Utilizing the Horsepower - Vectorization and Parallelization

As probably all of you know Valleyview is a SSE4.2 SIMD Atom processor with 4 physical cores. This means, the SoC is capable of parallelization on instruction level ("vectorization") and on thread or process level respectively. Let's have a look into what this means. We don't claim to have a comprehensive description of vectorization and parallelization nor to have a fair comparison of the various methods (which would easily fill a book) but only to demonstrate on a simple example what it is all about.

Vectorization & Parallelization

We start with a simple example:

```c
for (int n=start; n<end; n++) {
    y[n] = 0.0f;
    for (int j=0; j<length; j++) {
        y[n] += x[n - length + j+1] * h[length - j + 1];
    }
}
```

Loops like this you probably regularly face e.g. on multimedia tasks. Having a closer look at the loop we see there are no dependencies in the inner loop. Moreover for each iteration step the same operation is performed. The 128-bit SIMD vector unit in VLV can perfectly run X[m..m+3]*H[l..l+3] in 1 vector multiplication for 4 floats rather than 4 scalar multiplications. That of course can yield a quite remarkable gain. But similary we could tile the loop and start 4 threads on all 4 cores in parallel to work on a subset of the problem. That's what is called parallelization. Both can be combined to run 16 multiplications in parallel rather than 1.

How to vectorize

For the sake of this description let's go to an even simpler but also more artificial source which generates some load in only 1 loop and doesn't require any data preset:

```c
#include <iostream>
#include <math.h>
using namespace std;

void ParallelApplyFoo(float a[], size_t n) {
    for (int i=0; i<n; i++) {
        a[i] = sqrtf(cosf(sqrtf(sinf(i)*sinf(i)) / cosf(i))*cosf(sqrtf(sinf(i)*sinf(i)) / cosf(i)) + 2.31f)/0.229879834f;
    }
}

int main(int argc, char** argv) {
    size_t size = 1000000;
    float* a = new float[size];
    ParallelApplyFoo(a, size);
    cout << "computation done\n";
    cout << "0: " << a[0] << " - " << size << ": " << a[size-1] << endl;
    return 0;
}
```

At a first stage let's take a baseline with the gcc and the switches KP uses by default:

and run the code on BfH with a simple timing

$ time ./loop_gcc
computation done
0: 7.91431 - 10000000: 7.03673
real 0m12.555s
user 0m12.433s
sys 0m0.104s

Ok, it takes us about 12.5s to run this code. With "top", "perf top", or VTune we could have a look for the CPU utilization but we leave this out here. Now, we're interested whether there might be some vector code in it already. We do a quick check by looking at the assembly - single precision vector instructions do end with "ps" (packed single):

$ objdump -S loop_gcc | grep ps
8048921: 0f 28 d9 movaps %xmm1,%xmm3
804894c: d9 5d dc fstps -0x24(%ebp)
8048970: d9 5d a4 fstps -0x5c(%ebp)
80489cd: d9 5d a4 fstps -0x5c(%ebp)
80489f3: d9 5d a4 fstps -0x5c(%ebp)
8048a07: d9 5d a4 fstps -0x5c(%ebp)

Ok, there are some hits - but that's not really vector code. It's just some copy and store instruction. I.E. gcc auto vectorization doesn't work with those settings. Looking into the gcc manual we get a hint that "-ftree-vectorize" or "-O3" would be required for auto vectorization. However, even with both switches on auto vectorization fails and the times are still pretty much the same 12.5 sec. With that we're left on the realms of so called "intrinsics" or "inline assembly". Intrinsics are specific compiler additions. A vector multiplication might look like "mm_mul_ps" in the code. And inline assembly is really assembler code nested in the code within an "_asm" block. Although both paths have their own rights and might be required to gain optimum performance we won't cover them here in more details. See e.g. Intel's SW Dev Manuals, and Agner Fog's Optimization manuals which are both excellent resources.

Here, we want to switch gears and look onto auto vectorization coming with the Intel compiler. We use latest icc14 beta and build within a chroot environment which already supports Silvermont specific optimizations using the flag -xatom_sse4.2. Using the switch -vec-report<n> we already get some output about successful vectorization:

$ icc -xatom_sse4.2 -O3 -std=c++11 -vec-report3 loop.cpp -o loop
loop.cpp(17): (col. 3) remark: LOOP WAS VECTORIZED
loop.cpp(17): (col. 3) remark: LOOP WAS VECTORIZED

That's good to see: our main loop was vectorized by the compiler. And as we're always sceptical we check in the assembly
Wow, something happened: all in a sudden we do have more than 1000 vector instructions used. What does timing say?

```
$ time ./loop
computation done
0: 7.91431 - 10000000: 7.03673
real 0m2.531s
user 0m2.243s
sys 0m0.095s
```

Hold on, that's remarkable. We used a different compiler and the auto vectorizer. Ok, to be fair, I haven't checked all the switches used on gcc whether they are required for a specific KP purpose and on their impact to performance. But anyways, we have pretty much exactly 5x performance improvement without touching any loc.

It's more than 4x as the compiler doesn't only vectorize but seems to produce more efficient code. And well, the printed values are also identical so we gladly assume the results are fine. But that's not yet all - so far we touched only the vectorization. Let's move on to parallelization.

### How to parallelize

The easiest way to parallelize is to run processes in parallel. However, once the processes do have to communicate it's becoming way harder - IPC would be required (e.g. ulipc). Here it is much easier to work with threads which are kind of lightweight processes and share heap, data, as well as the code. So what we have to do here is to split the workload into chunks and offload it to 4 threads. There is the hard way using native threads (i.e. Posix threads, or pthreads respectively on KP). It's hard because you do have all the thread scheduling, load balancing, synchronization, ... on your own. And it introduces quite some code change. The easier way is to use OpenMP, TBB, or Cilk+ which we will describe in the following.

**OpenMP**

OpenMP adds pragmas to the code which the compiler has to support. I.E. in order to compile an OpenMP SW the compiler needs to support those additional pragmas. Otherwise the code will compile and run just serial. OpenMP is best suited for data parallel tasks but comes also with some pragmas for task parallelism. The pragma can be as easy as just an extra line of code with "#pragma omp parallel" directly prior to the loop:
$cat loop_omp.cpp
#include <iostream>
#include <math.h>
using namespace std;

void ParallelApplyFoo(float a[], size_t n) {
   #pragma omp parallel
   for (int i=0; i<n; i++) {
      a[i] = sqrtf(cosf(sqrtf(sinf(i)*sinf(i)) / cosf(i))*cosf(sqrtf(sinf(i)*sinf(i)) / cosf(i)) + 2.31f)/0.229879834f;
   }
}

In order to compile we need to tell the compiler to use openmp ("-openmp"):

icc -xatom_sse4.2 -O3 -std=c++11 -vec-report3 loop_omp.cpp -openmp -o loop_omp

one short comparison on the vectorization shows a similar number of vectorized loc:

$ objdump -S loop_omp | grep ps | wc
  1560 11850 84042

ok, let's take off:

time ./loop_omp
computation done
0: 7.91431 - 10000000: 7.03673
real 0m5.658s
user 0m14.148s
sys 0m0.142s

What's that? that's about twice as long as for the prior version? Let's start some analysis with VTune (here using the collector coming with VTune Amplifier XE within the CLI_install folder):

$ source /opt/intel/vtune_amplifier_xe/amplxe-vars.sh
Copyright (C) 2009-2013 Intel Corporation. All rights reserved.
Intel(R) VTune(TM) Amplifier XE 2013 (build 290588)
$ amplxe-cl --collect concurrency /ssd/Download/loop_omp
computation done
0: 7.91431 - 10000000: 7.03673
amplxe: Using result path `/ssd/Download/r002cc'
[...]

Vtune tells us that if the total time is 5.6 sec - exactly as we measured and that we integrate over all CPUs we had a total CPU time of 12.4 sec which is pretty bad.
Looking into more detail we can clearly see that about 1.8 sec all 4 cores are running but afterwards the concurrency level fades out. I.E. there is a poor load balancing.
In order to get some more info on the source code we also add the "-g" switch to the compile options and recompile.
Ok, that looks fine: pretty much all time is spent in our loop according to the left side source code view. But how comes we are not at least as fast as the serial vectorized version? Let's look at the assembly on the right size: for sine and cosine functions from "libm_sse2" are used. Let's have a closer look:

```
$objdump -S loop_omp | grep sin
8048e50:       e8 db 05 00 00          call  8049430 <__libm_sse2_sincosf>
8048f6b:       e8 35 21 00 00          call  804b0f0 <__svml_sincosf4>
8049055:       e8 96 20 00 00          call  804b0f0 <__svml_sincosf4>
8049140:       e8 ab 1f 00 00          call  804b0f0 <__svml_sincosf4>
8049430 <__libm_sse2_sincosf>:  
8049465:       0f 8c d4 00 00 00       jl     804953f <__libm_sse2_sincosf+0x10f>
804953a:       e9 b9 00 00 00          jmp    80495f8 <__libm_sse2_sincosf+0x1c8>
804954b:       77 1e                   ja     804956b <__libm_sse2_sincosf+0x13b>
8049566:       e9 8d 00 00 00          jmp    80495f8 <__libm_sse2_sincosf+0x1c8>
8049579:       74 75                   je     80495f0 <__libm_sse2_sincosf+0x1c0>
80495eb:       89 a3 ff ff ff          jmp    8049493 <__libm_sse2_sincosf+0xa63>
804b0f0 <__svml_sincosf4>:  
804b100 <__svml_sincosf4_dispatch_table_init>:  
804b127:       je  804b14d <__svml_sincosf4_dispatch_table_init+0x1c7>
804b147:       eb e3                   jmp    804b152 <__svml_sincosf4_dispatch_table_init+0xc>
804b176:       75 0c                   je     804b2a9 <__svml_sincosf4_n9+0x169>
804b2a7:       75 24                   je     804b303 <__svml_sincosf4_n9+0x1c3>
804b2ee:       7c f5                   jl     804b27e <__svml_sincosf4_n9+0x19a>
804b318:       e8 03 2a 00 00          call   804b31c <__svml_sincosf4_n9+0x19f>
804b320:       e8 03 ff ff ff          call   804b31c <__svml_sincosf4_n9+0x19f>
804b4a0:       ej 96 f8 ff ff          jmp    804b4a5 <__svml_sincosf4_n9+0x165>
804b4c1 <__static_scalar_sincosf>:  
804b4e2:       0f 84 19 01 00 00          je     804b5a6 <__static_scalar_sincosf+0x13a>
804b547:       eb 08                   jmp    804b552 <__static_scalar_sincosf+0x142>
804c02f:       e8 67 ff ff ff          call   804c03a <__static_scalar_sincosf>
804c035:       e8 66 ff ff ff          call   804c03a <__static_scalar_sincosf>
804c03b:       e8 ab ff ff ff          call   804c03a <__static_scalar_sincosf>
804c339 <__svml_sincosf4_p8>:  
 [...]
```

whereas most of the times the highly optimized Intel compiler short vector math library (SVML) is used, our high CPU load hits don't use this library. In contrary the serial code uses SVML all over the place.
That's for sure not a comprehensive analysis but only a hint which might explain the differences (don't want to spend too much time on that now).

In particular OpenMP offers options to adjust the balancing and the vectorization might be a compiler bug of the current compiler. But here it's not the time to proceed with the analysis but move on with Intel Threading Building Blocks (TBB)

TBB

Intel Threading Building Blocks are an open source library developed by Intel. Even in the NGI they are used from TCS for HMI load balancing to increase responsiveness. TBB target C++ STL developers. In contrast to OpenMP TBB uses task parallelism all over. Tasks are scheduled and once a worker thread is ready they are taken and worked on. Data parallel structures like the loop in our example are broken into many tasks by a smart algorithm and scheduled like other tasks. In order to use them we have to modify the source code a bit more. In the example below we use lambda functions which make the code easier readable (for ppl who can read lambda notations - for the rest it will become unreadable ;-)). In general there is a lot documentation on TBB on the web.
$ cat loop_tbb.cpp
#include <iostream>
#include "tbb/tbb.h"
#include "tbb/parallel_for.h"
#include "tbb/task_scheduler_init.h"
using namespace tbb;
using namespace std;

float Foo(float a) {
    return sqrtf(cosf(sqrtf(sin(a)*sin(a)) / cosf(a))) * cosf(sqrtf(sin(a)*sin(a)) / cosf(a)) + 2.31f) / 0.229879834f;
}

void ParallelApplyFoo(float a[], size_t n) {
    parallel_for(size_t(0), n, size_t(1) , [=](size_t i) {a[i]=Foo(float(i));});
}

int main(int argc, char** argv) {
    size_t size = 1000000;
    float* a = new float[size];
    task_scheduler_init init;
    ParallelApplyFoo(a, size);
    cout << "computation done\n"
    cout << a[0] << size << a[size-1] << endl;
    return 0;
}

First of all there are a couple of TBB header files to include. Then in the main function we do have to start the task scheduler. The loop itself is
split into the loop body (here Foo) and the loop control which stays in the function ParallelApplyFoo (for all those who wondered why I chose that
silly name for the loop function you now should know). There are several other ways to write the loop with TBB - some of them might be better
suited for vectorization. I don't want to go into detail here. For the compilation the tbb lib has to be linked in

$ icc -xatom_sse4.2 -O3 -std=c++11 -vec-report3 loop_tbb.cpp -ltbb -o loop_tbb

Unfortunately, the compiler doesn't cope with vectorizing the lambda function (which possibly in a non lambda way the compiler could do -
otherwise blocking within the loop, or Cilk+ compiler vectorization hints could help, or as a least ressort intrinsics within the loop). Also the
compiler doesn't use the SVML transcendentals but only the libm_sse2 ones. Still the code runs in about the time we have for the vectorized
code:

$ objdump -S loop_tbb | grep sin
8049489: e8 d2 0a 00 00 call 8049f60 <__libm_sse2_sincosf>
804964c: e8 0f 09 00 00 call 8049f60 <__libm_sse2_sincosf>
804991b: e8 40 06 00 00 call 8049f60 <__libm_sse2_sincosf>
8049f60 <__libm_sse2_sincosf>:
 8049f95: 0f 8c d4 00 00 j1 804a06f <__libm_sse2_sincosf+0x10f>
 804a06a: e9 b9 00 00 00 jmp 804a128 <__libm_sse2_sincosf+0x1c8>
 804a07b: 77 1e ja 804a09b <__libm_sse2_sincosf+0x13b>
 804a096: e9 8d 00 00 00 jmp 804a128 <__libm_sse2_sincosf+0x1c8>
 804a0a9: 74 75 je 804a120 <__libm_sse2_sincosf+0x1c0>
 804a11b: e9 a3 fe ff ff jmp 8049fc3 <__libm_sse2_sincosf+0x63>
$ time ./loop_tbb
computation done
0: 7.91431 – 10000000: 7.03673
real 0m0.670s
user 0m2.397s
sys 0m0.059s

This is due to the fact that the load balancing with TBB in this example is nearly perfect:
**Top Hotspots**

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

<table>
<thead>
<tr>
<th>Function</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>_ZNbbIO::Internal2Partition::createInterrupt()</td>
<td>2.410s</td>
</tr>
<tr>
<td></td>
<td>0.040s</td>
</tr>
</tbody>
</table>

**Thread Concurrency Histogram**

This histogram represents a breakdown of the Elapsed Time. It visualizes the percentage of the wall time the specific number of threads were running simultaneously. Threads are considered running if they are either actually running on a CPU or are in the runnable state in the OS scheduler. Essentially, Thread Concurrency is a measurement of the number of threads that were not waiting. Thread Concurrency may be higher than CPU usage if threads are in the runnable state and not consuming CPU time.

**CPU Usage Histogram**

This histogram represents a breakdown of the Elapsed Time. It visualizes what percentage of the wall time the specific number of CPUs were running simultaneously. CPU Usage may be higher than the thread concurrency if a thread is executing code on a CPU while it is logically waiting.

**Thread Concurrency Analysis**

<table>
<thead>
<tr>
<th>Function / Call Stack</th>
<th>CPU Time by Utilization</th>
<th>Wait Time by Utilization</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>_ZNbbIO::Internal2Partition::createInterrupt()</td>
<td>2.410s</td>
<td>0.000s</td>
<td>loop_bb</td>
</tr>
<tr>
<td></td>
<td>0.040s</td>
<td>0.000s</td>
<td>loop_bb</td>
</tr>
<tr>
<td>[TBB worker]</td>
<td></td>
<td></td>
<td>loop_bb</td>
</tr>
</tbody>
</table>

**CPU Usage Breakdown**

<table>
<thead>
<tr>
<th>Thread</th>
<th>Running</th>
<th>CPU Time</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBB Worker Thread 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TBB Worker Thread 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TBB Worker Thread 3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**CPU Usage**

- Elapsed Time: 0.701s
- Total Thread Count: 4
- Overhead Time: 0s
- Spin Time: 0s
- CPU Time: 2.460s
- Paused Time: 0s
With that we’ll leave TBB and look for just another means to get parallelized code. And guess what we will see that this way will yield parallelized and vectorized code and the best performance. Please note again: this brief doc is not at all comprehensive. Its intent is NOT to weight one approach against the other.

Cilk+

Cilk+ is a compiler addition currently only supported in its full extend from Intel compiler. Parts of it are already adopted in gcc. Not sure about Microsoft VisualStudio. It offers means for both parallelization and vectorization. Due to the fact that it shares the thread pool and the thread arbitrator with TBB it goes well hand in hand with it. Unfortunately there is not so much information available on Cilk+ yet. But the documentation coming with the Intel compiler is a good starting point.

Note: however, you shouldn’t really mix OpenMP code with TBB, or Cilk+ code (e.g. linking a OpenMP lib into a Cilk+/TBB code). This can easily end in an oversubscription of threads and with that a performance degradation.

Similar to TBB we have to include an extra header file. Apart from that the only change to the serial code is the replacement of for by the cilk language extension specific qualifier cilk_for.

```cpp
$ cat loop_cilk.cpp
#include <iostream>
#include <math.h>
#include <cilk/cilk.h>
using namespace std;

void ParallelApplyFoo(float a[], size_t n) {
  cilk_for (int i=0; i<n; i++) {
    a[i] = sqrtf(cosf(sqrtf(sinf(i)*sinf(i)) /
                  cosf(i)))*sqrtf(sinf(i)*sinf(i)) / cosf(i)) + 2.31f)/0.229879834f;
  }
}

int main(int argc, char** argv) {
  size_t size = 10000000;
  float* a = new float[size];
  ParallelApplyFoo(a, size);
  cout << "computation done\n";
  cout << "0: " << a[0] << " - " << size << ": " << a[size-1] << endl;
  return 0;
}
```

That’s it on the loop. The compiler tells us that the loop was auto-vectorized and our simple count tells us that the number of vector instructions is similar to what we had for the vectorized loop, or for OpenMP respectively.

```
$ icc -xatom_sse4.2 -O3 -std=c++11 -vec-report3 loop_tbb.cpp -ltbb -o loop_tbb
loop_cilk.cpp(9): (col. 32) remark: LOOP WAS VECTORIZED

$ objdump -S loop_cilk | grep ps | wc
 1542 11719 83083
```

Now let’s see what the time says:

```
$ time ./loop_cilk
computation done
0: 7.91431 - 10000000: 7.03673
real 0m0.328s
user 0m1.072s
sys 0m0.074s
```

That’s not bad, is it? Starting from 12.5 sec with pretty much no code modification we end up at 1 sec. I.E. 12.5x performance increase which is pretty close to the theoretical 16x (given that there is also some overhead for threading ...). Agreed, this is an artificial example and you most likely won’t see such speedups in real world scenarios. Still I want to encourage you with this example not to stand still with the given compiler and the given options but to challenge the defaults, to analyze the results and to work on performance optimizations. Besides system tools like
perf, the Intel compiler and the Intel tools can help you a lot on that task. I’d like to end this Wiki page with one last screenshot taken from the Cilk+ example. It shows a nearly perfect load balancing and the fact that there is some overhead for threading.

Have fun writing fast code on VLV.