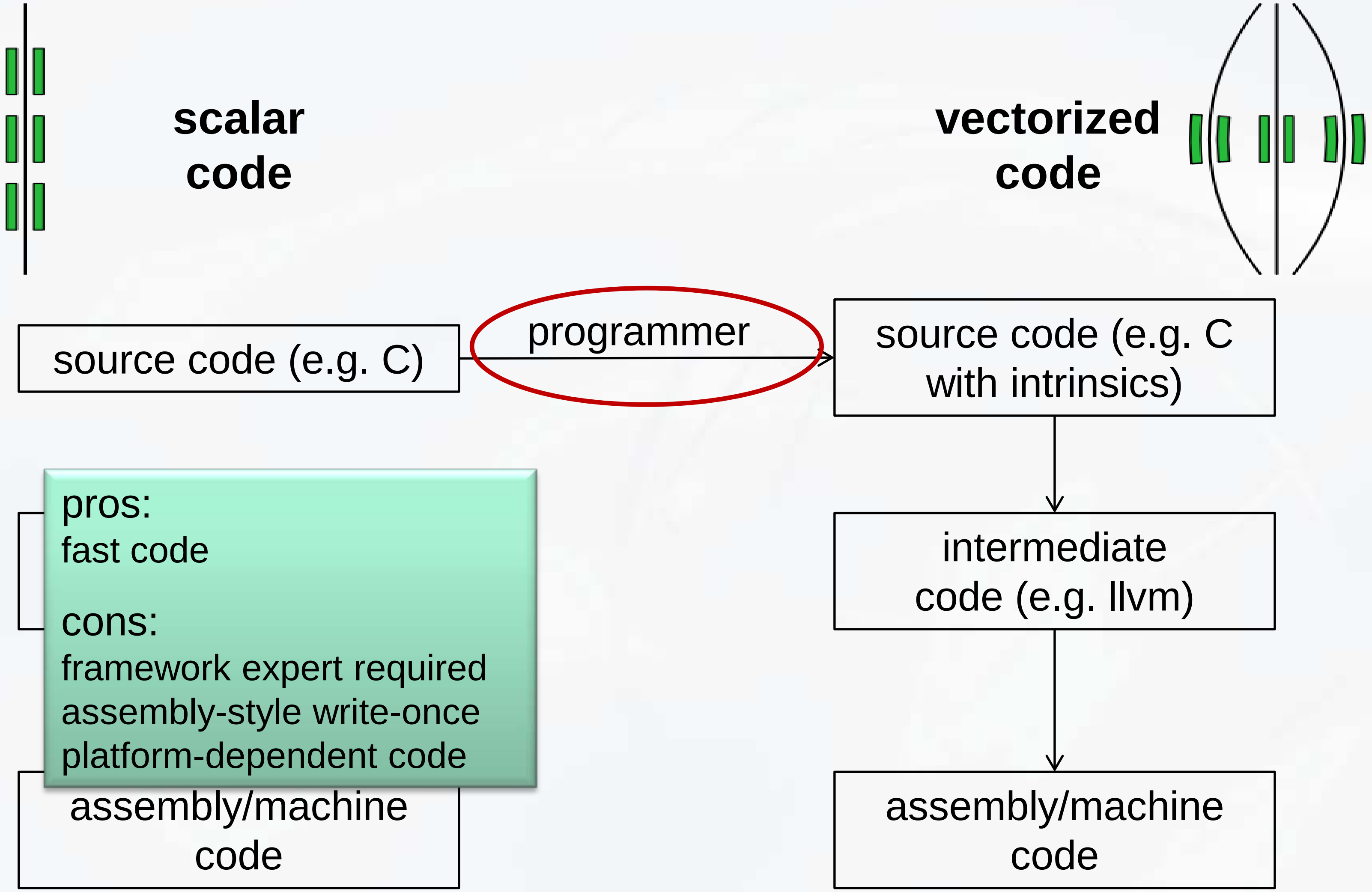
scalar code

vectorized code



scalar	vectorized
source code (e.g. C)	source code (e.g. C)
intermediate code (e.g. llvm)	intermediate code (e.g. llvm)
assembly/machine code	assembly/machine code

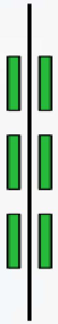
SIMD Vectorization



pros:
fast code

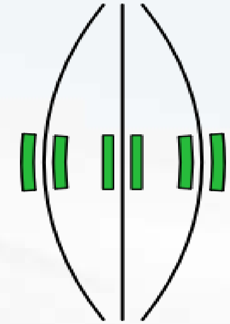
cons:
framework expert required
assembly-style write-once
platform-dependent code

SIMD Vectorization



scalar code

vectorized code



source code (e.g. C) → library → source code using a lib

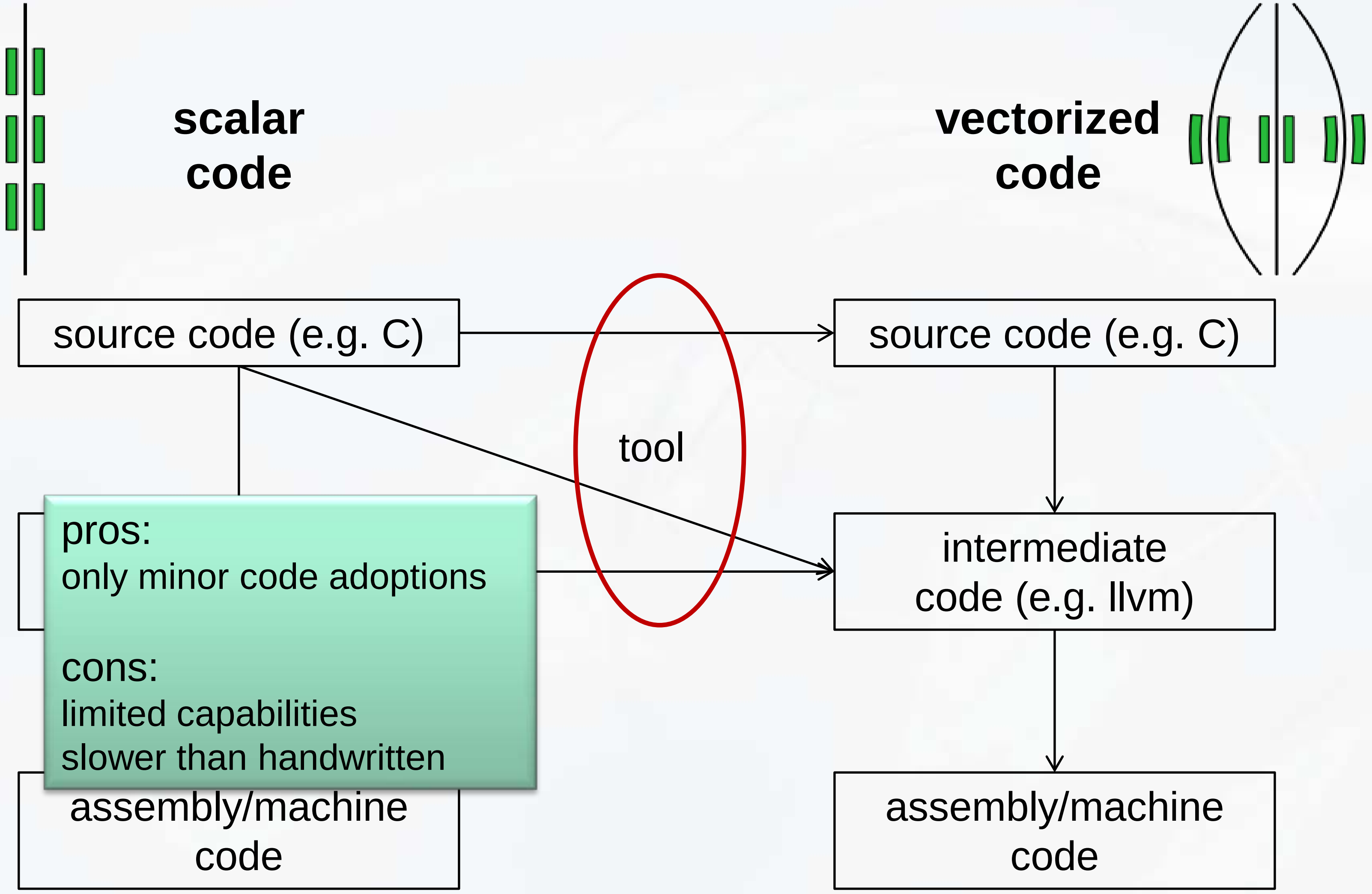
source code using a lib → intermediate code (e.g. llvm) → assembly/machine code

pros:
fast, readable code

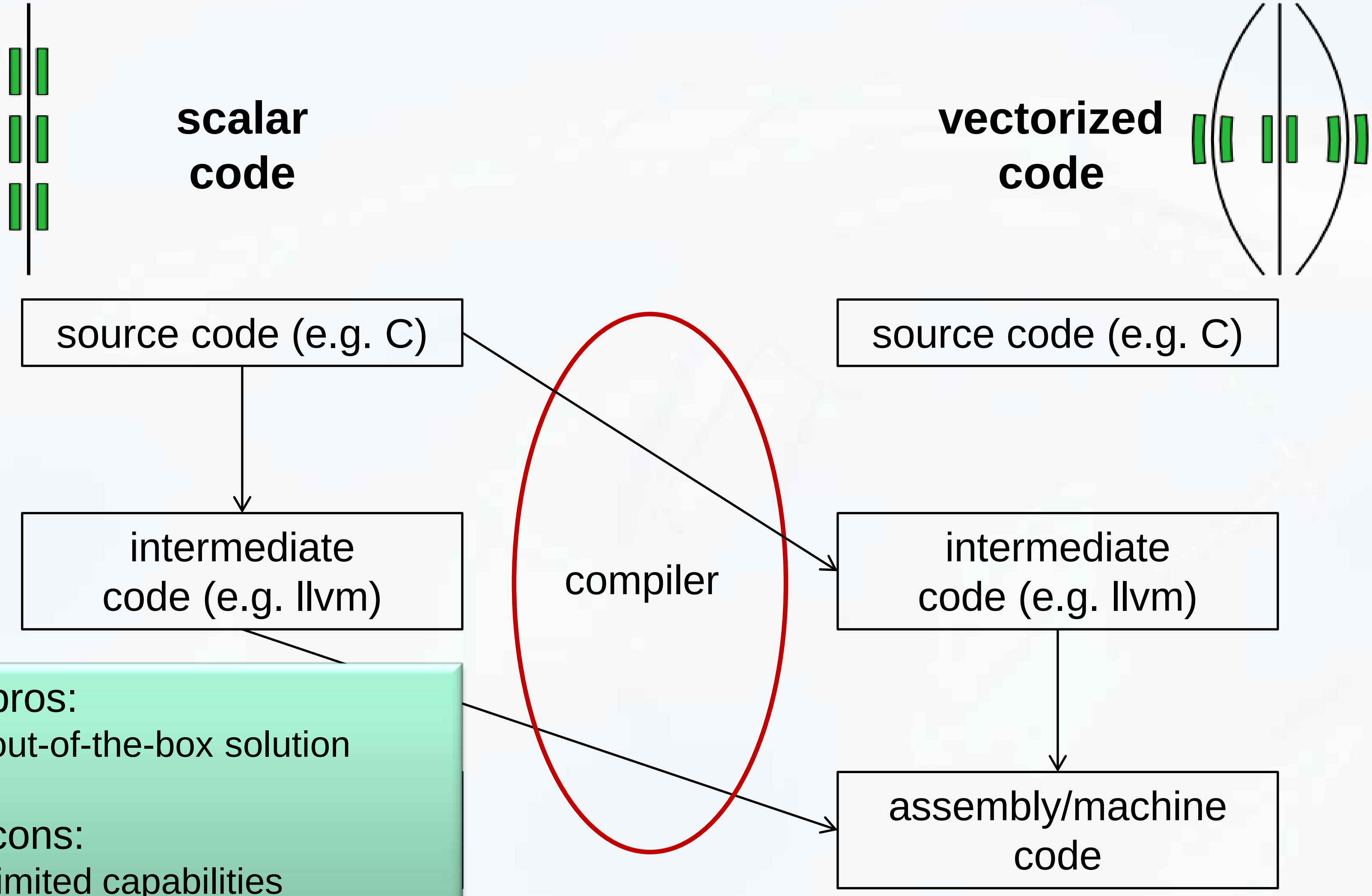
cons:
library-dependent code
adoptions → compromises

assembly/machine code

SIMD Vectorization



SIMD Vectorization



pros:
out-of-the-box solution

cons:
limited capabilities
slower than handwritten

SIMD Vectorization: Compilers

- automatic dependency analysis incapable
  - e.g. no way to reason about indirectly indexed arrays

    → auto-vectorization without meta-information sometimes impossible
  - Maleki, S. et. al.: An Evaluation of Vectorizing Compilers, PACT 2011
- pragma-based vectorization reveals new pecularities
  - e.g. requirement of an **unsigned** loop index
  - vectorization of real world application loops ranged from hard to impossible
- considerable improvements by icc V12 (**`#pragma simd`**)

SIMD Vectorization: Tools

- some tools are lost in a former millennium
  - techniques still remain
- academic tools:
  - Hohenauer et.al.: A SIMD optimization framework for retargetable compilers
  - Pokam et.al.: SWARP: a retargetable preprocessor for multimedia instructions
- commercial tools:
  - HMPP by CAPS Enterprise (source-to-source compiler)

List certainly incomplete...

Agenda today

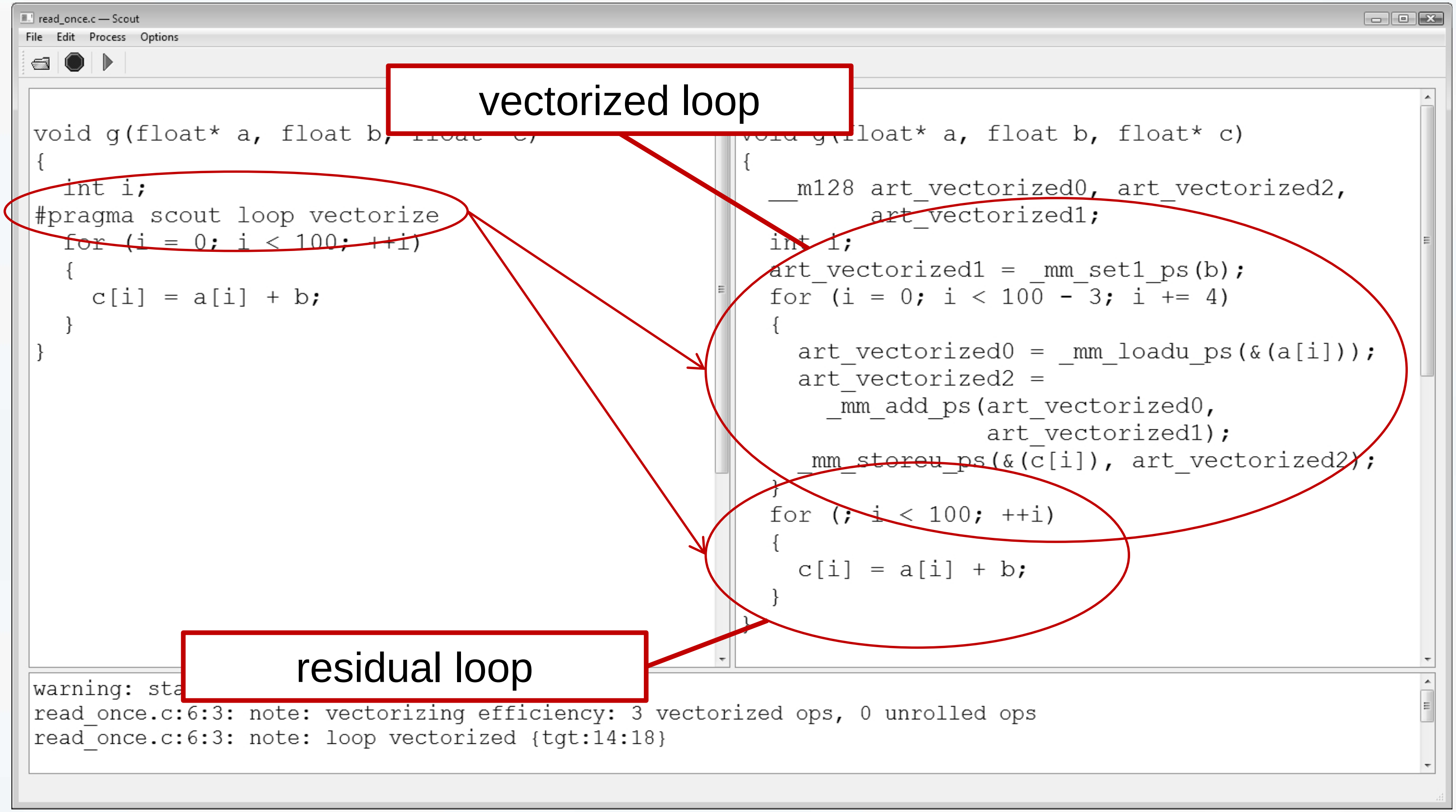
1. Motivation
   - where we come from: the HiCFD project
   - state of the art of (auto) vectorization
2. **Introducing Scout**
   - **overview**
   - **background: using the clang framework for source-to-source transformation**
   - **features, capabilites, configuration**
3. Applying Scout
   - measurements of two real-world CFD codes
4. Advanced vectorization techniques
   - Register blocking
   - Gather and scatter operations
5. Discussion

Scout: Introduction

Scout: A Source-to-Source Transformator for SIMD-Optimizations

- C as input and output source code
- focus on pragma-annotated loop vectorization
  - loop body simplification by function inlining and loop unswitching
  - unroll or vectorize inner loops
- User-Interface: GUI and command-line version
- Open source software tool: **`http://scout.zih.tu-dresden.de/`**
  - used in the production to accelerate code without much effort
  - a means to investigate new vectorization techniques

Scout: Impression



Scout: Command Line

- CLI for an automatic build process via Makefile:

```
> scout -scout:configuration=./config/avx2.cpp -scout:extension=sc
-scout:prolog=./config/prolog.inc –I/usr/include file.c
```

- compiler arguments passed to clang
  - clang uses mostly gcc-like syntax
- Scout-specific arguments start with **`-scout:`**

`-scout:configuration=file` `[req]`: configuration file

`-scout:preprocess=file` `[opt]`: source file(s) containing definitions of inlined functions

`-scout:extension=text` `[opt]`: target file extension

`-scout:prolog=file` `[opt]`: file content inserted in the target file

Scout: Command Line

Effect of `-scout:prolog=./prolog.inc -scout:extension=sse`:

prolog.inc:

```
/* prolog start */
#include "ia32intrin.h"
/* prolog end */
```

source.c:

```
#include "user.h"

void foo(float* a, float b)
{
#pragma scout loop vectorize
  for (i = 0; i < 100; ++i) {
    a[i] += b;
  }
}
```

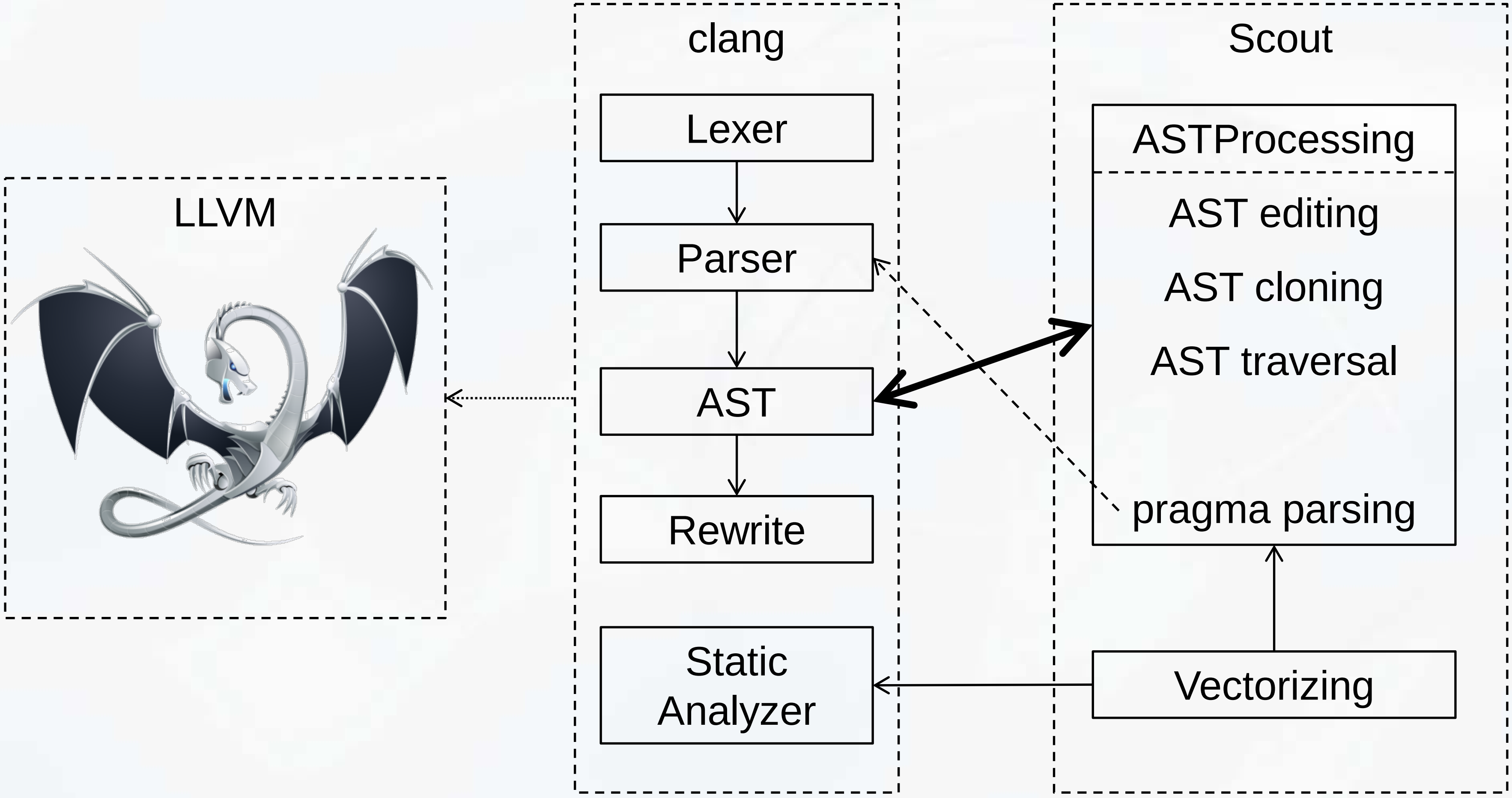
source.c.sse:

```
#include "user.h"
/* prolog start */
#include "ia32intrin.h"
/* prolog end */

void foo(float* a, float b)
{
 __m128 av, bv, tv;
 bv = _mm_set1_ps(b);
 for (i = 0; i < 100; i += 4)
 {
   tv = _mm_loadu_ps(a + i);
   av = _mm_add_ps(tv, bv);
   _mm_storeu_ps(a + i, av);
 }
}
```

- content of prolog file inserted before the first meaningful non-include line

→ target file directly compilable

Background: Scout and clang



Background: Scout and clang

- manipulation of clang's abstrac syntax tree:
  - actually immutable
  - actually more than an AST
  - → actually a tricky and sometimes hacky approach
- nevertheless scout::ASTProcessing works:

```
// x += y → x = x + y for arbitrary ops:
typedef CompoundAssignOperator CAO;
for (stmt_iterator<CAO> i = stmt_ibegin<CAO>(Root),
                        e = stmt_iend<CAO>(Root); i != e; ++i) {
  CAO* Node = *i;
  BinaryOperator::Opcode binOpc = transformOpc(Node->getOpc());
  Expr* clonedLhs = Clone_(Node->getLHS());
  clonedLhs->setValueKind(VK_RValue);       // the tricky part
  replaceStatement(
    Node,
    Assign_(Node->getLHS(),
            BinaryOp_(clonedLhs, Node->getRHS(), binOpc)));
}
```

Background: Scout and clang

- parsing expressions in pragma arguments
  - forecast: `#pragma scout loop vectorize aligned(a.array, a.ptr)`
  - not supported by clang → brute force patch
  - but they are going to need something similar for OpenMP
  - better solution: configurable C++11 attributes

- clang::StaticAnalyzer for alias analysis

```
struct ValueFlowContext {
  const MemRegion* getLValueMemRegion(Expr*);
  SVal getRValueSVal(Expr*);
  void bindValue(const MemRegion*, SVal);
  //uses clang::ProgramState module
};

ValueFlowContext VFCtx;
//...
assert(VFCtx.getLValueMemRegion(Node1) ==
       VFCtx.getLValueMemRegion(Node2));
```

```
double* b = //...
for (int i=0; ...) {
  double* ptr = b;
  // b[i] → Node1
  // ptr[i] → Node2
}
```

Agenda today

1. Motivation
   - where we come from: the HiCFD project
   - state of the art of (auto) vectorization
2. Introducing Scout
   - overview
   - background: using the clang framework for source-to-source transformation
   - **features, capabilites, configuration**
3. Applying Scout
   - measurements of two real-world CFD codes
4. Advanced vectorization techniques
   - Register blocking
   - Gather and scatter operations
5. Discussion

Scout: Configuration

- configuration file written in C++:
  - no need to learn another configuration syntax
  - usual preprocessing means (conditional compilation, includes) available
  - somewhat stretched semantics
- replace expressions by their vectorized intrinsic counterparts
  - even complex expressions like a+b*c (fmadd) or a<b?a:b (min)
  - and function calls like sqrt
- currently available configurations:
  - SSE2, SSE4 (blending): double, float
  - AVX, AVX2 (fmadd, gather): double, float
  - ARM NEON: float
- other architectures easy to provide
  - MIC (sorry, under NDA)
  - even upcoming or experimental ones

Scout: Configuration

- specialized template named **`config`** in the namespace **`scout`**:
  - type parameter: base type of the vector instruction set
  - integral parameter: vector size (number of vector lanes)
- name of target SIMD type introduced by a typedef named **`type`**
  - probably that name needs to be introduced
  - → no need to include the header of the appropriate SIMD architecture

```
namespace scout {

template<class T, unsigned size> struct config;

template<> struct config<float, 4> {

  typedef float __m128 __attribute__ ((__vector_size__ (16)));
  typedef __m128 type;
  enum { align = 16 };

};

}  // end of namespace
```

Scout: Configuration

- specialized template named **`config`** in the namespace **`scout`**:
  - type parameter: base type of the vector instruction set
  - integral parameter: vector size (number of vector lanes)
- name of target SIMD type introduced by a typedef named **`type`**
  - probably that name needs to be introduced
  - → no need to include the header of the appropriate SIMD architecture
- alignment requirement by a enum named **`align`**

```
namespace scout {

template<class T, unsigned size> struct config;

template<> struct config<float, 4> {

  typedef float __m128 __attribute__ ((__vector_size__ (16)));
  typedef __m128 type;
  enum { align = 16 };

};

}  // end of namespace
```

Scout: Configuration

- instruction definition by static member functions
  - last statement of a member function body always a string literal
  - that string is inserted in the target code
  - arguments are expanded using boost::format

```
template<> struct config<float, 4> {

  typedef float base_type;
  typedef float __m128 __attribute__ ((__vector_size__ (16)));
  typedef __m128 type;

  static type load_aligned(base_type*)
  {
    "_mm_load_ps(%1%)";
  }

  static void store_aligned(base_type*, type)
  {
    "_mm_store_ps(%1%, %2%)";
  }
};
```

Scout: Configuration

- picking the instruction by pre-defined method name:

`type load_[un]aligned(base_type* p)`	`return *p as vector`
`void store_[un]aligned(base_type* p, type v)`	`*p := v`
`void store_nt_packed_[un]aligned (base_type* p, type v)`	`*p := v (no cache pollution)`
`type splat(base_type x)`	`return { x, x,..., x }`
`type broadcast(base_type* p)`	`return {*p, *p,..., *p }`
`type set(base_type x1, ..., base_type xn)`	`return { xn, ... x1 }`
`base_type extract(type v, unsigned int i)`	`return v[i]`
`void insert(type v, base_type x, unsigned int i)`	`v[i] := x`

- additional names for gather/scatter support:

`gs_index_type get_uniform_gs_index(int)`
`type gather(base_type*, gs_index_type)`
`void scatter(base_type*, gs_index_type, type)`

Scout: Configuration

- picking the instruction by expressions or functions:
  - method name ignored
  - method signature uses fundamental types
  - string literal preceded by a list of expressions and/or function declarations

```
template<> struct config<float, 4> {
  ...
#ifdef SCOUT_CONFIG_WITH_SSE4
  static base_type blend_ge(base_type a, base_type b,
                            base_type c, base_type d)
  {
    a < b ? c : d;
    "_mm_blendv_ps(%4%, %3%, _mm_cmpge_ps(%1%, %2%))";
  }
#endif
};
```

Scout: Configuration

Example: complex expression mapping with SSE4

```
float a[100], b[100], x;

#pragma scout loop vectorize
for (i = 0; i < 100; ++i) {
  b[i] = a[i] >= 0.0 ?
           sqrt(a[i]) :
           x;
}
```

→

```
float a[100], b[100], x;
__m128 av, bv, cv, xv, tv;

cv = _mm_set1_ps(0.0);
xv = _mm_set1_ps(x);

for (i = 0; i < 100; i += 4) {
  av = _mm_loadu_ps(a + i);
  tv = _mm_sqrt_ps(av);
  bv = _mm_blendv_ps(xv, tv,
        _mm_cmpge_ps(av, cv));
  _mm_storeu_ps(b + i, bv);
}
```

- computation of all lanes
  → turn off fp exceptions (if not already done)
- blending → good acceleration

Scout: Technology

- simplify loop bodies
  - source-level inlining
  - loop unswitching (remove loop-invariant conditions)
- vectorize assignments
  - don't change the arrays-of-structs data layout
    → composite load / store operations required
  - complex loops with conditions and mixed data types
- modern SMD architectures provide rather short vector registers
  - whole-loop transformations better suited for traditional vector machines

→ vectorization by *unroll-and-jam* approach

Scout: Technology

Unroll-and-Jam:

```
#pragma scout loop vectorize
for (i = si; i < ei; ++i)
{
    s_i;
    t_i;
}
```

```
for (i=si; i<ei-vs+1; i+=vs)
{
   s_i;
   s_{i+1};
   ...
   s_{i+vs};
   t_i;
   t_{i+1};
   ...
   t_{i+vs};
}
// residual loop
```

- first step: unroll all statements according to the vector size

Scout: Technology

Unroll-and-Jam:

```
#pragma scout loop vectorize
for (i = si; i < ei; ++i)
{
    s_i;
    t_i;
}
```

```
for (i=si; i<ei-vs+1; i+=vs)
{
  s_i:i+vs;
  t_i;
  t_i+1;
  ...
  t_i+vs;
}
// residual loop
```

- first step: unroll all statements according to the vector size
- second step: jam vectorizeable statements
- unvectorizeable statements remain unrolled
  - but they don't inhibit vectorization

Scout: Technology

unrolling of non-inlined functions:

```
double a[100], d[100];
#pragma scout loop vectorize
for (i = 0; i < 100; ++i) {
  a[i] = foo(2.0 * d[i]);
}
```

```
__m128d cv, av1, av2, av3;
double a[100], d[100];
cv = _mm_set1_pd(2.0);
for (i = 0; i < 100; i += 2) {
 av1 = _mm_loadu_pd(d + i);
 av2 = _mm_mul_pd(cv, av1);
 av3 = _mm_set_pd(
  foo(_mm_extract_pd(av2,1)),
  foo(_mm_extract_pd(av2,0)));
 _mm_storeu_pd(a + i, av3);
}
```

- no support for functions with out-arguments

Scout: Technology

unrolling of if-statements:

```
double a[100], b[100], d[100];
#pragma scout loop vectorize
for (i = 0; i < 100; ++i)
{
  y = a[i];
  if (y < 0)
    b[i] = d[i] + y;
  else
    b[i] = d[i] - y;
  a[i] = b[i] * d[i];
}
```

→

```
__m128d yv, bv, tv, dv;
double a[100], b[100], d[100];
for (i = 0; i < 100; i += 2) {
  yv = _mm_loadu_pd(a + i);
  if (_mm_extract_pd(yv,0)<0)
    b[i] = d[i] +
      _mm_extract_pd(yv, 0);
  else
    b[i] = d[i] -
      _mm_extract_pd(yv, 0);
  if (_mm_extract_pd(yv,1)<0)
    b[(i + 1)] = d[(i + 1)] +
      _mm_extract_pd(yv, 1);
  else
    b[(i + 1)] = d[(i + 1)] -
      _mm_extract_pd(yv, 1);
  bv = _mm_loadu_pd(b + i);
  dv = _mm_loadu_pd(d + i);
  tv = _mm_mul_pd(bv, dv);
  _mm_storeu_pd(a + i, tv);
}
```

- only necessary for loop-variant conditions

Scout: Scalar Transformations

Loop unswitching:

```
double a[100], b, c[100];
int mode = /*...*/

#pragma scout loop vectorize scalar
for (i = 0; i < 100; ++i)
{
  if (mode == 0)
    c[i] = a[i] + b;
  else
    c[i] = a[i] - b;
}
```

```
double a[100], b, c[100];
int mode = /*...*/

if (mode == 0) {

  for (i = 0; i < 100; ++i)
  {
    c[i] = a[i] + b;
  }

} else {

  for (i = 0; i < 100; ++i)
  {
    c[i] = a[i] - b;
  }

}
```

- moving of loop-invariant conditions outside of loops
- code bloat outweighted by vectorization gains

Scout: Scalar Transformations

Loop unswitching:

```
double a[100], b, c[100];
int mode = /*...*/

#pragma scout loop vectorize scalar
for (i = 0; i < 100; ++i)
{
  c[i] = mode == 0 ? a[i] + b
                   : a[i] – b;
}
```

```
double a[100], b, c[100];
int mode = /*...*/

if (mode == 0) {

  for (i = 0; i < 100; ++i)
  {
    c[i] = a[i] + b;
  }

} else {

  for (i = 0; i < 100; ++i)
  {
    c[i] = a[i] – b;
  }

}
```

- moving of loop-invariant conditions outside of loops
- code bloat outweighted by vectorization gains
- also used for conditional expressions

Scout: Capabilities

Mixed data types:

```
float a[100];
double b[100], x;
#pragma scout loop vectorize
for (i = 0; i < 100; ++i) {
  x = a[i];
  x = x / b[i];
  a[i] = x;
}
```

```
float a[100];
double b[100];
__m128 av;
__m128d xv1, xv2, bv1, bv2;

for (i = 0; i < 100; i += 4) {
  av = _mm_loadu_ps (a + i);
  xv1 = _mm_cvtps_pd (av);
  xv2 = _mm_cvtps_pd (
    _mm_movehl_ps (av, av));
  bv1 = _mm_loadu_pd (b+i);
  bv2 = _mm_loadu_pd (b+i+2);
  xv1 = _mm_div_pd (xv1, dv1);
  xv2 = _mm_div_pd (xv2, dv2);
  av = _mm_movelh_ps (
    _mm_cvtpd_ps (xv1),
    _mm_cvtpd_ps (xv2));
  _mm_storeu_ps (a + i, av);
}
```

- vectorized according to the largest vector size
- by-product of the rather local scope of the *unroll-and-jam* approach

Scout: Capabilities

Arbitrary constant loop stride:

```
int k = /*...*/;
double a[100], b[100];
#pragma scout loop vectorize
for (i = 0; i < 100; i += k) {
  a[i] = b[i] * b[i];
}
```

→

```
int k = /*...*/;
__m128d av1, av2;
double a[100], b[100];
for (i = 0;
     i < 100 - 1;
     i += k * 2) {
  av1 = _mm_set_pd(b[(i + k)],
                   b[i]);
  av2 = _mm_mul_pd(av1, av1);
  a[i] =
    _mm_extract_pd(av2, 0);
  a[(i + k)] =
    _mm_extract_pd(av2, 1);
}
/* residual loop */
```

- array-of-structure data layout need composite load and store operations
  → this nets that feature anyway
- coming later (but today): gather and scatter

Scout: Capabilities

Partial vectorization:

```
double a [100], c[100];
int d [100];
#pragma scout loop vectorize
for (i = 0; i < 100; ++i) {
  int j = d[i];
  double b = a[j];

  // computations

  // introduces an inner-loop
  // dependency if
  // d[i]==d[i+1]:

#pragma scout vectorize unroll
  c[j] += b;
}
```

```
__m128d b_v;
int j_v [2];
double a [100], c[100];
int d [100];
for (i =0; i <100; i +=2) {
  j_v [0] = d[i];
  j_v [1] = d[(i + 1)];
  b_v = _mm_set_pd (
    a[j_v[0]], a[j_v[1]]);

  // vectorized computations

  // compute every element
  // separately:

  c[j_v[0]] += _mm_extract_pd
                  (b_v ,0);
  c[j_v[1]] += _mm_extract_pd
                  (b_v ,1);
}
```

- enables the vectorization of loops in the presence of dependencies
- transform the vectorizeable part and leave dependencies intact

Scout: Capabilities

Inner loop vectorization:

```
double *a, *b, c, *d;
#pragma scout loop vectorize
for (i = 0; i < 100; ++i)
{
  d[i] = a[i];
  for (j = 0; j < k; ++j)
  {
    d[i] += b[j * k + i];
  }
  d[i] *= c;
}
```

→

```
double *a, *b, c, *d;
__m128d dv, cv, bv;
for (i = 0; i < 100; i += 2)
{
  dv = _mm_loadu_pd(a + i);
  for (j = 0; j < k; ++j)
  {
    bv = _mm_loadu_pd(b+j*k+i);
    dv = _mm_add_pd(dv, bv);
  }
  dv = _mm_mul_pd(dv, cv);
  _mm_storeu_pd(d + i, dv);
}
```

- only for constant inner loop range

Scout: Capabilities

Inner loop vectorization:

```
double *a, *b, c, *d;
#pragma scout loop vectorize
for (i = 0; i < 100; ++i)
{
  d[i] = a[i];
  for (j = 0; j < k; ++j)
  {
    d[i] += b[i * k + j];
  }
  d[i] *= c;
}
```

→

```
double *a, *b, c, *d;
__m128d dv, cv, bv;
for (i = 0; i < 100; i += 2)
{
  dv = _mm_loadu_pd(a + i);
  for (j = 0; j < k; ++j)
  {
    bv = _mm_set_pd(
      b[(i * k + j + k)],
      b[i * k + j]);
    dv = _mm_add_pd(dv, bv);
  }
  dv = _mm_mul_pd(dv, cv);
  _mm_storeu_pd(d + i, dv);
}
```

- only for constant inner loop range

Scout: Capabilities

Inner loop vectorization:

```
double *a, *b, c, *d;
#pragma scout loop vectorize
for (i = 0; i < 100; ++i)
{
  d[i] = a[i];
  for (j = 0; j < k; ++j)
  {
    d[i] += b[i];
  }
  d[i] *= c;
}
```

```
double *a, *b, c, *d;
__m128d dv, cv, bv;
for (i = 0; i < 100; i += 2)
{
  dv = _mm_loadu_pd(a + i);
  for (j = 0; j < k; ++j)
  {
    bv = _mm_loadu_pd(b + i);
    dv = _mm_add_pd(dv, bv);
  }
  dv = _mm_mul_pd(dv, cv);
  _mm_storeu_pd(d + i, dv);
}
```

- only for constant inner loop range
- no displacement of inner-loop-invariant expressions by Scout

Scout: Capabilities

Reductions:

```
float *a, x, y;

#pragma scout loop vectorize
for (i = 0; i < 100; ++i)
{
  x += a[i];
  y = MIN(y, a[i]);
}
```

```
float *a, x, y;
__m128 av, xv, yv;
xv = _mm_set1_ps(0);
yv = _mm_set1_ps(y);

for (i = 0; i < 100; i += 2) {
  av = _mm_loadu_ps(a + i);
  xv = _mm_add_ps(xv, av);
  yv = _mm_min_ps(yv, av);
}

for (ti = 0U; ti < 4U; ++ti) {
  x = x + _mm_extract_ps(xv, ti);

}
for (ti = 0U; ti < 4U; ++ti) {
  y = y<_mm_extract_ps(yv, ti) ?
                              y :
         _mm_extract_ps(yv, ti);
}
```

- note the numerical instability due to a different computation order
- TODO: merging the loops and introduce horizontal operations

Scout: Fine-Tuning the Vectorization

- list of additional **`#pragma scout vectorize`** arguments:

`noremainder`	don't generate a residual loop
`size(N)`	unroll N times (N >= VS && N % VS==0)
`scalar`	no vectorization (only inlining and unswitching)
`aligned(expr)`	use aligned loads/stores
`nontemporal(expr)`	use nontemporal loads/stores (avoid cache pollution)
`align`	automatic alignment of the most often accessed memory location

- additional pragmas:

`#pragma scout loop vectorize unroll`	used in loop bodies: next statement remains unrolled
`#pragma scout function expand`	used in front of a function definition: all calls in that function are recursively inlined
`#pragma scout loop unroll`	used in front of a loop with constant iteration range: loop gets completely unrolled

Scout: Capabilities

Using **`aligned`** and **`nontemporal`**:

```
typedef struct {
  float* ptr;
  float array[100];
} A;

A a;
float b[100];
#pragma ... aligned(a, b)
for (i = 0; i < 100; ++i)
{
  a.array[i] = b[i];
  a.ptr[i] = b[i+1];
}
```

```
typedef struct {
  float* ptr;
  float array[100];
} A;

A a;
float b[100];
__m128 tv
for (i = 0; i < 100; i += 4)
{
  tv = _mm_load_ps(b + i);
  _mm_store_ps(a.array + i, tv);
  tv = _mm_load_ps(b + i+1);
  _mm_storeu_ps(a.ptr + i, tv);
}
```

- accesses to regions and their direct subregions are aligned

Scout: Capabilities

Using **`aligned`** and **`nontemporal`**:

```
typedef struct {
  float* ptr;
  float array[100];
} A;

A a;
float b[100];
#pragma ... aligned(a, b)
for (i = 0; i < 100; ++i)
{
  a.array[i] = b[i];
  a.ptr[i] = b[i+1];
}
```

→

```
typedef struct {
  float* ptr;
  float array[100];
} A;

A a;
float b[100];
__m128 tv
for (i = 0; i < 100; i += 4)
{
  tv = _mm_load_ps(b + i);
  _mm_store_ps(a.array + i, tv);
  tv = _mm_load_ps(b + i+1);
  _mm_storeu_ps(a.ptr + i, tv);
}
```

- accesses to regions and their direct subregions are aligned
- no reasoning about the index

Scout: Capabilities

Using **aligned** and **nontemporal**:

```
typedef struct {
  float* ptr;
  float array[100];
} A;

A a;
float b[100];
#pragma ... aligned(a, b[i])
for (i = 0; i < 100; ++i)
{
  a.array[i] = b[i];
  a.ptr[i] = b[i+1];
}
```

→

```
typedef struct {
  float* ptr;
  float array[100];
} A;

A a;
float b[100];
__m128 tv
for (i = 0; i < 100; i += 4)
{
  tv = _mm_load_ps(b + i);
  _mm_store_ps(a.array + i, tv);
  tv = _mm_loadu_ps(b + i+1);
  _mm_storeu_ps(a.ptr + i, tv);
}
```

- accesses to regions and their direct subregions are aligned
- no reasoning about the index → denote the access directly
  - all names must be declared before the pragma line

Scout: Capabilities

Using **aligned** and **nontemporal**:

```
typedef struct {
  float* ptr;
  float array[100];
} A;

A a;
float b[100];
#pragma ... aligned(a.ptr)
            nontemporal(a.ptr)
for (i = 0; i < 100; ++i)
{
  a.ptr[i] = b[i+1];
}
```

→

```
typedef struct {
  float* ptr;
  float array[100];
} A;

A a;
float b[100];
__m128 tv
for (i = 0; i < 100; i += 4)
{
  tv = _mm_loadu_ps(b + i+1);
  _mm_stream_ps(a.ptr + i, tv);
}
```

- accesses to regions and their direct subregions are aligned
- no reasoning about the index → denote the access directly
  - all names must be declared before the pragma line
- SSE nontemporal streaming requires aligned data

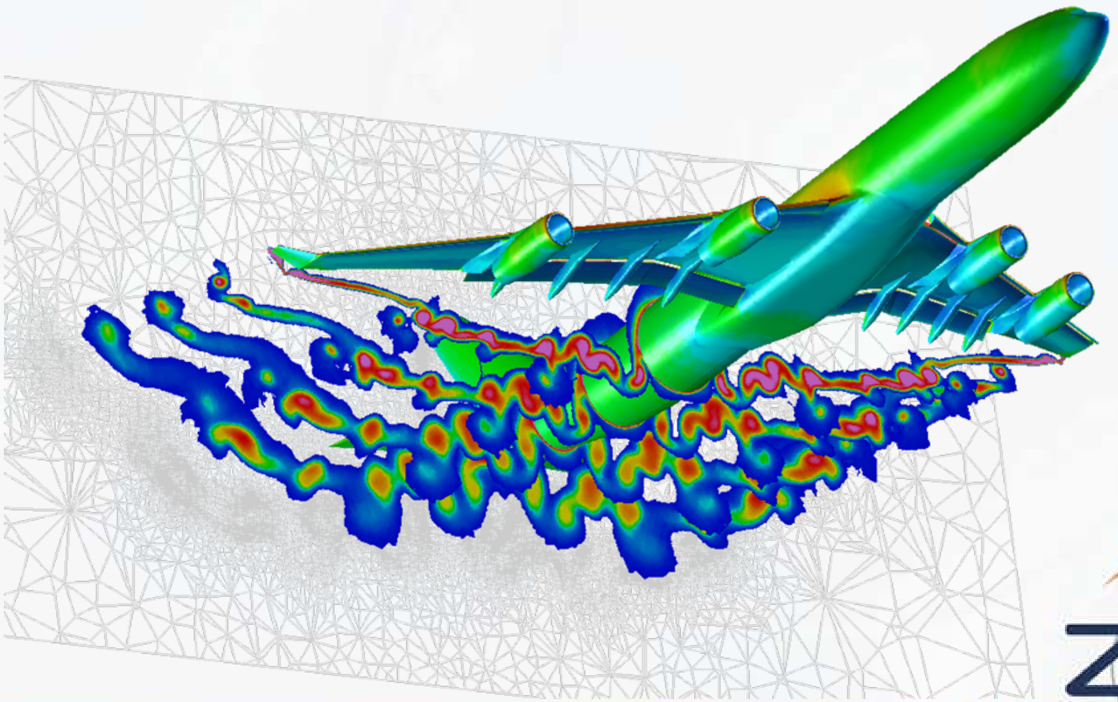
Agenda today

1. Motivation
   - where we come from: the HiCFD project
   - state of the art of (auto) vectorization
2. Introducing Scout
   - overview
   - background: using the clang framework for source-to-source transformation
   - features, capabilites, configuration
3. **Applying Scout**
   - **measurements of two real-world CFD codes**
4. Advanced vectorization techniques
   - Register blocking
   - Gather and scatter operations
5. Discussion

Scout: Measurements

first CFD computation kernel computes flow around air planes

- unstructured grid → indirect indexing
- arrays-of-structure data layout
- partial vectorization
  - enforcing compiler auto-vectorization by pragmas lead to incorrect results
- double precision, SSE platform → two vector lanes



Scout: Measurements

- Compiler: Intel 11.1, Windows 7, Intel Core 2 Duo, 2.4 MHz

```
int d [100];
#pragma scout loop vectorize
for (i = 0; i < 100; ++i) {
  int j = d[i];
  // first computations with
  // a[j], b[j] aso.
}

#pragma scout loop vectorize
for (i = 0; i < 100; ++i) {
  int j = d[i];
  // second computations with
  // a[j], b[j] aso.
}
```

speedup relation	original to vectorized
Grid 1	1.070
Grid 2	1.075

ideally a value near 2.0

→ unsatisfying result

Scout: Measurements

- lot of consecutive loops traverses over the same data structures:

```
int d [100];
#pragma scout loop vectorize
for (i = 0; i < 100; ++i) {
  int j = d[i];
  // first computations with
  // a[j], b[j] aso.
}

#pragma scout loop vectorize
for (i = 0; i < 100; ++i) {
  int j = d[i];
  // second computations with
  // a[j], b[j] aso.
}
```

hand-crafted →

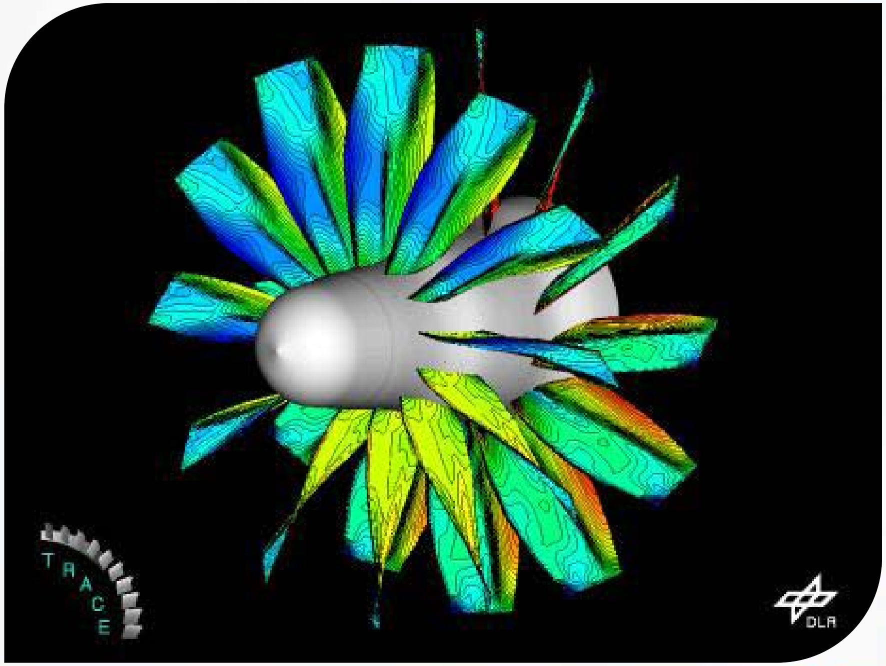
```
int d [100];
#pragma scout loop vectorize
for (i = 0; i < 100; ++i) {
  int j = d[i];
  // first computations with
  // a[j], b[j] aso.


  // second computations with
  // a[j], b[j] aso.
}
```

speedup relation	original to vectorized	merged to merged+vect.	original to merged+vect.
Grid 1	1.070	1.391	1.489
Grid 2	1.075	1.381	1.484

Scout: Measurements

second CFD computation kernel computes interior flows of jet turbines

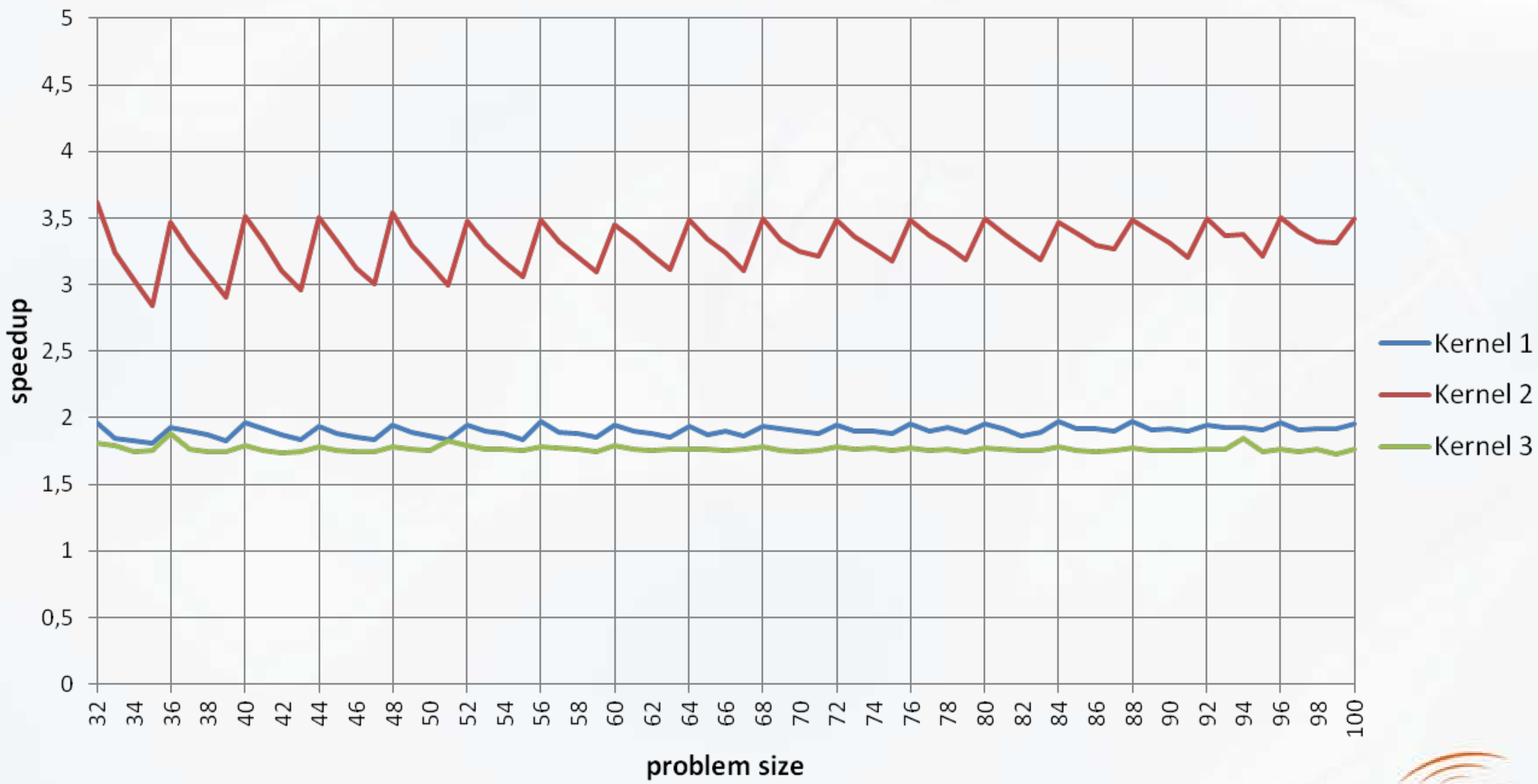
- structured grid → direct indexing
- arrays-of-structure data layout
- divided in three sub-kernels
- vectorization of complete loops



Scout: Measurements

a CFD computation kernel computing interior flows of jet turbines

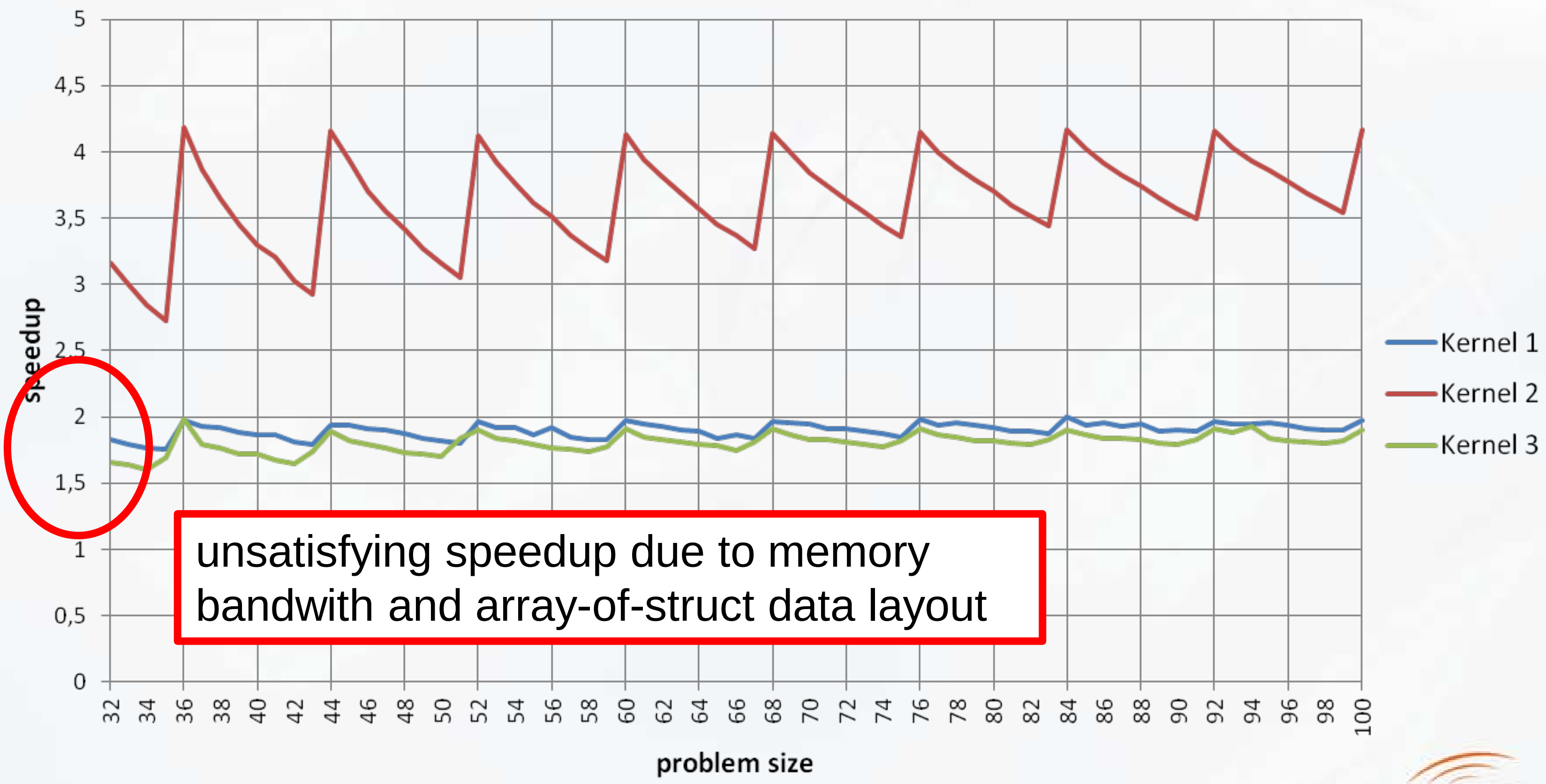
- three-dimensional grid → (problem size)³ cells to compute
- single precision, target architecture: SSE → 4 vector lanes



Scout: Measurements

a CFD computation kernel computing interior flows of jet turbines

- single precision, target architecture: AVX → 8 vector lanes



Agenda today

1. Motivation
   - where we come from: the HiCFD project
   - state of the art of (auto) vectorization
2. Introducing Scout
   - overview
   - background: using the clang framework for source-to-source transformation
   - features, capabilites, configuration
3. Applying Scout
   - measurements of two real-world CFD codes
4. **Advanced vectorization techniques**
   - **Register blocking**
   - **Gather and scatter operations**
5. Discussion

Scout: Register Blocking

derived from loop blocking (aka loop tiling):

- treats register file like a zero-level cache
- example: SIMD architecture provides 4 vector lanes

```
#pragma ... size(8)
for (i = si; i < ei; ++i)
{
    s_i;
    t_i;
}
```

→

```
for (i = si; i < ei-7; i += 8)
{
    s_i:i+3;
    s_i+4:i+7;
    t_i:i+3;
    t_i+4:i+7;
}
```

- vectorized statements are logically blocked, technically unrolled

Scout: Register Blocking

derived from loop blocking (aka loop tiling):

- treats register file like a zero-level cache
- example: SIMD architecture provides 4 vector lanes

```
float b[100], d[100], x;

#pragma ... size(8)
for (i = 0; i < 100; ++i)
{
  x = d[i];
  for (j = 0; j < k; ++j)
  {
    x += b[j];
  }
  d[i] = x;
}
```

→

```
float b[100], d[100], x;

for (i=0; i < 100 - 7; i += 8)
{
  dv0 = _mm_loadu_ps(d+i);
  dv1 = _mm_loadu_ps(d+i+4);
  for (j = 0; j < k; ++j)
  {
    tv0 = _mm_set1_ps(b[j]);
    dv0 = _mm_add_ps(dv0,tv0);
    dv1 = _mm_add_ps(dv1,tv0);
  }
  _mm_storeu_ps(d+i, dv0);
  _mm_storeu_ps(d+i+4, dv1);
}
```

- number of loads for **tv0** halved

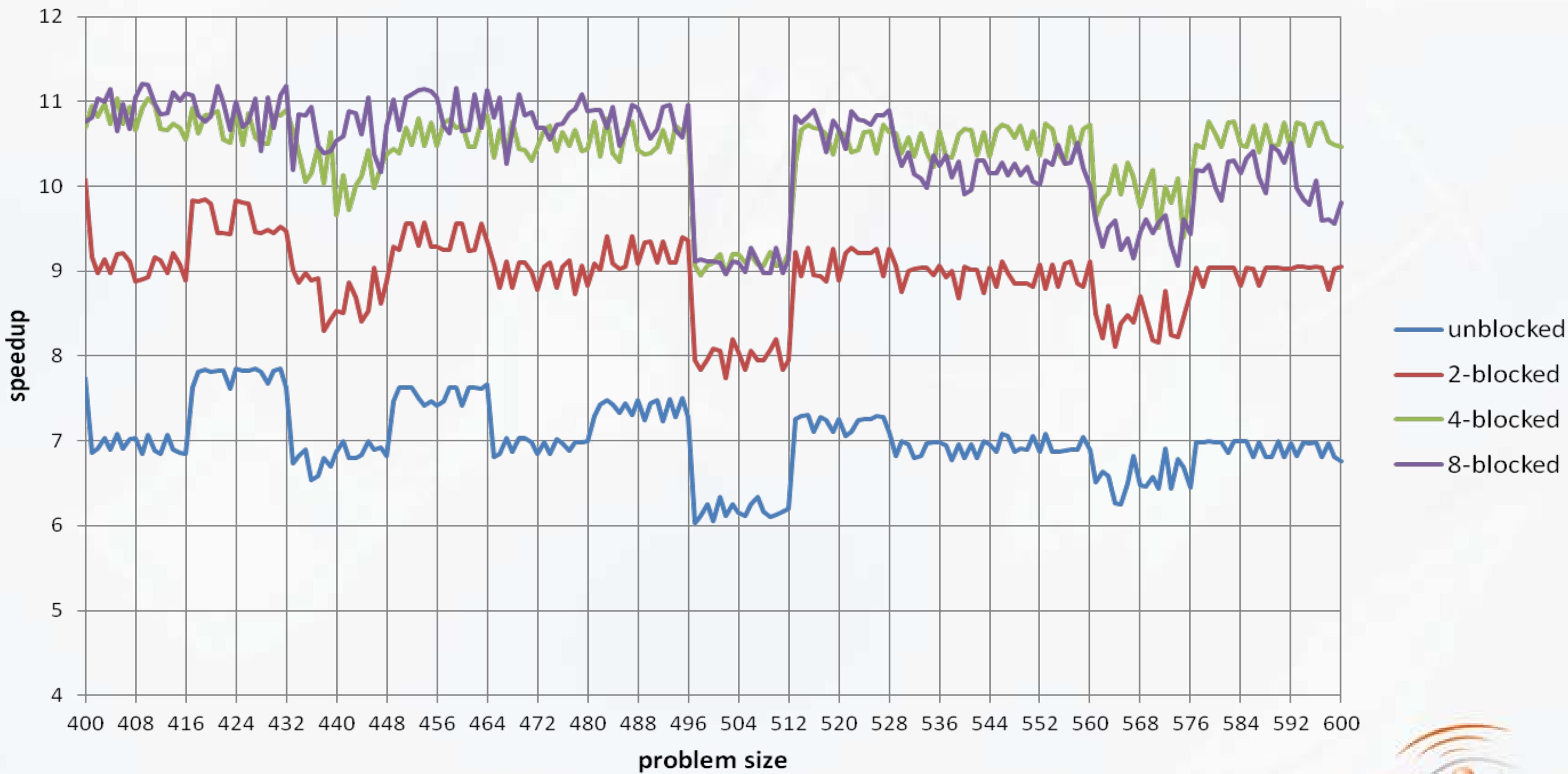
Scout: Register Blocking

- useful for loop-invariant variables
  - e.g. variables in inner loops not depending on the outer-loop index
- test case derived from production code:

```
#pragma scout loop vectorize size(BLOCK_SIZE) align(a,b,c)
  for (i = 0; i < S; ++i)
  {
    for (j = 0; j < G; j++)
    {
      float x = a[j];
      for (d = 0; d < D; d++)
      {
        x += b[j*D+d] * c[d*S+i];
      }
      output[i * G + j] = x;
    }
  }
```

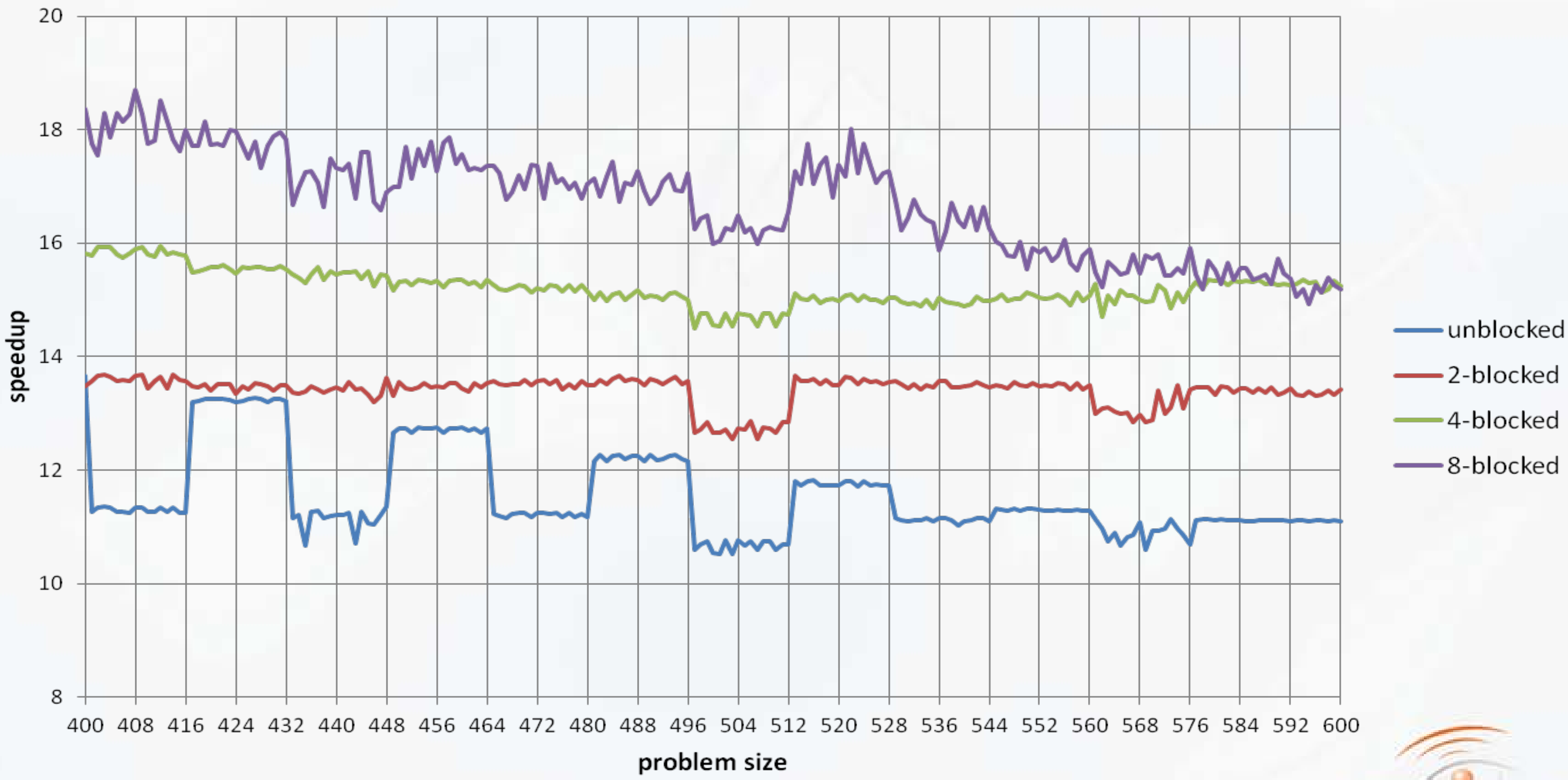
Scout: Register Blocking

- ICC12, SSE, 8 registers, no scalar residual loop:
  - no spillings in up to 4-blocked loops
  - 8-blocked loop: 2 spillings in innermost and 8 in 2nd innermost loop



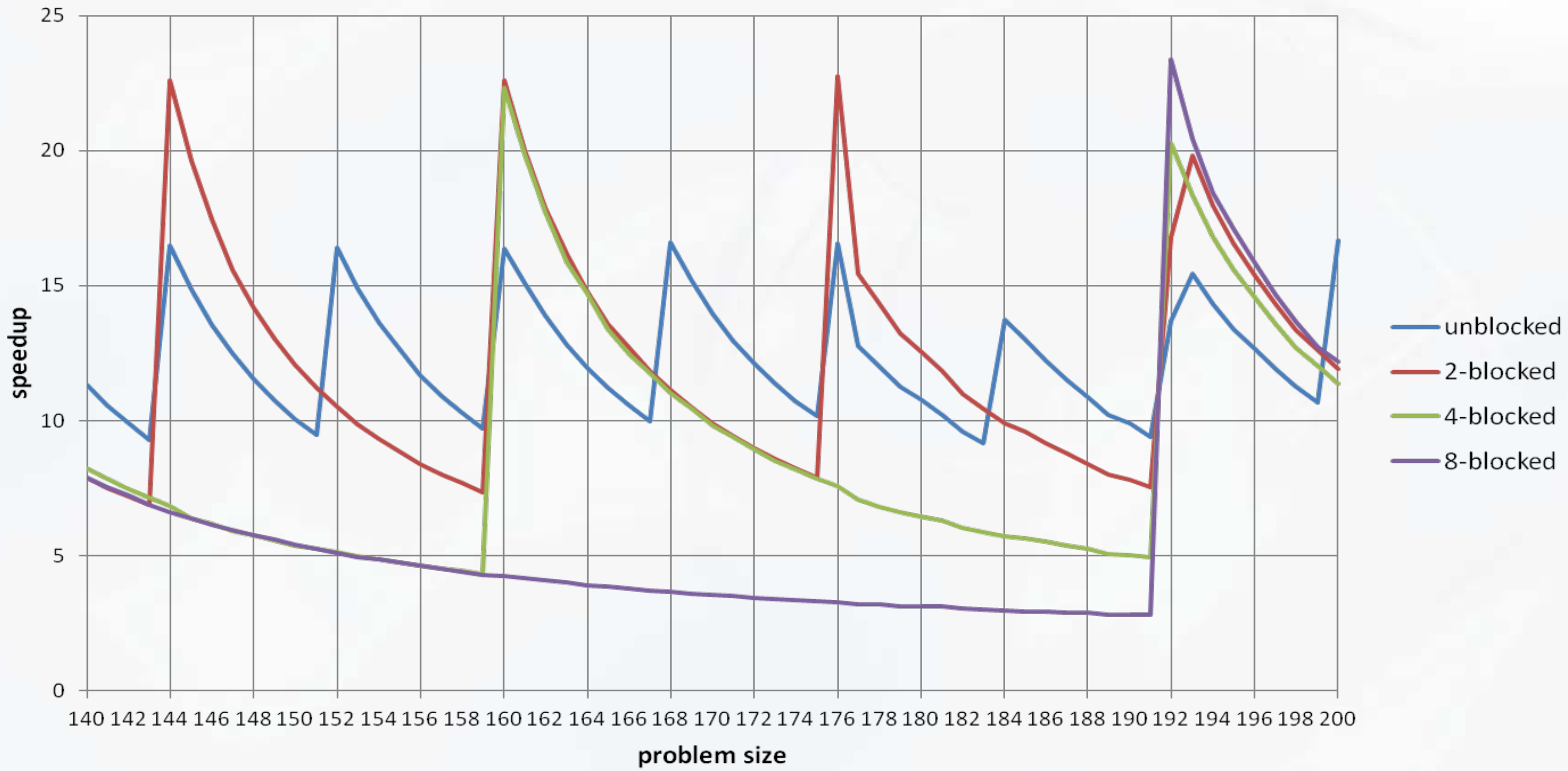
Scout: Register Blocking

- ICC12, AVX, 16 registers, no scalar residual loop:
  - no spillings in up to 4-blocked loops
  - 8-blocked loop: 5 spillings in the 2nd innermost loop (no spilling in the innermost loop)



Scout: Register Blocking

- beware of simple scalar residual loops!
  - same nice speedup only at the peaks



Scout: Gather / Scatter

- array-of-struct data layout forces the use of composite loads and stores
  - composite intrinsics (e.g. `_mm_set_pd(x,y)`) result in a series of scalar assembly instructions
  - broader vector registers makes the problem even worse

Thus either

→ rearrange your data layout

or

→ use gather and scatter operations

Scout: Gather / Scatter

- AVX example with comile-time constant gather distance:

```
struct V { float x, y; };
struct A { V v[5]; };
A a[100];

#pragma scout loop vectorize
for (i=0; i<S; ++i) {
  s = a[i].v[1].x;
}
```

```
struct V { float x, y; };
struct A { V v[5]; };
A a[100];

for (i=0; i < S-7; i += 8) {

  s_v = _mm256_set_ps(
 a[i+7].v[1].x, a[i+6].v[1].x,
 a[i+5].v[1].x, a[i+4].v[1].x,
 a[i+3].v[1].x, a[i+2].v[1].x,
 a[i+1].v[1].x, a[i].v[1].x);

}
```

Scout: Gather / Scatter

- AVX2 example with comile-time constant gather distance:

```
struct V { float x, y; };
struct A { V v[5]; };
A a[100];

#pragma scout loop vectorize
for (i=0; i<S; ++i) {
  s = a[i].v[1].x;
}
```

```
struct V { float x, y; };
struct A { V v[5]; };
A a[100];

_m256i d_v =
 _mm256_set_epi32(
  sizeof(A)*7,sizeof(A)*6,
  sizeof(A)*5,sizeof(A)*4,
  sizeof(A)*3,sizeof(A)*2,
  sizeof(A)  ,0);

for (i=0; i < S-7; i += 8) {

 s_v = _mm256_i32gather_ps(
   &a[i].v[1].x, dist_v, 1);

}
```

- scalar inititialization outside of the loop

Scout: Gather / Scatter

- SDE of Intel gives some estimations of the effect

	total # of instructions		parallel portion	
	AVX	AVX 2	AVX	AVX 2
Kernel 1	1244	793	43%	90%
Kernel 2	2451	2232	37%	45%
Kernel 3	2885	2666	51%	57%

- Kernel 1: only scatter is missing
- Kernel 2: unrolled loop-variant condition
- Kernel 3: indirect indexing → needs a recursive gather → TODO

Scout: Gather / Scatter

Remember the promise I made: gather with arbitrary constant loop stride

```
int k = /*...*/;
double a[100], b[100];
#pragma scout loop vectorize
for (i = 0; i < 100; i += k) {
  a[i] = b[i] * b[i];
}
```

```
int k = /*...*/;
double a[100], b[100];
int kd = k * sizeof(double);
__m128d av1, av2;
__m128d ki = _mm_set_pi(
       kd * 3, kd * 2, kd, 0);
for (i = 0; i < 100 - 1;
     i += k * 4) {
  av1 = _mm256_i32gather_pd(
    b + i, art_vect1, 1);
  av2 = _mm_mul_pd(av1, av1);
  // ...
}
/* scalar epilog */
```

Scout: Conclusion

Coding Advices

- allowed statements in loop bodies:
  - (compound) assign expressions
    - including copy assignments of records
  - function calls
  - if- and for-statements
  - TODO: while, switch/case
- better use ?: - expressions for loop-variant conditions
  - vectorized blend operations are much faster than unrolled if-statements
- inlined functions must use SESE style
  - otherwise the inliner generates **`gotos`**
- functions configured for direct vectorization are not inlined
  - allows for integration of user-implemented vector code

Scout: Conclusion

SIMD is a rather cheap way to exploit data-parallelism

- even a simple application of Scout nets remarkable speedups
  - just augment your code with pragmas and put Scout in your makefile
- overall speedup of hybird-parallelized CFD production codes due to Scout:
  - TAU: 8 – 10 %
  - TRACE: 20 – 80 % (1-Proc/Node AVX/Sandy Bridge EP)
- out-of-the-box performance results:
  - exploit hardware development without programming effort
- great flexibility by using a source-to-source approach

Scout: Conclusion

Future

- C++ support:
  - improved function inlining
  - will require `-fno-access-control` or the like
- folllow-up project waits in the wings:
  - investigate automatic data layout transformations

Questions?

Code Snippets?

http://scout.zih.tu-dresden.de/