REPORT ON OpenDreamKit DELIVERABLE D5.7

Take advantage of multiple cores in the matrix Fourier Algorithm component of the FFT for integer and polynomial arithmetic,and include assembly primitives for SIMD processor instructions (AVX, Knight's Bridge, etc.), especially in the FFT butterflies

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/120`	

CONTENTS

Deliverable description, as taken from Github issue #120 on 2017-02-28 1
1. Report on parallelising the FFT 1
1.1. Problem statement 1
1.2. The method 2
1.3. Results 3
1.4. Testing the parallel FFT 3
2. Report on writing assembly primitives for the FFT butterflies 3
2.1. Problem statement 3
2.2. Results 4
2.3. Blog post 6

DELIVERABLE DESCRIPTION, AS TAKEN FROM GITHUB ISSUE #120 ON 2017-02-28

- **WP5:** High Performance Mathematical Computing
- **Lead Institution:** University of Kaiserslautern
- **Due:** 2017-02-28 (month 18)
- **Nature:** Demonstrator
- **Task:** T5.4 (#102): Singular, T5.5 (#103): MPIR
- **Proposal:** P. 52
- **Final report**

1. REPORT ON PARALLELISING THE FFT

1.1. Problem statement

Given two polynomials of length n, the time to multiply them using classical schoolboy multiplication is O(n^2). But there are numerous algorithms which can do better. The Karatsuba method already takes time O(n^log_2(3)). There are other methods, including Toom-Cook which slightly improve the exponent.

The Fast Fourier technique allows multiplication of such polynomials in O(n log(n)) operations. This is a technique that goes back as far as Gauß, but has seen extensive development since then, with over 800 papers on the method and related techniques, with applications from signal processing to string search or polynomial and integer arithmetic.

The version of the FFT that is used in `Flint` and `MPIR` is the Schoenhage-Strassen method. Instead of doing a convolution over the complex numbers, which would make use of imprecise floating point numbers, which would be subject to rounding error, it makes use of an exact ring, namely Z/pZ where p = 2^(2^n) + 1. This technique allows exact multiplication of polynomials and integers with nearly linear complexity.

In summary, the existing FFT in `Flint` is used for:

- Large integer multiplication
- Schoenhage-Strassen univariate polynomial multiplication
- Kronecker-Segmentation univariate polynomial multiplication

The purpose of this task was to parallelise the FFT in `Flint`.

Typically, parallelising the FFT algorithm is difficult. However, `Flint` makes use of a cache-friendly implementation of the FFT which uses the Matrix Fourier Algorithm. This breaks one very large FFT convolution up into many smaller FFT's.

The existing FFT implementation in `Flint` (and `MPIR`) is world class and includes:

- truncated fourier transform
- use of low level GMP/MPIR assembly optimised functions
- square root of 2 trick
- Matrix Fourier Algorithm
- Nussbaumer convolution
- Chinese remainder with naive convolution

1.2. The method

In order to thread the FFT in `Flint`, we used `OpenMP`. The level at which we threaded it was at the level of the Matrix Fourier Algorithm. This involved separating temporary storage that is used throughout the algorithm, on a per thread level, and then adding `OpenMP` primitives to the part of the Matrix Fourier Algorithm that breaks the FFT into lots of smaller FFTs.

We also threaded the code which splits large integers into FFT coefficients. Unfortunately it is difficult, or even impossible to fully parallelise the recombination that happens after the FFT convolution has run, so this wasn't attempted. However, it is a negligible portion of the run time.

Fortunately, once the Matrix Fourier Algorithm becomes more efficient than a single large FFT (due to its cache aware properties), the threaded version also becomes more efficient than the single-threaded version. In fact, the tuning crossover was found to be at exactly the same point! This is an interesting coincidence and made tuning very easy.

To maximise the benefit of threads, we combine parts of the small inward FFTs, the relevant pointwise multiplications and parts of the outward inverse FFTs into combined blocks that each run on a single thread without interruption. The whole FFT convolution consists of many of these smaller blocks. This was by design rather than accident!

The algorithm in `Flint` also combines the truncated Fourier transform and Matrix Fourier algorithm in such a way that the entire large FFT breaks down exactly into the smaller threaded blocks discussed above, with no additional bits that have to be dealt with serially. This is due to an innovation in the `Flint` FFT which isn't available elsewhere. Again, this was a design feature, not an accident. The scope of this method is exceptionally technical and well beyond the scope of this report to describe.

In fact, we were able to preserve every single one of the technical tricks mentioned above in our parallel implementation of the FFT in `Flint`.

1.3. Results

The new code for the threaded Matrix Fourier algorithm has been implemented as part of this deliverable and merged into the main `Flint` repository.

Here are timings of the new code in `Flint` on a single core, versus four and eight cores for various sized integer multiplications on a 64 bit machine.

limbs	1 core	4 core	8 core
114525	0.066s	0.049s	0.049s
229725	0.14s	0.11s	0.11s
360237	0.32s	0.12s	0.09s
721709	0.65s	0.25s	0.19s
1245101	1.14s	0.39s	0.27s
2492333	2.33s	0.81s	0.55s
4587132	4.45s	1.52s	1.02s
9178748	9.07s	3.02s	2.06s
25947772	28.1s	9.35s	6.25s
51908220	57.9s	24.0s	13.8s
118997068	143s	48.4s	33.2s
238026828	309s	105s	65.7s
506425420	801s	241s	146s

1.4. Testing the parallel FFT

The Flint repository is available here.

To build and test the code mentioned above, you must have `GMP`/`MPIR` and `MPFR` installed on your machine (refer to your system documentation for how to do this). Then do:

```
git clone https://github.com/wbhart/flint2
cd flint
./configure --with-mpir=/path/to/mpir \
            --with-mpfr=/path/to/mpfr \
            --enable-openmp
export OMP_NUM_THREADS=4
make
make check MOD=fft
```

Full instructions on how to build `Flint` are available in the `Flint` documentation, available at the `Flint` website.

The description of the FFT interface is well beyond the scope of this documentation, but can be found in the `Flint` documentation (625 pp.) There is also additional information specific to the FFT in the Flint FFT README

2. Report on writing assembly primitives for the FFT butterflies

2.1. Problem statement

For this deliverable, our task was to improve existing functions or write new ones to use features of recent microprocessors (esp. AVX2) to speed up the Schönhage-Strassen FFT butterflies. Such assembly primitives are provided by the `MPIR` library.

The main operations used in the FFT butterflies are:

- Compute a+b, a-b for given a,b

- Compute -(a+b), a-b for given a,b
- Bit-wise shifts by varying bit-counts
- Subtraction, and to a lesser extent addition and negation

Some of these operations already had assembly primitives available as part of the `MPIR` library. However, these were not optimised for recent architectures using AVX, for example. In this task, we also added a new assembly primitive, as described below, which is used directly in the FFT butterflies (where most of the FFT work is actually done).

Each year or two, Intel and AMD release new CPU microarchitectures. The ones we focused on for this deliverable were Intel Haswell and Skylake and AMD Bulldozer. These are not the most recent architectures, but they are coming into widespread use at the present time.

2.2. **Results**

The microarchitectures for which we optimized the code are mainly Intel Haswell and Intel Skylake, and to a lesser extent AMD Bulldozer. For Bulldozer (and Piledriver) it should be noted that the opportunities

for optimization are rather limited: the microarchitecture generally performs poorly, especially in hyper-threading mode, and the AVX instructions in particular are so slow as to be practically useless. The newer AMD Steamroller fares better, but we did not have access to one.

For Haswell and Skylake, the `mpn_lshift1`, `mpn_rshift1`, `mpn_lshift`, and `mpn_rshift` have been written anew, using AVX2 instructions which gave a large speed-up over the previous code. The `mpn_add_n`/`mpn_sub_n` functions (which are identical, performance-wise) have been modified from existing code and optimized according to the respective micro-architecture. An `mpn_sumdiff_n` (computes a+b, a-b) has been introduced into `MPIR`; this function existed for older processors but not for recent x86_64.

We are very grateful to Jens Nurmann who contributed significant amounts of code and expertise on AVX2 programming.

2.2.1. *Haswell microarchitecture.* Timings in cycles per limb:

Function	Old	New
mpn_lshift1	1.11	0.564
mpn_rshift1	1.39	0.589
mpn_lshift	1.85	0.568
mpn_rshift	1.40	0.578
mpn_add_n	1.32	1.11
mpn_sumdiff_n	2.62(1)	2.42
mpn_nsumdiff_n	3.23(2)	2.64

(1) The sum of the times of `mpn_add_n`, `mpn_sub_n`.

(2) The sum of the times of `mpn_add_n`, `mpn_sub_n`, `mpn_neg_n`.

Timings for the full Schönhage-Strassen large integer multiplication (`mpn_mul_n`) in seconds:

Limbs	Old	New	Ratio
10000	0.002399728	0.002171788	0.91
100000	0.026374851	0.022960783	0.87
1000000	0.357847841	0.302023203	0.84

Note that these timings include the effect of code improvements made for D5.5 (#118), in particular, better `mpn_mul_basecase` and Karatsuba code.

2.2.2. *Skylake microarchitecture.* Timings in cycles per limb:

Function	Old	New
mpn_lshift1	1.01	0.601
mpn_rshift1	1.52	0.601
mpn_lshift	2.01	0.608
mpn_rshift	1.52	0.606
mpn_add_n	1.22	1.04
mpn_sumdiff_n	2.44(1)	2.04
mpn_nsumdiff_n	3.06(2)	2.32

Of note here is the speed of mpn_add_n/mpn_sub_n, at essentially 1c/l for the core loop, optimal both in terms of the data dependency chain and memory accesses, as Skylake can in theory execute 2 read and 1 write per clock cycle. In practice, presumably the instruction scheduler falls into a bad pattern after running at 1c/l for a while, and from then on runs the loop only at ˜1.2c/l. Jens Nurmann found that inserting a meaningless AVX2 instruction into the core loop (which does not otherwise use AVX2)

breaks up this bad scheduling pattern, allowing these critically important core functions to run at the optimal speed reliably.

Timings for mpn_mul_n in seconds:

Limbs	Old	New	Ratio
10000	0.002125143	0.001711500	0.81
100000	0.025231292	0.020712453	0.82
1000000	0.304166761	0.258099884	0.85

2.2.3. *Bulldozer microarchitecture.* Much less optimization effort was made for Bulldozer than for Haswell and Skylake, owing to the age and poor performance of this processor. No code was written from scratch, but among all the existing implementations for a given function, the one that ran fastest on Bulldozer was chosen.

Among those functions that were replaced by faster versions, these three are relevant to the FFT butterflies:

Function	Old	New
com_n	1.28	0.723
rshift	2	1.11
lshift	2.42	1.24

Timings for mpn_mul_n in seconds:

Limbs	Old	New	Ratio
10000	0.004771156	0.004764643	1.0
100000	0.054624774	0.053038739	0.97
1000000	0.651062127	0.652278285	1.0

Unfortunately, the improvements to the mpn_[lr]shift functions are barely visible in the integer multiplication benchmark on Bulldozer.

All code written for this deliverable has been committed to Alex Kruppa's fork of the `MPIR` repository at `https://github.com/akruppa/mpir` and merged into the main `MPIR` repository at `https://github.com/wbhart/mpir` and will be available in the MPIR-3.0.0 release, available at the `MPIR` website.

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` assembler 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.

2.3. Blog post

A blog post about the design of the Flint FFT and the work done for this project is available at `https://wbhart.blogspot.de/2017/02/parallelising-integer-and-polynomial.html`.

Disclaimer: this report, together with its annexes and the reports for the earlier deliverables, is self contained for auditing and reviewing purposes. Hyperlinks to external resources are meant as a convenience for casual readers wishing to follow our progress; such links have been checked for correctness at the time of submission of the deliverable, but there is no guarantee implied that they will remain valid.