A Design Case Study: CPU vs. GPGPU vs. FPGA

Daniel L. Rosenband
danlief@alum.mit.edu

Till Rosenband
norstadt@alum.mit.edu

Abstract

This paper describes our winning submission for the Absolute Performance category of the MEMOCODE 2009 Design Contest. We show that our GPGPU-based design achieves performance within a factor of four of theoretical maximum performance for the implemented algorithm. This result was reached after a short design-cycle of 2 man-days, which indicates that the NVIDIA CUDA platform allows for rapid development and optimization of applications that make substantial use of all available GPGPU computing resources. We also analyze the maximum theoretical performance of alternative computing systems that could have been used to implement the algorithm.

1. Introduction

This paper describes our design experience during the MEMOCODE 2009 Design Contest[1]. Our submission won the Absolute Performance category using an NVIDIA GPGPU. We are GPGPU novices and ended where we did as much through trial-and-error design refinements as via the reading of random web-forum postings and papers that attempt to reverse engineer many of the un-documented GPGPU internals. As such, we do not attempt to convey deep insights into the internal GPGPU workings or how to eek out that last bit of performance. Nonetheless, our design resulted in performance within a factor of four of what we believe is the theoretical maximum for the GPGPU and algorithm we selected. We believe this is a significant result given that it led to high performance (29 GFLOPs, 90 GOPs and 16 GB/s memory bandwidth) in only a short design-time.

The highlight of our contest experience was the observation that we could rapidly transition from a simple C-based x86 implementation to a reasonably well optimized GPGPU implementation via as series of small design refinements. Each design refinement led to a 0x to 10x performance improvement and could be implemented in minutes and tested in seconds. We believe if available, a design-flow that enabled similar exploration and refinement on FPGAs would broaden the appeal of FPGA accelerators.

The next sections describe: design contest problem description (Section 2), the design choices we made (Section 3), implementation details (Section 4), results (Section 5), and a commentary on design flows (Section 6).

2. Problem Description

This section briefly describes the design contest problem. A full description can be found in [2].

The core contest problem requires visiting each point in an *NxN* polar grid. At each point in the grid the design must compute the average value associated with the four surrounding points of an overlaid *NxN* Cartesian grid. The resulting average value must then be assigned to an array entry associated with the polar point. Figure 1 shows the overlaid coordinate systems.

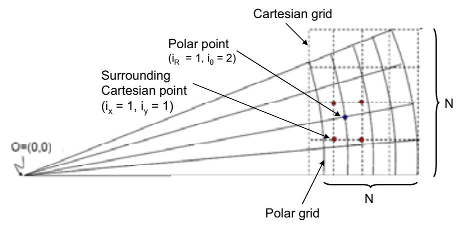
Cartesian grid
Polar point (i_R = 1, i_θ = 2)
Surrounding Cartesian point (i_x = 1, i_y = 1)
O=(0,0)
N
N
Polar grid

Figure 1: Overlaid coordinate systems

The pseudo-code in Figure 2 illustrates the core algorithm. In this algorithm:

- *N* represents the problem (grid) size.
- *index2polar*, *polar2cart*, and *cart2index* convert between coordinate systems and require floating point operations.
- *C* and *P* are the arrays that hold the values associated with the Cartesian and polar points.

```
for $i_\theta$ = 0 to N - 1 do
  for $i_R$ = 0 to N - 1 do
    ($\theta$, R) = index2polar($i_R$, $i_\theta$);
    (x, y) = polar2cart($\theta$, R);
    ($i_x$, $i_y$)= cart2index(x, y);
    P[$i_R$][$i_\theta$] := (C[$i_x$][$i_y$] + C[$i_x$+1][$i_y$] +
                 C[$i_x$][$i_y$+1] + C[$i_x$+1][$i_y$+1]) / 4;
```

Figure 2: Core algorithm

3. Design platform choice

Abstract performance model

The design implementation is constrained by two components:

- *External memory bandwidth*: For each inner-loop iteration the design must read 4 values from array *C* and write one value to array *P*. Each value is of size 4 Bytes. An optimized design caches (portions of) the *C* array as it is being read. This results in an external memory transfer requirement of at least $4N^2$ Bytes read and $4N^2$ bytes written, or $8N^2$ total bytes transferred to/from external memory.
- *Computation*: The algorithm in Figure 2 can be optimized to perform some of the computation just once per outer-loop iteration. The resulting computation requirement, where *T*, *F*, and *I* are the number of cycles per transcendental (*sin* and *cos*), floating-point, and integer operation respectively, is approximately:
  a. *Outer-loop*: $2T + 12F$
  b. *Inner-loop*: $4F + 40I$
  c. *Total*: $N(2T + 12F) + N^2(4F + 40I)$

  (Note: our implementation required double-precision for the transcendental and floating-point operations.)

Platforms considerations

We considered the following platforms for our contest submission:

• *x86*:	Intel Core i7 3.2GHz
• *GPGPU*:	NVIDIA GTX 285 (used here)
• *XUPV2*:	Xilinx board with Virtex-II Pro
• *V5*:	Conceptual Xilinx Virtex5 board

With the exception of *V5*, each of these platforms is readily available for approximately $1,000. We include *V5* as a conceptual board since it highlights the capabilities that one could achieve with a state-of-the-art FPGA. The *V5* concept replaces the GPGPU on a graphics card with a Xilinx Virtex 5 SX240T. It assumes high-speed memory attached to most of the FPGA pins (1000 pins at 800 Mbps).

Performance Estimates

Figure 3 shows "will-not-exceed" performance estimates for each of the platforms[3][4][5][6]. These numbers assume fully utilized hardware resources. Note: not all performance characteristics are publically available or could be easily determined. For example, we did not find reliable references on transcendental performance. Similarly, integer performance on FPGA platforms is not well defined since it is application dependent. We attempted to make accurate estimates when data was not available.

The last three rows in Figure 3 show the memory transfer time (T_{DT}) and compute time (T_C) for the design contest problem of size *N=1000*. We also included the total time estimate, which is the maximum of T_{DT} and T_C since we assume computation and memory accesses are pipelined. A few noteworthy observations:

- *GPGPU* has the highest memory bandwidth.
- *GPGPU* has the best double precision floating point performance.
- *V5* has the best overall performance since we estimate that it has better transcendental and integer performance than an x86 CPU or a GPGPU.

	x86	GPGPU	XUPV2	V5	Units
Memory Bandwidth	32	159	2	100	GB/s
Clock rate	3.20	1.51	0.10	0.55	GHz
Cores	4	30	9	124	
CPI (per core)					
Transcendental	10	100	20	20	cycles
Double Precision Float	0.25	0.5	0.5	1	cycles
Integer	0.125	0.125	0.0625	0.031	cycles
DP GFLOPs	51.20	90.72	1.80	68.20	
Memory Transfer Time T_{DT} (N=1000)	250	50	3759	80	µs
Compute Time T_C (N=1000)	471	159	5051	78	µs
Total Time = Max(T_{DT}, T_C) (N=1000)	471	159	5051	80	µs

Figure 3: Estimated will-not-exceed performance

Design platform for contest submission

Since the *V5* platform is purely a concept, we chose a GPGPU as our main contest platform. The platform consists of:

- Host CPU: AMD Athlon 64 X2 4200+
  - 2200 MHz
  - 200 MHz DRAM clock
  - 2 GB DDR DRAM
- GPGPU: NVIDIA GTX 285 GPGPU
  - 666 MHz core clock
  - 1512 MHz shader clock
  - 2484 MHz DRAM clock
  - 1 GB GDDR3 DRAM

(Shortly before the contest deadline) we realized that the GPGPU has a 10µs - 20µs start-up cost associated with each GPGPU kernel execution. This was a substantial overhead for the small design test cases (*N=10*). Thus, we ended up using a hybrid approach: For small input sizes ($N \leq 40$) we use the host CPU, for large problems ($N > 40$) we use the GPGPU. We did not have time to optimize the host CPU code (or platform), so results for small *N* are slower than they might have been.

4. GPGPU implementation optimizations

This section highlights some of the design optimizations that improved the GPGPU performance from our initial implementation. As previously mentioned, we are GPGPU novices, so these are rather basic optimizations. We found [4], [7] and a slew of web-sites which reverse engineer GPGPU internals most helpful in understanding the GPGPU architecture.

GPGPU threading

The GPGPU is capable of executing many *threads* in parallel. The programmer can group threads into *blocks*. All threads in a block execute on one of the GPGPU multiprocessors (our GPGPU has 30 multiprocessors). Within a GPGPU multiprocessor, the GPGPU schedules threads in groups of 32 threads, also known as *warps*.

The most straight-forward implementation would have assigned a single thread to each inner-loop iteration (polar point) in the algorithm of Figure 2. For reasons we explain in the next subsections, we chose to implement the design as *N / 4* blocks of 64 threads each. Each thread iterates over a subset of the two nested loops in Figure 2. Each thread computes values for:

- 4 angles (hence *N / 4* blocks)
- 1/64th of the radial values (hence 64 threads per block)

Memory access optimization

The GPGPU has very high maximum (and achievable) memory bandwidth. Programmers must carefully lay out the memory access patterns in order to achieve this bandwidth. In general, if threads in a warp accesses sequential memory addresses then the GPGPU memory controller coalesces the accesses and takes best advantage of the available bandwidth.

In our case a thread computes on points with radial index *r, r+64, r+128*, etc. The "next" thread computes on points with radial index *r+1, r+65, r+129*, etc. Since each multiprocessor effectively computes on all 32 threads in a warp simultaneously, the hardware can efficiently coalesce these memory references. For example, the first memory access would return data for the first iteration of 32 of the threads: *r, r+1, ..., r+31.*

We believe that we were able to take advantage of GPGPU internal cacheing by making each thread compute over 4 angles. However, cacheing structures are not well documented, and we used trial-and-error to determine the optimal value of 4 angles per thread.

Computational optimizations

For a given angle, all inner-loop (radial) computations share approximately 2 transcendental and 12 floating point computations. If one thread is assigned to each inner-loop iteration, then the GPGPU must replicate the shared computation in each thread – a substantial overhead. We were able to reduce that overhead by making each thread compute points for multiple radial values for a given angle. We found that 64 threads per angle produced optimal results. For example, for *N=1000*, the GPGPU now performs the shared computation 64 times (once per thread), rather than 1000 times if we had assigned a thread to each radial coordinate. (Note: we did not find an efficient mechanism to share values across threads, which could have further reduced this overhead.)

5. Results

Figure 4 shows the execution times for the 12 required test-cases. The columns contain:

- *T0* and *T1* show the time for two separate executions.
- *Best* picks the fastest time of *T0* and *T1*
- *Ref* shows the reference time that the contest organizers achieved using an un-optimized software only implementation on the XUPV2.
- *Speedup* shows the performance improvement in our design over the reference implementation.
- *Geometric mean of speedup* takes the geometric mean of the *speedup* values. This is the value that determined the winner in the Absolute Performance category.

Test	T0 (us)	T1 (us)	Best (us)	Ref (us)	Speedup
0	11	3	3	10,958	3,653
1	375	378	375	55,619,458	148,319
2	602	607	602	113,455,224	188,464
3	3	3	3	11,033	3,678
4	316	313	313	56,893,223	181,767
5	516	511	511	116,084,610	227,171
6	3	3	3	11,021	3,674
7	298	280	280	55,963,646	199,870
8	482	469	469	112,892,179	240,708
9	4	4	4	18,709	4,677
10	310	301	301	53,248,232	176,904
11	454	431	431	94,741,983	219,819
Geometric mean of speedup			**53,064**		

Figure 4: Results

Figure 5 shows the achieved execution time (big black dots) vs. the predicted will-not-exceed execution times (lines). For large *N* we achieved within a factor of 4 of the maximum GPGPU performance. We did not optimize the execution time for small *N*.

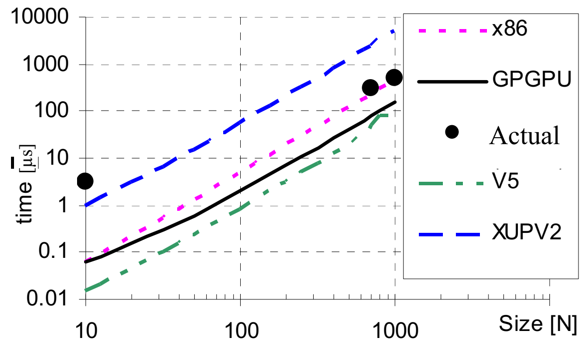

Figure 5: Actual vs. will-not-exceed performance

6. Commentary on design methodologies

This design contest was our first practical exposure to GPGPU computing. The design methodology we used was:

- Generate a working solution. This was very similar to a C version that runs on x86.
- Refine the design through small changes. In some cases these changes led to large performance improvements. Most changes took minutes to implement and seconds to compile and test.
- Gradually gain understanding of GPGPU internals and programming tricks.

Our impressions were:

- It was very quick to get "something" working.
- Many of the GPGPU internals are not well documented. Nonetheless, the core concepts are understandable and reasonably easy to code to. Given the lack of understanding, trial-and-error was sometimes required to optimize performance.
- We did not use some of the more advanced GPGPU capabilities which could have further improved performance, e.g. the texture memory interpolation capabilities.
- The resulting performance (1/4 of will-not-exceed performance) was in-line with our goals. We spent less than 2 man-days on implementation and are certain that with further effort additional performance is obtainable.

We believe a hardware design-flow that allows designers to very quickly get "something" working, and then quickly iterate on refinements would broaden the appeal of FPGA accelerators. This would almost certainly lead to sub-optimal results (similar to our GPGPU case), but given the tremendous compute capabilities in today's FPGAs, in many cases that may be acceptable.

7. Acknowledgements

We would like to thank the contest organizers and sponsors for holding the MEMOCODE 2009 Design Contest. We especially thank Prof. James Hoe for his responsiveness to questions throughout the contest.

8. References

[1] F. Brewer and J. C. Hoe, "2009 Memocode Co-Design Contest", *http://www.ece.cmu.edu/~jhoe/wiki/index.php/ 2009_Memocode_Co-Design_Contest*

[2] F. Brewer and J.C. Hoe, "MEMOCODE 2009 Design Contest: Cartesian-to-Polar Interpolation", *http://www.ece.cmu.edu/~jhoe/distribution/mc09contest/ contest09.pdf*

[3] K.J. Barker, K. Davis, A. Hoisie, D.J. Kerbyson, M. Lang, S. Pakin, and J.C. Sancho, "A performance evaluation of the Nehalem quad-core processor for scientific computing", *Parallel Processing Letters*, Vol. 18, No.4 (2008), pp. 453-469.

[4] V. Volkov and J.W. Demmel, "Benchmarking GPUs to tune dense linear algebra", *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, November 15-21, 2008, Austin, Texas

[5] G. Kuzmanov and W.M. van Oijen, "Floating-Point Matrix Multiplication in a Polymorphic Processor", *International Conference on Field-Programmable Technology* (2007), pp. 249-252

[6] "Xilinx Virtex-5 Multi-Platform FPGAs", http://www.xilinx.com/products/virtex5/

[7] "NVIDIA CUDA Programming Guide", Version 2.2, http://developer.download.nvidia.com/compute/cuda/2_2/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.2.pdf