High Performance Computing

Roofline

Project 3

Johannes Winklehner 1226104

Armin Friedl 1053597

June 24, 2016

A *roofline model* for a multicore-processor is obtained by calcuating the theoretical peak performance of the processor and benchmarking the peak memory bandwith. Two artificial computational kernels with operational intensities of $\frac{1}{16}$ GFLOPs/Byte and 8 GFLOPs/Byte are devised. The performance of the two kernels is then compared to the theoretical calculations in the roofline model.

Contents

1 Introduction

2 Roofline Model

In this section a roofline model [8] will be created for the Intel® Core™ i5-4210U. In Section 2.1 the theoretical floating-point peak performance of the CPU is calculated. Section 2.2 then shows memory bandwidth measurements gathered with NUMA-STREAM [1]. These ingredients are put together into the roofline model which is constructed in Section 2.3.

2.1 Theoretical Peak Performance

The CPU under test was a Intel® Core™ i5-4210U. Table 1 shows the relevant specifications for this processor according to Intel Ark [6].

Specification	Value
# of Cores	2
# of Threads	4
Microarchitecture	Haswell
Max Turbo Frequency	2.7 GHz
Processor Base Frequency	1.7 GHz
Instruction Set Extension	SSE 4.1/4.2, AVX 2.0

Table 1: Relevant processor specifications

According to Intel [3, 5-2 Vol.1] the 4th generation Intel Core processors provide FMA (Fused Multiply-Add) units and AVX (Advanced Vector Extension). Whereas AVX can be the main driver for floating-point peak performance, the peak in this case is mainly determined by the FMA unit.

In general an FMA unit is capable of multiple floating-point (FP) operations during a single cycle. This is directly backed by the hardware (operations are *"fused"* together). Specifically the FMA unit of a Haswell processor is capable of "[...] 256-bit floating-point instructions to perform computation on 256-bit vectors" [3, 5-28 Vol.1].

Since even a DP (double-precision) FP element has only 64-bit, 256-bit would be obviously overprovisioned. But the FMA instructions do not just take scalars as arguments. Instead up to 4 DP FP elements can be packed together in a vector and operations are conducted pairwise. An example mulitply-add instruction is given in [4].

Unfortunately no definite source could be found but according to Shimpi [7] the Haswell architecture is built with 2 FMA units per core. Taking all together we get:

1. Two operations are conducted at once ("fused") and up to four DP FP elements can be packed into the argument vectors. At optimal untilization the FMA unit therefore provides $2 * 4 = 8$ DP FLOPs each cycle.
2. Two cores each with two FMAs can then calculate $2 * 2 * 8 = 32$ DP FLOPs

At maximum turbo frequency the processor therefore has a theoretical peak performance of $32 * 2.7 = 86.4$ GFLOP/s. At base frequency it is capable of $32 * 1.7 = 54.4$ GFLOP/s.

2.2 Memory Bandwidth

To benchmark the memory bandwidth NUMA-STREAM [1] was used. The binary ran on a Fedora 23 system with kernel `4.5.7-200.fc23.x86_64 x86_64` in `multi-user.target` to turn off as many distractors as possible. Compilation was done with gcc and the following options: `-O3 -std=c99 -fopenmp -lnuma -DN=80000000 -DNTIMES=100`.

Again the details of the processor architecture offer a bit of a challenge. The i5-4210U is hyper threaded meaning it provides 4 hardware threads on 2 physical cores. It is not immediately obvious how many threads NUMA-STREAM should be configured with. For this test both configurations[1] were tested and the best one was chosen. The results for NUMA-STREAM configured with two threads are in Listing 1. Prefixes are given in metric scale, i.e. $M = 10^6$ not 2^{20}.The highest achieved rate was 10608 MB/s with the triad function. The triad function is the most demanding kernel of NUMA-STREAM defined at [2] as `a[j] = b[j]+scalar*c[j]`. All other tested configurations had worse results for all 4 kernels although with at most 300 MB/s difference.

```
Function      Rate (MB/s)   Avg time     Min time     Max time
Copy:         9373.3846       0.1368       0.1366       0.1390
Scale:        9414.1304       0.1361       0.1360       0.1381
Add:         10614.6002       0.1812       0.1809       0.1835
Triad:       10607.7910       0.1813       0.1810       0.1834
```

Listing 1: NUMA-STREAM results for two threads

2.3 Graph

The graph of the roofline model is defined by [8]:

```
Attainable GFLOP/s = Min(Peak FLOP,
                         Peak Memory Bandwidth*Operational Intensity)
```

The resulting graph for the values obtained in Section 2.1 and Section 2.2 can be seen in Figure 1.

[1] plus two configurations with 8 and 1 threads respectively for cross checking

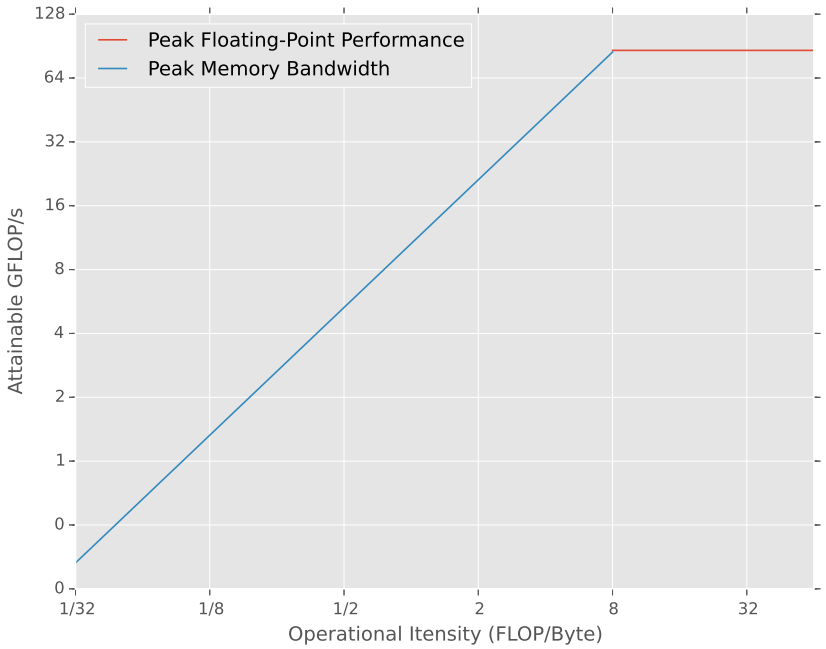

Figure 1: Roofline graph from the values obtained in Section 2.1 and Section 2.2

3 Kernels

Kernels with operational intensity (OI) of $^1/_{16}$ and 8 have been implemented. The kernels are introduced in the following sections.

However the effective operational intensity of a given kernel in a high-level language (as C) is not obvious when compiled to processor instructions. Furthermore, due to today's advanced processor architecture, adaptions had to be made to account for special capabilites. This resulted in several different kernels. Not all of them are machine independent with regard to operational intensity.

All kernels were compiled with `gcc 5.3.1` and different options. The compilation was checked with `objdump -d -M intel-mnemonics`. For a more elaborate analysis of the disassembly on the testers computer, please refer to the header file `aikern.h` that should come with this report. Additionally `Makefile` provides all informations about the used and tested compiler options.

Good results[2] were achieved with `-O2 -mavx -mfma`. But `-O2 -maxv -mfma` is a tradeoff between the best possible results and obviously correct compiled code. In fact the disassembly almost looks like handwritten. If even more optimization is wanted `-O3` can be used. To fully utilize FMA with packed doubles `-Ofast` or `-Ofast -ffast-math` has to be used. Be aware that more optimization than `-O2 -maxv -mfma` results in a very hard to understand disassembly. `-ffast-math` can even introduce rounding errors or reduce the executed FLOPs. It is not completely obvious that the highly optimized compiled still has the wanted operational intensity. `-O0` never works out.

[2] all, including the special FMA kernels, use only expected memory access, doing everything else in registers

Remark: Contrary to popular believe the roofline model is built atop the notion of operational intensity[3] kernels. The differences to arithmetic intensities are outlined in Williams, Waterman, and Patterson [8]. Depending on the definition used these two terms are not necessarily interchangeable. The notion of operational intensity in the following sections might be what some would understand by the term arithmetic itensity.

3.1 1/16 ≠ 1/16. Or: The Fancy Arithmetics of a Compiler

In order to understand why the following kernels are implemented the way they are, an example of a badly behaving $^1/_{16}$ OI kernel is given in Listing 2. The kernel has one FP operation ($*$) and reads 16 bytes (a[i], b[i]) from memory. But in practice this algorithm does not work as expected. There are several ways how one could write the same kernel.

- Submitting `volatile`. This results in the loop being optimized away completely for optimization levels above `-O0`.
- Using no optimization i.e. `-O0`. No advanced features of the processor will be used (e.g., FMA requires at least `-O2`). Also just about everything is read and written from and to the stack. Even loop variables. One may now assume that this is cached anyway — or one ain't so.
- Using `volatile` and optimization. When volatile is used gcc reads and writes variable `tmp` from and to the stack, even in `-O3`. If tmp is cached or not is hard to predict. It's not improbable but relying on that assumption can yield wrong results.
- Using `register`, `volatile` and optimization. Unfortunately `register` just *advises* the compiler to use a register. It does not force the compiler to do so. Seemingly `volatile` overrules `register` in this case – `tmp` is read and written from and to the stack. Again assuming any caching behaviour is adventurous at least.

In the worst case found (no optimization, no volatile, no register) this results in reads of 16 bytes (a[i],b[i]) plus 8 bytes (i), and writes of 16 bytes (i, tmp assignment). Making no caching assumptions this results in an effective operational intensity of $^1/_{40}$ for a superficial $^1/_{16}$ OI kernel. For more complex kernels the results get even worse. A triad `t=a*b+c` will store easy-to-miss intermediate results on the stack if no special care is taken.

To prevent this, one could write assembly directly or rely on compiler intrinsics. The kernels in this report though consist just of normal C code which was hand-crafted until an acceptable compilation was reached. The generated machine code was disassembled and manually checked for hidden memory access. The results are therefore compiler and machine specific, but should be quite generalizable for the most part.

```
1 volatile register double tmp = 0.1;
2 for(size_t i=0; i<size; i++)
3     tmp = a[i] * b[i];
```

Listing 2: Simple $^1/_{16}$ kernel with questionable compiled form

[3] FLOPs against bytes written to DRAM

3.2 The 1/16 OI Kernel

Two $^1/_{16}$ kernels have been implemented. The kernel in Listing 3 is a standard kernel which does not assume special processor capabilities. The second kernel in Listing 4 however is designed to make use of a processor's FMA unit.

The simple kernel in Listing 3 reads 8 bytes (a[i]) once for both operands of $*$ and writes 8 bytes (again to a[i]). This results in 16 byte operations. Only one FP instruction is executed, namely $*$. At `-O2` the loop variable is held in a register. This results in an $^1/_{16}$ OI kernel.

```
#pragma omp parallel for
for(size_t i=0; i<size; i++)
  a[i] = a[i] * a[i];
```

Listing 3: Simple $^1/_{16}$ OI kernel

The FMA aware kernel in Listing 4 is a bit more involved. First a triad operation is used ($*$ and $+$ operations have to be balanced). This results in 2 FP instructions executed per round. $3 * 8 = 24$ bytes have to be read (a[i], b[i], c[i]) and 8 bytes have to be written (a[i]), in sum 32 byte operations. This results in an $^2/_{32} = {}^1/_{16}$ OI kernel. The loop variable is again held in a register.

Be aware that the FMA kernel *cannot* be used on a non-FMA processor. For the FMA aware kernel to work correctly it is important that (i) the processor has an FMA unit (ii) the `aikern.c` library is compiled with at least `-O2 -mavx -mfma` (iii) the compiled binary really makes use an FMA instruction (such as `vfmadd132sd` [5] or even `vfmadd132pd` [4] on the testers machine). Otherwise the results are meaningless due to write-outs of intermediary values.

Also note that in order to use the full capabilities of Intel's FMA the doubles must be packed. This happens if `-Ofast` is given to `gcc` in addition. However this also triggers other optimizations such that the disassembly gets long and complex. It is not immediately obvious that the generated disassembly is correct. But no instructions could be found that do not solely use registers, except loading and storing data from and to the arrays – just as wanted.

```
#pragma omp parallel for
for(size_t i=0; i<size; i++)
  a[i] = a[i] * b[i] + c[i];
```

Listing 4: FMA aware $^1/_{16}$ OI kernel

3.3 The 8 OI Kernel

In this section the implemented 8 OI kernels are shown. Listing 6 is a simple 8 OI kernel which should work on any processor. The kernel in Listing 7 is tailored for processors with an FMA unit. For the kernels macros were used to repeat the floating point instructions. In some sense this behaves like a huge loop unrolling. Some of the used repeating macros are shown in Listing 5.

```
#define REP0(X)
#define REP1(X)    X
#define REP2(X)    REP1(X)   REP1(X)
#define REP3(X)    REP2(X)   REP1(X)
//[...]
#define REP9(X)    REP8(X)   REP1(X)
#define REP10(X)   REP9(X)   REP1(X)
#define REP20(X)   REP10(X)  REP10(X)
//[...]
#define REP100(X)  REP50(X)  REP50(X)
```

Listing 5: Macros for bulk repeating instructions

The simple kernel in Listing 6 reads 8 bytes (a[i]) and writes 8 bytes (a[i]) while performing 128 FLOPs in total. Therefore this represents a $^{128}/_{16} = {}^{8}/_{1}$ OI kernel.

```
#pragma omp parallel for
for(size_t i=0; i<size; i++){
  a[i] = REP100(a[i]*)
         REP20(a[i]*)
         REP8(a[i]*)
         REP1(a[i]);
}
```

Listing 6: Simple 8 OI kernel

For the most part things mentioned already in Section 3.2 hold true for the 8 OI FMA aware kernel too. Please refer to Section 3.2 for more detailed information about the rationale behind. Compiling this with `-O2 -mavx -mfma` yields an obviously correct result. However if one wants to make use of packed doubles `-Ofast` has to be used which optimizes the code further so that the disassembly is hard to grasp. Anyway it seems that with `-Ofast` at least no malicious read/writes are introduced.

The FMA aware kernel in Listing 7 reads 8 bytes (a[i]) and writes 8 bytes (a[i] but only once per iteration), totalling 16 bytes. Please keep in mind that intermediate a[i] are not written back but instead (at least with `-O2` or better) held in a register. There is only one `vmovsd` instruction for writing the value back in each iteration. The kernel executes $64 * 2 = 128$ FLOPs. Therefore this is a $^{128}/_{16} = {}^{8}/_{1}$ OI kernel.

```
#pragma omp parallel for
for(size_t i=0; i<size; i++){
  REP60(a[i] = a[i] * a[i] + a[i];)
  REP4(a[i] = a[i] * a[i] + a[i];)
}
```

Listing 7: FMA aware 8 OI kernel

3.3.1 Some Further 8/1 Kernel

Since some effort was put in getting results near peak performance `-Ofast -ffast-math` was used to stretch compiler optimization to the maximum. Unfortunately `-ffast-math` does not preserve strict IEEE compliance. It is therefore allowed to ignore non-associativity of floating point operations. For example $x = x * x * x * x * x * x * x * x$ can be optimized to $x *= x; x *= x; x *= x;$. Clearly this has an effect on the OI of the kernel. To test fastmath the kernel in Listing 8 was introduced. Mind that a[i] is written out only once and held in registers during a single iteration.

```
#pragma omp parallel for
for(size_t i=0; i<size; i++){
    REP100(a[i]=a[i]*a[i];);
    REP20(a[i]=a[i]*a[i];);
    REP8(a[i]=a[i]*a[i];);
  }
```

Listing 8: FMA aware 8 OI kernel with fastmath correctness

Since the results were still not satisfying another kernel 8/1 OI kernel which makes use of handcrafted compiler intrinsics was introduced too. This kernel makes full use of the 256-bit-packed-doubles fused-multiply-add floating-point operation the FMA unit of the processor provides. The kernel can be seen in Listing 9. At least in theory this should yield peak performance. The disassembly under full optimization (options can be seen in `Makefile`) behaves very much like handwritten assembly.

```
#pragma omp parallel for
for(size_t i=0; i<(size-4); i+=4){
  // pack doubles
  __m256d packvec = _mm256_set_pd(a[i], a[i+1], a[i+2], a[i+3]);

  REP60(packvec = _mm256_fmadd_pd(packvec, packvec, packvec););
  REP4(packvec = _mm256_fmadd_pd(packvec, packvec, packvec););

  a[i] = packvec[0];
  a[i+1] = packvec[1];
  a[i+2] = packvec[2];
  a[i+3] = packvec[3];
}
```

Listing 9: FMA aware 8 OI kernel with intrinsics

4 Results

The best results for various kernels are given in Table 2. The optimization binary `roofline_full_manpack` was used for these results. This is the binary with all optimizations and the intrinsics kernel

enabled. The following parameters were used: `roofline_full_manpack -s 150000000 -r 5`. One double array was therefore 1144.41 MB big – clearly too big for the cache.

Note how `simple8` is clearly flawed with `-ffast-math` enabled. This is due to the non IEEE compliant optimization as described in Section 3.3.1. At this level of optimization only `simple8fastmath` (which is fastmath safe but flawed with lower optimization levels) should be considered as a *replacement* of `simple8`.

The simple* kernel are those kernels that do not make use of FMA but can be safely used with processors without an FMA unit. fma* kernels on the other hand are those that should make use of FMA. simple8fastmath is a simple8 that can be safely used with `-ffast-math` optimization. And fma8manpack is the kernel which uses intrinsics to ensure that is solely operates with FMA instructions on packed floats.

Kernel	Max. GFLOP/s
simple16	0.9919
fma16	0.9891
simple8	123.4004
simple8fastmath	8.7187
fma8	21.7866
fma8manpack	18.9066

Table 2: Results for various kernels

The rooftop graph with the best runs of the 2 best kernels of each category (`simple16` and `fma8`) is depictured in Figure 2.

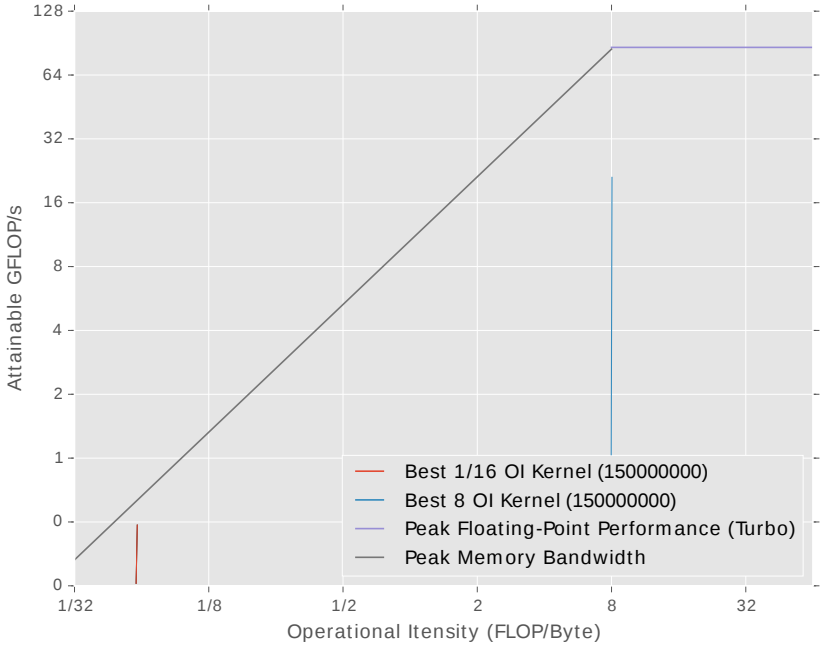

Figure 2: Roofline graph with kernel results

Best results for an input size of 100000000 are given in Table 3 and Figure 3.

Kernel	Max. GFLOP/s
fma16	0.9816
fma8	21.8837

Table 3: Best results for 100000000

Best results for an input size of 250000000 are given in Table 4 and Figure 4.

Kernel	Max. GFLOP/s
simple16	1.0476
fma8	21.4297

Table 4: Best results for 250000000

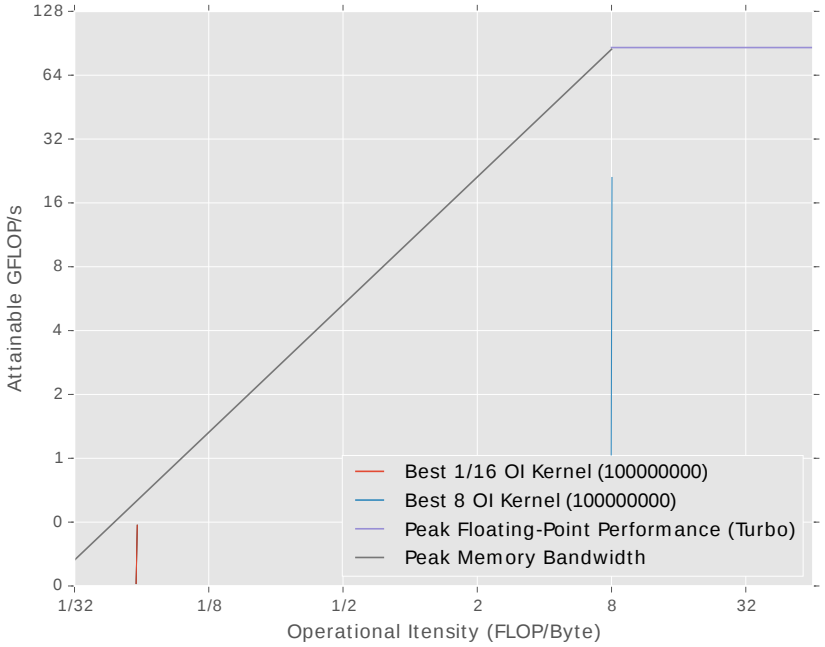

Figure 3: Roofline graph with best results for 100000000

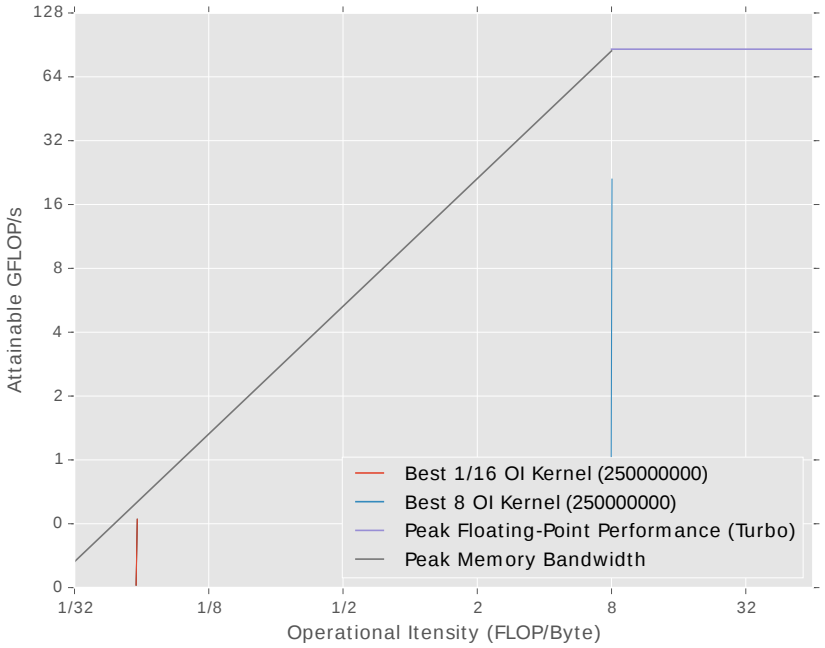

Figure 4: Roofline graph with best results for 250000000

5 Discussion

The results are not really unexpected. 22 GFLOP/s for a consumer grade CPU are quite good. That the peak performance wasn't reached can have many reasons. One might be that the binary had to share it's CPU time with other things running on the same machine etc.

It bears some curiosity that the `fma8` kernel is faster than the `fma8manpack` kernel. But there's more to this story. The `fma8` kernel performs much worse (1/3) than the `fma8manpack` at other optimization levels than `-O3`. Taking a look at the disassembly reviels that `fma8` uses some weird combination of packed and unpacked FMA instructions. It is not even a hundred percent sure that the optimization didn't tinker with the effective OI of the `fma8` kernel.

One baseline of this report is that a superficial OI of a kernel in a high-level language does not really resembles the OI of the kernel when compiled to the machine.

Please also refer to the header file `aikern.h` for a more technical analysis of the disassembly of various kernels. The `Makefile` generates many different variations of the kernels to play with. The `log/` folder will contain details about time an GFLOP of every kernel when a binary is run.

References

[1] Lars Bergstrom. *NUMA-STREAM*. URL: https://github.com/larsbergstrom/NUMA-STREAM (visited on 06/20/2016).

[2] Lars Bergstrom. *stream.c*. URL: https://github.com/larsbergstrom/NUMA-STREAM/blob/e5aa9ca4a77623ff6f1c2d5daa7995565b944506/stream.c#L286 (visited on 06/20/2016).

[3] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B, 3C and 3D*. Intel. Apr. 2016. URL: https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[4] Intel. *Intel Intrinsics Guide: vfmadd132pd*. URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=AVX2,FMA&text=vfmadd132pd&expand=2365 (visited on 06/19/2016).

[5] Intel. *Intel Intrinsics Guide: vfmadd132sd*. URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=AVX2,FMA&text=vfmadd132sd&expand=2365,2403 (visited on 06/19/2016).

[6] Intel Ark. *Intel® Core™ i5-4210U Processor Specifications*. URL: http://ark.intel.com/products/81016/ (visited on 06/19/2016).

[7] Anand Lal Shimpi. *Haswell's Wide Execution Engine*. Oct. 5, 2012. URL: http://www.anandtech.com/show/6355/intels-haswell-architecture/8 (visited on 06/19/2016).

[8] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: an insightful visual performance model for multicore architectures". In: *Communications of the ACM* 52.4 (2009), pages 65–76.