Trends in systems and how to get efficient performance

Martin Hilgeman
HPC Consultant
martin.hilgeman@dell.com
The landscape is changing

“We are no longer in the general purpose era… the argument of tuning software for hardware is moot. Now, to get the best bang for the buck, you have to tune both.”

- Kushagra Vaid, general manager of server engineering, Microsoft Cloud Solutions

https://www.nextplatform.com/2017/03/08/arm-amd-x86-server-chips-get-mainstream-lift-microsoft/amp/
Moore’s Law vs. Amdahl’s Law

- The clock speed plateau
- The power ceiling
- IPC limit

Industry is applying Moore’s Law by adding more cores

Meanwhile Amdahl’s Law says that you cannot use them all efficiently

Moore’s Law vs Amdahl's Law - “too Many Cooks in the Kitchen”

Industry is applying Moore’s Law by adding more cores

Meanwhile Amdahl’s Law says that you cannot use them all efficiently
System trend over the years (1)

~1970 - 2000

2005

Multi-core:

TOCK
System trend over the years (2)

2007

Integrated Memory controller:

2012

Integrated PCIe controller:

Internal Use - Confidential
Future

Integrated Network Fabric Adapter:

SoC designs:

Internal Use - Confidential
Improving performance - what levels do we have?

- **Challenge:** Sustain performance trajectory without massive increases in cost, power, real estate, and unreliability.

- **Solutions:** No single answer, must **intelligently turn** “Architectural Knobs”

\[
(Freq) \times \left( \frac{\text{cores}}{\text{socket}} \right) \times (\#\text{sockets}) \times \left( \frac{\text{inst or ops}}{\text{core} \times \text{clock}} \right) \times (\text{Efficiency})
\]

1. Hardware performance
2. What you really get
3. Software performance
Turning the knobs 1 - 4

1. Frequency is unlikely to change much. Thermal/Power/Leakage challenges

2. Moore’s Law still holds: 130 -> 14 nm. LOTS of transistors

3. Number of sockets per system is the easiest knob. Challenging for power/density/cooling/networking

4. IPC still grows. FMA3/4, AVX, FPGA implementations for algorithms. Challenging for the user/developer
What is Intel telling us?

- **Thurley Platform**
  - Intel® Microarchitecture Codename Nehalem
  - Nehalem
  - 45nm
  - New Micro-architecture
  - Tock

- **Romley Platform**
  - Intel® Microarchitecture Codename Sandy Bridge
  - Sandy Bridge
  - 32nm
  - New Micro-architecture
  - Tock

- **Grantley Platform**
  - Intel® Microarchitecture Codename Haswell
  - Haswell
  - 22nm
  - New Micro-architecture
  - Tock

- **Purley Platform**
  - Intel® Microarchitecture Codename Skylake
  - Skylake
  - 14nm
  - New Micro-architecture
  - Tock

**Future Product**

Internal Use - Confidential
New capabilities according to Intel

Thurley Platform
- Intel® Microarchitecture Codename Nehalem
  - Nehalem
  - 45nm
  - New Micro-architecture

Westmere
- New Process Technology

Sandy Bridge
- New Micro-architecture
- 32nm
- New Process Technology

Ivy Bridge
- 22nm
- New Micro-architecture
- New Process Technology

Grantley Platform
- Intel® Microarchitecture Codename Haswell
  - Haswell
  - 22nm
  - New Micro-architecture
  - New Process Technology

Broadwell
- 14nm
- New Micro-architecture
- New Process Technology

Purley Platform
- Intel® Microarchitecture Codename Skylake
  - Skylake
  - 14nm
  - New Micro-architecture
  - New Process Technology

Future Product

SSSE3
- 2007

SSE4
- 2009

AVX
- 2012

AVX
- 2013

AVX2
- 2014

AVX2
- 2015

AVX-512
- 2017

Internal Use - Confidential
## The state of ISV software

<table>
<thead>
<tr>
<th>Segment</th>
<th>Applications</th>
<th>Vectorization support</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFD</td>
<td>Fluent, LS-DYNA, STAR CCM+</td>
<td>Limited SSE2 support</td>
</tr>
<tr>
<td>CSM</td>
<td>CFX, RADIOSS, Abaqus</td>
<td>Limited SSE2 support</td>
</tr>
<tr>
<td>Weather</td>
<td>WRF, UM, NEMO, CAM</td>
<td>Yes</td>
</tr>
<tr>
<td>Oil and Gas</td>
<td>Seismic processing</td>
<td>Not applicable</td>
</tr>
<tr>
<td></td>
<td>Reservoir Simulation</td>
<td>Yes</td>
</tr>
<tr>
<td>Chemistry</td>
<td>Gaussian, GAMESS, Molpro</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Molecular dynamics</td>
<td>NAMD, GROMACS, Amber,…</td>
<td>PME kernels support SSE2</td>
</tr>
<tr>
<td>Biology</td>
<td>BLAST, Smith-Waterman</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Molecular mechanics</td>
<td>CPMD, VASP, CP2k, CASTEP</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Bottom line:** ISV support for new instructions is poor. Less of an issue for in-house developed codes, but programming is hard.
Meanwhile the bandwidth is suffering
Add to this the Memory Bandwidth and System Balance

Sustained (streaming) Memory Bandwidth is falling behind Peak FLOPS rates, but every other kind of memory access is falling behind even faster....

And data is becoming sparser (think “Big Data”)

- Most entries are zero
- Hard to exploit SIMD
- Hard to use caches

• This has very low arithmetic density and hence memory bound
• Common in CFD, but also in genetic evaluation of species
## What does Intel do about these trends?

<table>
<thead>
<tr>
<th>Problem</th>
<th>Westmere</th>
<th>Sandy Bridge</th>
<th>Ivy Bridge</th>
<th>Haswell</th>
<th>Broadwell</th>
<th>Skylake</th>
</tr>
</thead>
<tbody>
<tr>
<td>QPI bandwidth</td>
<td>No problem</td>
<td>Even better</td>
<td>Two snoop modes</td>
<td>Three snoop modes</td>
<td>Four (!) snoop modes</td>
<td>• UPI</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• COD snoop modes</td>
</tr>
<tr>
<td>Memory bandwidth</td>
<td>No problem</td>
<td>Extra memory channel</td>
<td>Larger cache</td>
<td>Extra load/store units</td>
<td>Larger cache</td>
<td>• Extra load/store units</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• +50% memory channels</td>
</tr>
<tr>
<td>Core frequency</td>
<td>No problem</td>
<td>• More cores</td>
<td>• Even more cores</td>
<td>• Still more cores</td>
<td>• Again even more cores</td>
<td>• More cores</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• AVX</td>
<td>• Above TDP Turbo</td>
<td>• AVX2</td>
<td>• optimized FMA</td>
<td>• Larger OOO</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Better Turbo</td>
<td></td>
<td>• Per-core Turbo</td>
<td>• Per-core Turbo based on instruction type</td>
<td>• AVX engine</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• AVX-512</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• 3 different core frequency modes</td>
</tr>
</tbody>
</table>
Optimizing without having the source code
Tuning knobs for performance

Hardware tuning knobs are limited, but there's far more possible in the software layer

- Compiler hints
- Source changes
- Adding parallelism
- MPI (parallel) tuning
- Use of performance libs (Math, I/O, IPP)
- I/O cache tuning
- Process affinity
- Memory allocation
- BIOS
- P-states
- Memory profile

Easy to hard
dell_affinity.exe – automatically adjust process affinity for the architecture

Original: 152 seconds

With dell_affinity: 131 seconds

<table>
<thead>
<tr>
<th>Mode</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>6342</td>
</tr>
<tr>
<td>Platform MPI</td>
<td>6877</td>
</tr>
<tr>
<td>dell_affinity</td>
<td>4844</td>
</tr>
</tbody>
</table>
MPI profiling interface – dell_toolbox

- Works with every application using MPI - Plugin-and-play!
- Provides insight in computation/computation ratio
- Allows to dig deeper into MPI internals
- Generates CSV files that can be imported into Excel
STAR CCM+ MPI tuning for F1 team

R022, lemans_trim_105m, 256 cores, PPN=16

AS-IS: 2812 seconds
Tuned MPI: 2458 seconds

12.6% speedup
Altair Hyperworks MPI tuning

Neon benchmark
16x M620, E5-2680v2, Intel MPI

14.4% faster

As-is
1507 seconds

Tuning knob #1
1417 seconds

Tuning knob #2
1287 seconds

Wall clock time (s)

0 40 80 120 160 200 240 280
0 40 80 120 160 200 240 280

MPI Rank

0 40 80 120 160 200 240 280

Wall clock time (s)

0 40 80 120 160 200 240 280
0 40 80 120 160 200 240 280

As-is
Tuning knob #1
Tuning knob #2

Altair Hyperworks

MPI_Waitany
MPI_Wait
MPI_Send
MPI_Reduce
MPI_Recv
MPI_Isend
MPI_Irecv
MPI_Gather
MPI_Bcast

App

16x M620, E5-2680v2, Intel MPI

14.4% faster

As-is
1507 seconds

Tuning knob #1
1417 seconds

Tuning knob #2
1287 seconds

Wall clock time (s)

0 40 80 120 160 200 240 280
0 40 80 120 160 200 240 280

MPI Rank

0 40 80 120 160 200 240 280
0 40 80 120 160 200 240 280

Wall clock time (s)

0 40 80 120 160 200 240 280
0 40 80 120 160 200 240 280

As-is
Tuning knob #1
Tuning knob #2

Altair Hyperworks

MPI_Waitany
MPI_Wait
MPI_Send
MPI_Reduce
MPI_Recv
MPI_Isend
MPI_Irecv
MPI_Gather
MPI_Bcast

App
Optimization examples
Case study: 3D anti-leakage FFT
Seismic reflection and processing

Oil companies use reflection seismology to estimate the properties of the Earth’s surface by analyzing the reflection of seismic waves

Image obtained from: https://en.wikipedia.org/wiki/Reflection_seismology#/media/File:Seismic_Reflection_Principal.png
Seismic data processing

Data processing is being used to reconstruct the reflected seismic waves into a picture (deconvolution)

These events can be relocated into **space** or **time** to the exact location by discrete Fourier Transformations

\[ F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-i\omega t} dt \]
Benchmarking and profiling

- A single record was used to limit the job turnaround time
- All cores (including logical processors) are used
- Profiling was done using Intel® Advisor XE
  - Shows application hotspots with most time consuming loops
  - Shows data dependencies
  - Shows potential performance gains (only with 2016 compilers)
  - Makes compiler output easier to read (only with 2016 compilers)

Image obtained from: https://software.intel.com/en-us/intel-advisor-xe
Profile from a single record

- 76.5% scalar code
- 23.5% vectorized
- Two routines make up for the majority of the wall clock time
Data alignment and loop simplification

```
float *amp_tmp = mSub.amp_panel;
int m = 0;
int m1 = (np - ifq) * ny * nx;

for (int n1 = 0; n1 < ny; n1++) {
    for (int n2 = 0; n2 < nx; n2++) {
        float a = four_data[m1].r + four_out[m1].r;
        float b = four_data[m1].i + four_out[m1].i;
        amp_tmp[m] = (a * a + b * b);
        m1 ++;
        m ++;
    }
}

mSub.amp_panel = (float *) _mm_malloc(mySize * sizeof(float), 32);
float *amp_tmp = mSub.amp_panel;
int m1 = (np - ifq) * ny * nx;

for (int n1 = 0; n1 < ny * nx; n1++) {
    __assume_aligned(amp_tmp, 32);
    __assume_aligned(four_data, 32);
    __assume_aligned(four_out, 32);
    amp_tmp[n1] = (((four_data[m1 + n1].r + four_out[m1 + n1].r) *
                    (four_data[m1 + n1].r + four_out[m1 + n1].r)) +
                    (((four_data[m1 + n1].i + four_out[m1 + n1].i) *
                      (four_data[m1 + n1].i + four_out[m1 + n1].i));
```

AS-IS

```

```
Compiler diagnostic output

LOOP BEGIN at test3d_subs.cpp(701,4)
remark #15389: vectorization support: reference four_data has unaligned access
remark #15389: vectorization support: reference four_out has unaligned access
remark #15389: vectorization support: reference amp_panel has unaligned access
remark #15381: vectorization support: unaligned access used inside loop body
remark #15385: vectorization support: vector length 16
remark #15309: vectorization support: normalized vectorization overhead 0.393
remark #151300: LOOP WAS VECTORIZED
remark #15442: entire loop may be executed in remainder
remark #15450: unmasked unaligned unit stride loads: 2
remark #15451: unmasked unaligned unit stride stores: 1
remark #15460: masked strided loads: 4
remark #15475: --- begin vector loop cost summary ---
remark #15476: scalar loop cost: 29
remark #15477: vector loop cost: 3.500
remark #15478: estimated potential speedup: 6.310
remark #15488: --- end vector loop cost summary ---
LOOP END

LOOP BEGIN at test3d_subs.cpp(692,8)
remark #15388: vectorization support: reference amp_panel has aligned access
remark #15389: vectorization support: reference four_data has unaligned access
remark #15389: vectorization support: reference four_out has unaligned access
remark #15381: vectorization support: unaligned access used inside loop body
remark #15305: vectorization support: vector length 16
remark #15309: vectorization support: normalized vectorization overhead 0.120
remark #151300: LOOP WAS VECTORIZED
remark #15449: unmasked aligned unit stride stores: 1
remark #15450: unmasked unaligned unit stride loads: 2
remark #15460: masked strided loads: 8
remark #15475: --- begin vector loop cost summary ---
remark #15476: scalar loop cost: 43
remark #15477: vector loop cost: 3.120
remark #15478: estimated potential speedup: 9.070
remark #15488: --- end vector loop cost summary ---
LOOP END
Expensive type conversions

```c
for (int i = 0; i < nx; i++){
    int ii = i - nxh;
    float fii = (float) ii*scale + (float) (nxh);

    if (fii >= 0.0) {
        ii = (int) fii;
        ip = ii+1;
        if(ip < nx && ii >= 0) {
            t = (fjj - (float) jj);
            u = (fii - (float) ii);
            int n1 = jj*nx + ii;
            int n2 = jp*nx + ii;
            int n3 = jp*nx + ip;
            int n4 = jj*nx + ip;
            amp[m] += ((1.0-t)*(1.0-u)*amp_tmp[n1]+t*(1.0-u)*amp_tmp[n2]+t*u*amp_tmp[n3]+(1.0-t)*u*amp_tmp[n4]);
        }
    }
    m ++;
}
```

remark #15417: vectorization support: number of FP up converts: single precision to double precision 8  
test3d_subs.cpp(784,26) 
remark #15418: vectorization support: number of FP down converts: double precision to single precision 1   
test3d_subs.cpp(784,26) 
remark #15300: LOOP WAS VECTORIZED 
remark #15475: --- begin vector loop cost summary --- 
remark #15476: scalar loop cost: 48 
remark #15477: vector loop cost: 25.870 
remark #15478: estimated potential speedup: 1.780 
remark #15487: type converts: 12
Expensive type conversions - solution

```c
union convert_var_type { int i; float f; };
union convert_var_type convert_fii_1, convert_fii_2;

for (ii = -nxh; ii < nx - nxh; ii++) {
    convert_fii_1.f = ii * scale + nxh;
    fii = convert_fii_1.f;
    if (fii >= 0.0f) {
        convert_fii_2.i = fii;
        iii = convert_fii_2.i;
        if(iii < nx - 1 && iii >= 0) {
            convert_iii.f = iii;
            t = jjj - jjf;
            u = fii - convert_iii.f;
            n1 = jjj * nx + iii;
            n2 = (jjj + 1) * nx + iii;
            amp[m] += ((1.0f-t)*(1.0f-u)*amp_tmp[n1]+t*(1.0f-u)*amp_tmp[n2]
                        +t*u*amp_tmp[n2+1]+(1.0f-t)*u*amp_tmp[n1+1]);
        }
    }
    m ++;
}
```

remark #15300: LOOP WAS VECTORIZED
remark #15475: --- begin vector loop cost summary ---
remark #15476: scalar loop cost: 42
remark #15477: vector loop cost: 20.370
remark #15478: estimated potential speedup: 1.960
remark #15487: type converts: 3
What about parallelization?

Window processing is embarrassingly parallel

```c
#pragma omp parallel for schedule(dynamic, 1)
for (int iwin = 0; iwin < NumberOfWindows; iwin++) {
    /* thread number handling the window */
    int tn = omp_get_thread_num();
    << Do computation >>
}
```

Memory allocation is not

```c
for (int i = 0; i < numThreads; i++) {
    mySize = maxntr_win;
    if(!(mem_sub[i].trcid_win = new int[mySize])) ierr = 1;
    memset(mem_sub[i].trcid_win, 0, sizeof(int) * mySize);
    mb += sizeof(int) * mySize;
}
```
What happens?

Memory allocation is done by thread
All references for other threads end up on the NUMA domain of thread
**Correct way of “first touch” memory allocation**

Do the memory allocation and population in parallel so that each thread initializes its own memory

```c
for (int i = 0; i < numThreads; i++) {
    mySize = maxntr_win;
    if(!((mem_sub[i].trcid_win = new int[mySize]))) ierr = 1;
}

#pragma omp parallel
{
    int i = omp_get_thread_num();
    memset(mem_sub[i].trcid_win, 0, sizeof(int) * mySize);
    mb += sizeof(int) * mySize;
}
```
Final result of optimized code
### Final results

<table>
<thead>
<tr>
<th>Node</th>
<th>Original version</th>
<th>Optimized version</th>
</tr>
</thead>
<tbody>
<tr>
<td>E5-2670 @ 2.6 GHz, 32 threads</td>
<td>192 seconds</td>
<td>145 seconds (+24.4%)</td>
</tr>
<tr>
<td>E5-2680 v3 @ 2.5 GHz, 48 threads</td>
<td>156 seconds</td>
<td>112 seconds (+28.2%)</td>
</tr>
</tbody>
</table>

- 24.4% speedup obtained on SNB, 28.2% on HSW
- Even better with 16.1 compilers, optimized version:

<table>
<thead>
<tr>
<th>Node</th>
<th>With AVX</th>
<th>With FMA + AVX2</th>
</tr>
</thead>
<tbody>
<tr>
<td>E5-2680 v3 @ 2.5 GHz, 48 threads</td>
<td>86 seconds</td>
<td>68 seconds (+20.9%)</td>
</tr>
</tbody>
</table>