Compiling for Scalable Computing Systems – the Merit of SIMD

Ayal Zaks
Intel Corporation

Acknowledgements: too many to list
For Inspiration and Recognition of Science and Technology – the leading pre-university program of the Technion

05/02/2015 Technion Honors Dean Kamen, inventor and entrepreneur, with Honorary Doctorate
Takeaways

1. SIMD is mainstream and ubiquitous in HW
2. Compiler support for SIMD is necessary
Takeaways

1. SIMD is mainstream and ubiquitous in HW
2. Compiler support for SIMD is necessary, and it’s fun 😊
Takeaways

1. SIMD is mainstream and ubiquitous in HW
2. Compiler support for SIMD is necessary, and it’s fun 😊, but it’s insufficient 🙁
3. Help SIMD become mainstream in SW
   – with Compiler, Programming Language and Tools support
## More Cores, Threads, Wider SIMD

<table>
<thead>
<tr>
<th>Core(s)</th>
<th>Threads</th>
<th>SIMD Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>128</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>128</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>128</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>128</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>256</td>
</tr>
<tr>
<td>12</td>
<td>24</td>
<td>256</td>
</tr>
<tr>
<td>18</td>
<td>36</td>
<td>256</td>
</tr>
</tbody>
</table>

1. SIMD is mainstream and ubiquitous in HW

*Product specification for launched and shipped products are available on ark.intel.com.

1. Not launched or in planning.
Single Instruction Multiple Data in a

```
float a[N], b[N];
for (int i = 0; i < 8; ++i) {
    c[i] = a[i] + b[i];
}
VADDPS YMM0, YMM1, YMM2
```

```
float a[N], b[N];
for (int i = 0; i < 16; ++i) {
    c[i] = a[i] + b[i];
}
VADDPS ZMM0, ZMM1, ZMM2
```

Compiler Vectorization

16 x 256-bit registers
In each register, e.g.,
8 float or 4 double or
8 int or 4 long

32 x 512-bit registers
In each register, e.g.,
16 float or 8 double or
16 int or 8 long
Wider Vectors + Richer Instructions

2. Compiler support for SIMD is necessary
“The vectorizer is truly such a wonderfully sophisticated and delicate algorithm. Debugging it to find and pin-point the unlikely problem is something I found, simply put, fun.”

Dr. Shahar Timnat
The CRAY-1’s Fortran compiler (CFT) is designed to give the scientific user immediate access to the benefits of the CRAY-1’s vector processing architecture. An optimizing compiler, CFT, “vectorizes” innermost DO loops. Compatible with the ANSI 1966 Fortran Standard and with many commonly supported Fortran extensions, CFT does not require any source program modifications or the use of additional nonstandard Fortran statements to achieve vectorization. Thus the user’s investment of hundreds of man months of effort to develop Fortran programs for other contemporary computers is protected.
Innermost DO loop Vectorization

<table>
<thead>
<tr>
<th>K=1</th>
<th>K=2</th>
<th>K=1..2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ld C(1)</td>
<td>Ld C(2)</td>
<td>Ld C(1) Ld C(2)</td>
</tr>
<tr>
<td>Ld B(1)</td>
<td>Ld B(2)</td>
<td>Ld B(1) Ld B(2)</td>
</tr>
<tr>
<td>Add</td>
<td>Add</td>
<td>Add Add</td>
</tr>
<tr>
<td>St A(1)</td>
<td>St A(2)</td>
<td>St A(1) St A(2)</td>
</tr>
</tbody>
</table>

Scalar code       GCC by Dorit Nuzman/IBM LLVM by Nadav Rotem/Apple

Vector code generation is straightforward
Emphasis on analysis and disambiguation
Vectorization Today

for(p=0; p<N; p++) {
    // Brown work
    if(...) {
        // Green work
    } else {
        // Red work
    }
    while(...) {
        // Gold work
        // Purple work
    }
    y = foo (x);
    // Pink work
}

Insufficient: Vector code generation has become a more difficult problem
It’s hard to vectorize what diverges

Two main Divergence challenges for SIMD: 1. Control, 2. Data
Vectorization Today: Challenges*

- Divergence Analysis
- Predication and Masking Optimizations
- Outer-loop Vectorization
- Less-than-full-vector Vectorization
- Gather/Scatter Optimizations
- Sophisticated Idioms Vectorization
- Partial Vectorization
- Function Vectorization

Increasing need for user guidance for correctness and profitability

* nonexclusive list
Parallel Program Development

- Threading using operating system calls
  - `pthread_create(...)`
  - `pthread_join(...)`

- Applicability
- Development cost
- Performance
- Maintainability
- Scalability

- Industry standard directives for Explicit Parallelization
  - `#pragma omp for`
  - `for (i=0; i<...) { ... }
  - `cilk_for(i=0; i<...) { ...`
  - Or ...

- Automatically using the compiler
Vector Program Development

Applicability
Development cost
Performance

Explicit Vectorization

MS128 t1, t2;
t1 = _mm_mulps(&a[i], &b[i]);
t2 = _mm_addps(t1, t1);

#pragma omp simd
for (i=0; i<n; i++)
  c[i] = a[i] + b[i];

Automatic using the compiler

Intrinsics

Maintainability
Scalability
Explicit Vectorization – Example

```c
#pragma omp parallel for
for (int y = 0; y < ImageHeight; ++y) {
    #pragma omp simd
    for (int x = 0; x < ImageWidth; ++x) {
        count[y][x] = mandelbrot(in_vals[y][x], max_iter);
    }
}
```

```c
#pragma omp declare simd uniform(limit)
int mandelbrot(fcomplex c, int limit) {
    fcomplex z = c; int iters = 0;
    while ((cabsf(z) < 2.0f) && (iters < limit)) {
        z = z * z + c; iters++;
    }
    return iters;
}
```
(Explicit) Whole Function Vectorization

  - Expands vectorization across function boundary
  - Originally for implicit data-parallel languages

- OpenMP 4.0:
  - Employs WFV with new, explicit SIMD annotations
  - Recent examples published include SIMD examples
    http://openmp.org/mp-documents/openmp-examples-4.0.2.pdf
Some Recent References re:WFV*

- **Books:**
  - High Performance Parallelism Pearls volume 2, James Reinders and Jim Jeffers, August 2015

- **Conference Papers:**
  - Locality-centric thread scheduling for bulk-synchronous programming models on CPU architectures, Hee-Seok Kim et al., CGO 2015. Effectively maximizes WFV factors where possible and profitable on CPUs
  - Exploring the Design Space of SPMD Divergence Management on Data-Parallel Architectures, Yunsup Lee et al., Micro-47. Compares SW and HW techniques for supporting control divergence on GPUs
  - The Impact of the SIMD Width on Control-Flow and Memory Divergence, Thomas Schaub et al., HiPEAC 2015
  - Optimizing Overlapped Memory Accesses in User-directed Vectorization, Diego Caballero et al., to appear in ICS 2015. Technique and OpenMP proposal to support memory divergence on CPU and MIC

- **Workshop papers**
  - Predicate Vectors If You Must, Shahar Timnat et al., WPMVP@PPoPP 2014
  - Streamlining Whole Function Vectorization in C using Higher Order Vector Semantics, Gil Rapaport et al., PLC@IPDPS 2015

* nonexclusive list
Feedback from static optimizing compilation improving
Intel® Advisor XE – Vectorization Advisor

Integrates Compiler diagnostics + Performance Data + SIMD efficiency stats
Guidance: detect penalties and recommend improvements using OpenMP 4.0
Deep dive: dependence checkers, memory access pattern analysis
Takeaways

1. SIMD is mainstream and ubiquitous in HW
2. Compiler support is necessary, fun, but insufficient
3. Help SIMD be mainstream in SW by Explicit Vectorization
   - Need user guidance for today’s applications
   - Similar to what OpenMP/Cilk/TBB did for parallelization
   - Maps threaded execution to SIMD hardware
   - Has advantages in development cost, applicability, performance, maintenance, scalability
4. Tools are improving
The 4th Compiler, Architecture and Tools Conference

Mark Your Calendars and Stay Tuned!

November 23rd, 2015

@ Intel IDC Haifa

Organized by: Gadi Haber, Ayal Zaks, Michal Nir and Leeor Peled from Intel SSGi and PEG
Dorit Nuzman from IBM, Erez Petrank from the Technion, Yosi Ben-Asher from Haifa U.
Legal Disclaimer & Optimization Notice

• INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

• Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

• Copyright © 2014, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
THE END
OpenMP 4.0

**simd construct**

Based on a proposal from Intel based on customer success with the Intel® Cilk™ Plus features in Intel compilers.

**Summary**

The `simd` construct can be applied to a loop to indicate that the loop can be transformed into a SIMD loop (that is, multiple iterations of the loop can be executed concurrently using SIMD instructions).

```c
#pragma omp simd reduction(+:val) reduction(+:val2)
for (int pos = 0; pos < RAND_N; pos++) {
    float callValue = expectedCall(Sval,Xval,MuByT,VBySqrtT,l_Random[pos]);
    val += callValue;
    val2 += callValue * callValue;
}
```
**OpenMP 4.0**

**simd construct**

**Summary**

The `simd` construct can be applied to a loop to indicate that the loop can be transformed into a SIMD loop (that is, multiple iterations of the loop can be executed concurrently using SIMD instructions).

```
#pragma omp simd [clause[,...] clause ...] new-line
for-loops
```

where `clause` is one of the following:

- `safelen(length)`
- `linear(list[,linear-step])`
- `aligned(list[,alignment])`
- `private(list)`
- `lastprivate(list)`
- `reduction(reduction-identifier:list)`
- `collapse(n)`

The `simd` directive places restrictions on the structure of the associated `for-loops`. Specifically, all associated `for-loops` must have canonical loop form (Section 2.6 on page 51).

Note: per the OpenMP standard, the “for-loop” must have canonical loop form.
declare simd construct

Summary
The declare simd construct can be applied to a function (C, C++ and Fortran) or a subroutine (Fortran) to enable the creation of one or more versions that can process multiple arguments using SIMD instructions from a single invocation from a SIMD loop. The declare simd directive is a declarative directive. There may be multiple declare simd directives for a function (C, C++, Fortran) or subroutine (Fortran).

C/C++

```c
#pragma omp declare simd [clause[[], clause] ...] new-line
```

Fortran

```fortran
!omp declare simd(proc-name) [clause[[], clause] ...]
```

where clause is one of the following:

C/C++

- `simdlen(length)`
- `linear(argument-list[:constant-linear-step])`
- `aligned(argument-list[:alignment])`
- `uniform(argument-list)`
- `inbranch`
- `notinbranch`

Fortran

- `simdlen(length)`
- `linear(argument-list[:constant-linear-step])`
- `aligned(argument-list[:alignment])`
- `uniform(argument-list)`
- `inbranch`
- `notinbranch`
OpenMP 4.0

Loop SIMD construct

Parallelize and Vectorize

Summary

The loop SIMD construct specifies a loop that can be executed concurrently using SIMD instructions and that those iterations will also be executed in parallel by threads in the team.

Syntax

C/C++

```
#pragma omp for simd [clause [...] new-line
for-loops
```

where `clause` can be any of the clauses accepted by the `for` or `simd` directives with identical meanings and restrictions.

C/C++

```
!$omp do simd [clause [...] do-loops
[nowait]
```

where `clause` can be any of the clauses accepted by the `simd` or `do` directives, with identical meanings and restrictions.

If an `end do simd` directive is not specified, an `end do simd` directive is assumed at the end of the do-loop.
HW provides many parallel SIMD lanes
SW+Compiler need to run them, or