REPORT ON OpenDreamKit DELIVERABLE D5.5

Write an assembly superoptimiser supporting AVX and upcoming Intel processor extensions for the MPIR library and optimise MPIR for modern processors.

WILLIAM HART

Due on	02/28/2017 (Month 18)
Delivered on	02/27/2017
Lead	University of Kaiserslautern (UNIKL)
Progress on and finalization of this deliverable has been tracked publicly at: `https://github.com/OpenDreamKit/OpenDreamKit/issues/118`	

CONTENTS

DELIVERABLE DESCRIPTION, AS TAKEN FROM GITHUB ISSUE #118 ON 2017-02-27

- **WP5:** High Performance Mathematical Computing
- **Lead Institution:** University of Kaiserslautern
- **Due:** 2017-02-28 (month 18)
- **Nature:** Demonstrator
- **Task:** T5.5 (#103): MPIR
- **Proposal:** P. 52
- **Final report**

1. CONTEXT AND PROBLEM STATEMENT

`MPIR` is a highly optimised library for bignum arithmetic forked from GMP. It is a fundamental building block for many open source mathematical computational components (SageMath, FLINT, Nemo, Eiffelroom, GMPY, Advanpix, PHP and MPIR.net), and therefore its fine optimization on a variety of processor architecture is important for the High Performance aims of OpenDreamKit.

For this deliverable the task was to implement a superoptimizer which tries valid permutations (i.e., that do not change program behaviour) of instructions in assembly functions, times each permutation, and chooses the fastest one. In addition, new MPIR functions for recent processor

architectures were to be written, making use of recently added features like AVX2 instructions, and be optimized with the super-optimizer where applicable.

It is usually the case that the difference between assembly optimised code and C code compiled by an optimising compiler such as GCC is a factor of 4-12 for bignum arithmetic. But each new processor microarchitecture requires new assembly language code to be written. One can use older assembly code, but each new microarchitecture can do around 20% better than the previous one if hand optimisation is done. In addition to that, speedups due to superoptimisation can be anywhere from 5% to 100%.

In MPIR, we are typically comparing superoptimised code that was written for a previous, but related microarchitecture, and so if the job is done properly, we expect about 20% improvement. We see that, and more, below.

2. Work completed

For the first six months of the project, we wrote the `ajs` superoptimizer (`https://github.com/akruppa/ajs`), based on the open-source `AsmJit` library (`https://github.com/asmjit/asmjit`), a complete Just In Time and remote assembler for C++ language.

For the second six months, we solved several problems with the `ajs` superoptimizer, especially erratic timings that had put the concept in jeopardy, and, with contributions from Jens Nurmann, wrote and/or optimized a set of core functions for MPIR and some auxiliary functions used internally (see below).

2.1. The `ajs` superoptimizer

The biggest problem with the superoptimizer was the highly erratic timings it measured for function executions. This made it practically impossible to have it automatically choose (one of) the fastest permutations for a given function.

The major problem was that the RDTSC(P) instructions no longer count cpu core cycles, but cycles of a fixed-frequency counter, i.e., elapsed natural time. Due to extensive clock scaling features of recent cpus, the measured time depended much more on power saving decisions made by the cpu than on the (comparatively small) speedup by finding a good permutation. This is especially true as functions may have to be superoptimized in several pieces, e.g., separately for lead-in, core loop, and lead-out, to reduce the search space so that decent permutations are found within acceptable time.

The solution we used was the RDPMC instruction which provides low-latency access to performance measurement counters, including the "second fixed- function counter" (FFC2) which does, in fact, count cpu core clock cycles. The problem was enabling access to this counter from user mode applications, which requires setting some bits in MSR/CR. Attempts to do so via kernel modules we wrote turned out unreliable as the kernel disabled the bits again (and my modules killed machines on multiple occasions).

Eventually an excellent solution to this problem was found in the jevents library of the pmu-tools (`https://github.com/andikleen/pmu-tools/`) which provides an API to the perf subsystem of the Linux kernel. This allows enabling RDPMC to read FFC2 without the kernel spuriously disabling it again.

The resulting timings within one program run were much more stable than before, usually resulting in the same cycle count for a given function. The timings still vary by 1 occasionally (very rarely 2 or more); we have tried to find the source of the remaining variance, but to no avail.

Another major source of error, but invariant within one super-optimizer run, was the alignment of the stack, which appears to be randomly chosen at program start. The writes to the stack (PUSH/CALL) on a function call could alias (mod 4096) with the measured function's input

operands, causing "partial address alias" stalls which inflated execution time by as much as 10 cycles. This problem was solved by forcing a particular stack alignment.

Other problems that occurred within `ajs` and which were solved:

- Jump instruction were always encoded in long form by `asmjit`, changing instruction alignment compared to other assemblers. We now manually annotate those instructions that require long form; all other use short. This requires manual work to annotate and verify the resulting instruction encodings.
- Allow new registers introduced with AVX2, and instructions with 4 operands
- Various fixes and extensions to asm parsing code

All in all, fixing the aforementioned problems in `ajs` consumed well over 2 months of time on the project. The code to generate permutations that honour data dependencies is quite powerful; however, subtle interactions with the cpu hardware made it very time-consuming to get nearly cycle-accurate timings as we required.

2.2. **Optimized functions for MPIR**

We now review the functions that have been optimized on various processor microarchitectures (Intel Haswell and Skylake and AMD Bulldozer).

Whilst these aren't the most recent architectures from the major chip manufacturers, they are coming into widespread use around now. Indeed it is difficult to get access to more recent machines. Naturally access to the particular architecture is required in order to optimise for it.

For Haswell and Skylake, the following set of core functions was re-implemented or existing code optimized to take advantage of the respective micro-architecture: `add_n`, `sub_n`, `addmul_1`, `submul_1`, `addlsh1_n`, `sublsh1_n`, `com_n`, `copyi`, `copyd`, `rshift1`, `lshift1`, `rshift`, `lshift`, `mul_1`, `mul_basecase`.

The only AMD CPU to which we could gain access was a Bulldozer which is a fairly old and poorly designed microarchitecture; in particular, new instruction set extensions like AVX2 are so slow on Bulldozer (and Piledriver) that they are best avoided. This left little room for optimization, and we opted not to write new code for this outdated cpu, but to cherry-pick existing code that performs well.

We are very grateful to Jens Nurmann who contributed significant amounts of code and expertise on AVX2 programming, to Brian Gladman for porting the new code to the Microsoft Visual C build system, and to William Stein for granting us access to a Bulldozer machine.

2.2.1. *Haswell microarchitecture.*

For Haswell, new AVX2 versions of `com_n`, `copyd`, `copyi`, `lshift`, `lshift1`, `rshift`, `rshift1` were written anew and super-optimized.

The `addmul_1`, `submul_1`, `mul_1`, `mul_basecase`, and `sqr_basecase` functions for Haswell in the GMP library were copied as these are extremely well optimized already - we did not think we could produce better in what little time we had left. Attempts to super-optimize these functions did not find better code.

Existing `add_n`, `sub_n`, `karaadd`, `karasub`, `hgcd2` functions were modified for Haswell and super-optimized, while `sumdiff_n` and `nsumdiff_n` were written anew.

To give a summary of the speedups obtained, we include here results obtained with the `mpir_bench` program (`https://github.com/akruppa/mpir_bench_two`). Higher values are better (function executions per unit time); the apparent slow-down for size < 512 GCD is to be investigated.

Program multiply (weight 1.00)	Old	New
128 128	108222650	107111633
512 512	22816149	26895874
8192 8192	228124	289984

Program multiply (weight 1.00)	Old	New
131072 131072	3884	5015
2097152 2097152	173	203
128 128	108109328	107223557
512 512	17689648	20384648
8192 8192	155145	189057
131072 131072	2771	3479
2097152 2097152	118	133
15000 10000	80120	91788
20000 10000	61030	71776
30000 10000	37966	42448
16777216 512	501	658
16777216 262144	24.6	28.7

Program gcd (weight 0.50)	Old	New
128 128	3729465	3646816
512 512	767983	554155
8192 8192	10974	15908
131072 131072	175	223
1048576 1048576	9.38	11.5

Program gcdext (weight 0.50)	Old	New
128 128	2628011	2036197
512 512	595026	451973
8192 8192	7900	11192
131072 131072	129	171
1048576 1048576	6.04	7.94

The new code can be found in the directory `https://github.com/akruppa/mpir/tree/master/mpn/x86_64/haswell`.

2.2.2. *Skylake microarchitecture.* For Skylake, `add_n`, `sub_n`, `mul_1`, `add_err1_n` and `sub_err1_n` were written anew and super-optimized. The `addmul_1`, `mul_basecase` and `sqr_basecase` functions were taken from GMP. The other functions for Haswell are used as fall-backs.

Program multiply (weight 1.00)	Old	New
128 128	123326551	123312872
512 512	29477397	33899135
8192 8192	298474	358841
131072 131072	4924	6024
2097152 2097152	213	246
128 128	123340235	123340948
512 512	22551903	25322713
8192 8192	208058	238204
131072 131072	3497	4316
2097152 2097152	142	155

Program multiply (weight 1.00)	Old	New
15000 10000	104503	112647
20000 10000	80121	89101
30000 10000	47871	54247
16777216 512	611	693
16777216 262144	29.1	33.6

Program gcd (weight 0.50)	Old	New
128 128	4387356	4373122
512 512	814864	682194
8192 8192	11468	18970
131072 131072	208	274
1048576 1048576	11.3	14.1

Program gcdext (weight 0.50)	Old	New
128 128	2750101	2562046
512 512	640358	557060
8192 8192	8526	13743
131072 131072	155	212
1048576 1048576	7.50	9.83

The new code can be found in the directory `https://github.com/akruppa/mpir/tree/master/mpn/x86_64/skylake`.

2.2.3. *Bulldozer microarchitecture.* On Bulldozer, the speed gains obtained are much more humble than on Haswell and Skylake, as relatively few functions were replaced by faster ones. This microarchitecture is not a profitable target for code optimization any more.

Program multiply (weight 1.00)	Old	New
128 128	55322152	55550756
512 512	12248577	12586138
8192 8192	139406	138848
131072 131072	2406	2421
2097152 2097152	101	105
128 128	55781257	51370568
512 512	7690668	8710261
8192 8192	90386	83592
131072 131072	1587	1584
2097152 2097152	64.0	65.9
15000 10000	44703	45193
20000 10000	33852	35294
30000 10000	20000	20199
16777216 512	268	294
16777216 262144	12.7	13.4

Program gcd (weight 0.50)	Old	New
128 128	2597029	2611829
512 512	284031	289573
8192 8192	6800	6810
131072 131072	108	107
1048576 1048576	5.77	5.77

Program gcdext (weight 0.50)	Old	New
128 128	1270472	1239850
512 512	223972	218197
8192 8192	4944	4924
131072 131072	78.1	78.0
1048576 1048576	3.65	3.65

The new code can be found in the directory `https://github.com/akruppa/mpir/tree/master/mpn/x86_64/bulldozer`.

2.3. **Additional work**

Since the end of the project, we have added preliminary Broadwell CPU support. This does not include any superoptimisation at this point. Broadwell is essentially a revision of Haswell, but with some Skylake features. We have added processor detection to MPIR and sped up this CPU by making use of the Haswell code written for this project. Work is underway to make some of the new Skylake code available to Broadwell chips, and to write new assembly code for Broadwell. Many thanks to our volunteers, Jens Nurmann and David Cleaver who have agreed to work on this.

3. FUTURE WORK

The superoptimizer works reasonably reliably now and can be used to optimize more functions in MPIR and other software projects. At this stage MPIR is the only project that has made use of the superoptimiser, however we have already received a support request, so we expect there to be more use cases soon.

The division and GCD functions in MPIR are worthwhile targets for additional optimization work.

The new Zen microarchitecture of AMD was released towards the end of our project, and looks promising for scientific computation. An optimization effort here would be worthwhile; it will require to get access to such a machine.

4. SOURCE CODE

The `ajs` superoptimizer can be found at `https://github.com/akruppa/ajs`. The optimized functions for `MPIR` are merged into the main `MPIR` repository at `https://github.com/wbhart/mpir`.

5. TESTING THIS CODE

Build instructions for MPIR are as follows:

Download MPIR-3.0.0 from `http://mpir.org/`

Note that you also need to have the latest yasm to build MPIR: `http://yasm.tortall.net/`

To build yasm, download the tarball:

```
./configure
make
```

To test MPIR, download the tarball:

```
./configure --enable-gmpcompat --with-yasm=/path_to_yasm/yasm
make
make check
```

A Haswell, Skylake, or Bulldozer CPU is required to test the changes referred to above.

6. BLOG POST

I have blogged about this project at `https://wbhart.blogspot.de/2017/02/assembly-super` `html`.