Fitting FFT onto the G80 Architecture

Vasily Volkov Brian Kazian

University of California, Berkeley
May 19, 2008.

Abstract

In this work we present a novel implementation of FFT on GeForce 8800GTX that achieves 144 Gflop/s that is nearly 3x faster than best rate achieved in the current vendor's numerical libraries. This performance is achieved by exploiting the Cooley-Tukey framework to make use of the hardware capabilities, such as the massive vector register files and small on-chip local storage. We also consider performance of the FFT on few other platforms.

1 Motivation

There are two sources of motivation for this work. First is the recent success in running matrix-matrix multiply on G80 GPUs. Volkov and Demmel [2008] demonstrate routines that outperform vendor's libraries by 60% and show predictable performance. They outline a novel guidelines for programming G80 GPUs that promise speedups in other applications.

Second motivation is that vendor's libraries show performance in FFT that is substantially below any reasonable estimate.

The goal of this paper is to use techniques outlined by Volkov and Demmel [2008] to control performance of FFT on G80 GPUs. This includes aggressively exploiting the large register files on the GPU, keeping usage of shared memory low and using shorter vectors (thread blocks).

We also felt that it was important to look at how FFT's perform on other multicore architectures, such as Clovertown and Niagara II. This enables us to better understand the pitfalls of each architecture and suggest methods for better applying a parallel mapping to the given architecture.

2 Introduction to FFT

The discrete Fourier transform (DFT) is defined as

$$y_j = \sum_{k=0}^{N-1} w_N^{jk} x_k, \tag{1}$$

where $x_1, \ldots, x_N$ are inputs, $y_1, \ldots, y_N$ are outputs and

$$w_N = \exp\left(-\frac{2\pi i}{N}\right), i = \sqrt{-1}.$$

For example, for $N = 2$ the transform is

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \tag{2}$$

and for $N = 8$ it is

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & u & -i & v & -1 & -u & i & -v \\ 1 & -i & -1 & i & 1 & -i & -1 & i \\ 1 & v & i & u & -1 & -v & -i & -u \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & -w & -i & -v & -1 & u & i & v \\ 1 & i & -1 & -i & 1 & i & -1 & -i \\ 1 & -v & i & -u & -1 & v & -i & u \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{pmatrix}, \tag{3}$$

where $u = w_8 = (1 - i)/\sqrt{2}$ and $v = w_8^3 = (-1 - i)/\sqrt{2}$.

1. Lay out the data into $N_1 \times N_2$ matrix in column-major order.
2. Perform DFT on each row of the matrix (Eq. 6).
3. Scale the matrix component-wise (Eq. 5).
4. Transpose the matrix (Eq. 5).
5. Perform DFT on each row of the matrix (Eq. 4).

Figure 1: Cooley-Tukey framework.

The transform matrices have amounts of structure, due to relations such as $w_N^N = 1$, $w_N^{jk} = w_N^{kj}$, and $\sum_{j=0}^{N-1} w_N^j = 0$. This may be used to compute the transform in a smaller number of arithmetic operations than N^2 assumed by the naïve matrix-vector multiply in (1).

One particular fashion of exploiting the structure of matrix W is due to Cooley and Tukey [1966] (see also Duhamel and Vetterli [1990]). If $N = N_1 N_2$ for some integer N_1, $N_2 > 1$, then (1) can be rewritten as

$$y_{j_1 N_2 + j_2} = \sum_{k=0}^{N_1 - 1} w_{N_1}^{j_1 k} z_{k N_2 + j_2}, \tag{4}$$

where

$$z_{j_1 N_2 + j_2} = w_N^{j_1 j_2} s_{j_1 + j_2 N_1}, \tag{5}$$

and

$$s_{j_1 + j_2 N_1} = \sum_{k=0}^{N_2 - 1} w_{N_2}^{j_2 k} x_{j_1 + k N_1}. \tag{6}$$

Scales used in (5) are known as "twiddle factors". This can be understood as similar to 2D Fourier transform, see Figure 1.

This framework reduces one large DFT into many smaller DFTs. Also, it reduces the count of complex operations from N^2 down to $N_1 N_2^2 + N_2 N_1^2 = N(N_1 + N_2)$. The technique can be applied recursively. If N is a power of r, the total operation count then can be reduced to $2r \log_r N$. This allows computing DFT for very large N at nearly linear time and thus is known as "Fast Fourier Transform" or FFT. Small r results in lower number operations and $r = 2$ or $r = 4$ are often preferred as they don't require floating point multiplications in the smaller DFTs (as in Eq. 2). Note that even if N is a power of two, it is not necessary to run recursion down to radix-2 DFTs — radix-4 DFTs provide about same efficiency.

Note, that transposition changes only the layout of data and thus can be moved sooner or later in the algorithm, in and out the recursion, provided that data locations are properly adjusted in all of the involved computations. Instead of doing DFTs on rows, transposing and doing them on rows again as shown in Fig. 1, one can first do rows, then columns, then transpose. Or, first transpose, then do columns and then rows. If desired, all transpositions can be moved out of the recursion to the beginning of the algorithm. This technique is known in signal processing literature as decimation-in-time (DIT). Similarly, all transpositions can be moved to the end of the algorithm. This is

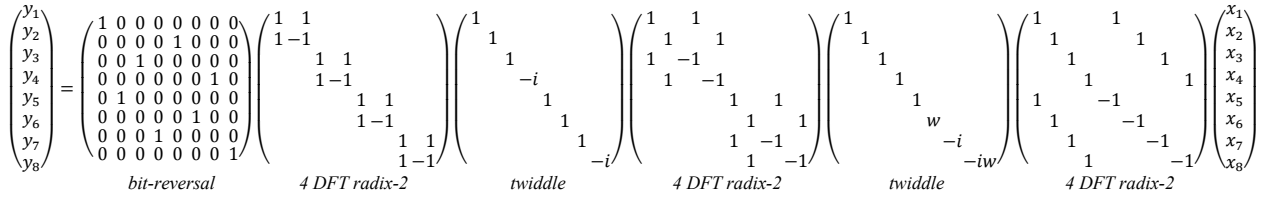

Figure 2: DIF version of complex-valued Cooley-Tukey FFT for $N = 8$ that does the same work as in (3). Here, $w = (1 - i)/\sqrt{2}$. Note that only 4 floating point multiplies are required to evaluate the expression. One can recognize numerous radix-2 DFTs as in (2).

called decimation-in-frequency (DIF). The aggregated transpositions correspond to "bit-reversal" — data is moved to the location found by reversing the order of bits in the binary representation of the original index. An example of applying this framework to case $N = 8$ is shown in Figure 2. Compare it to Eq. 3 that does the same work.

Similar transformations can be used to adapt the algorithm to a particular hardware. Note, for example, that in Fig. 2 the rightmost matrix defines transform that can be implemented as operations on stride-1 vectors of four. This corresponds to radix-2 operations on rows in Cooley-Tukey framework. However, further matrices in the Figure don't have this nice property. This difficulty can be solved again using Cooley-Tukey framework. In that case N is factored into N_1 and N_2 that are both sufficiently large. In that case all row transforms in Fig. 1 will always be replicated over many rows and thus involve long stride-1. This technique is useful for vector computers and is discussed for example by Bailey [1987].

Cooley-Tukey framework also allows exploiting memory hierarchy. In that case we choose N_2 small enough so that N_2 elements fit into the desired memory level at once. Then the framework proceeds but with extra transpose in the begin so that first set of Fourier transforms is done on columns, each column fitting into that memory hierarchy level. Technique can be applied recursively. Bayley [1990] uses this technique to design out-of-core FFT algorithms.

There are other frameworks that exploit the structure of the DFT matrix to reduce the operation count. Good [1958] describes a similar algorithm that does not require scaling by twiddle factors but requires N_1 and N_2 be co-prime. Rader [1968] and Bluestein [1970] describe O(N log N) algorithms that work with prime N. All DFT algorithms that run at $O(N \log N)$ time are usually called FFT.

Minor adjustment to the algorithm may produce the inverse transform to DFT. Applying DFT to rows and columns of a given matrix yields a 2D Fourier transform. Similar algorithms exist that work with real-valued inputs and/or real-valued outputs, perform discrete sine and cosine transforms.

3 Related Work

CUFFT is NVidia's implementation of an FFT solver on their CUDA architecture. It operates by taking a user-defined plan as input which specifies the parameters of the transform. It then optimally splits the transform into more manageable sizes if necessary. These sub-FFT's are then farmed out to the individual blocks on the GPU itself which will handle the FFT computation. CUFFT employs a radix-n algorithm and was the impetus for this project. The algorithm seems fairly ill-suited to the optimal method of coding for the architecture. The performance also falls far short of the bandwidth peak of the architecture.

Lloyd, et. al [2008] implemented an out-of-place, radix- 2 Stockham algorithm. They chose this algorithm as it eliminates the need for bit reversal which can be a costly operation. This algorithm utilized the texture stores for holding the FFT data. Since texture memory cannot be written to, this led to the implementation being forced to be an out-of-order one, requiring twice as much memory for the transform. The performance of this implementation fell short of CUFFT in all aspects except for real 2D FFTs. By utilizing complex data types to hold two reals, they were able to see a performance increase. Although this approach investigated different memory models for performing an FFT transform, this does not seem to be the way to go. The problem lies more with the global communication in the algorithm as opposed to the actual memory access patterns of the existing implementation.

FFTW was investigated for this paper as a benchmarking tool for platforms other than CUDA. FFTW is a library of FFT routines which will provide optimized code for a given transform. FFTW was the interface from which CUDA was derived as it also creates a plan for a given transform and can then continually execute it. FFTW achieves its competitive performance by actually processing the transform initially when the plan is created. It checks through a series of optimized codes to see which one performs best for the given transform on the current architecture. These optimized routines are known as "codelets" and are chosen at runtime. The user can also create a series of codelets for FFTW to choose from. This approach to optimization is one that was looked into for the project. We felt that the best performance for FFT on any architecture necessitates some form of specialized codes for a given subset of problem sizes. Since FFTW has been pretty successful with this approach, we felt that we may too.

Very similar to FFTW is the implementation of SPIRAL. The main differences between the two is that SPIRAL determines the problem approach at compile-time and searches over a series of mathematical expressions as opposed to the lower-level details in FFTW. Another big difference is that SPIRAL is machine dependent. This follows our logic that to get the best performance out of our FFT routine, we need to clearly optimize for the target platform.

4 Processor Architecture

4.1 GeForce 8800 GTX

The GPU architecture is described in CUDA programming guide and is analyzed in detail in [Volkov and Demmel 2008]. GeForce 8800GTX has 16 SIMD cores that run 32-element vector threads. The cores have 8 scalar lanes. Vector thread can communicate via shared on-core memory and this allows simulating variable vector length. Vector length 64 is often found best.

4.2 Niagara II

The Niagara II boasts a collection of 8 cores, each of which can execute 8 threads simultaneously. These 8 threads are formed

by two groups of 4 threads each. Each group of threads has access to a fully pipelined FPU which is located on each core. This is a large improvement over the original Niagara which only had one FPU shared amongst all cores. With each core being able to issue one floating point operation per cycle, there is a total of 1.4 GFlop/s per core and an aggregate maximum of 11.2 GFlop/s for the socket.

Each core has its own data and instruction cache and a shared L2 cache among all cores. The L1 cache is 8KB and is possibly shared by 8 threads. The L2 cache is a 16 way set-associative cache and is 4MB total. There is an 8x9 crossbar attaching all of the cores to the L2 cache which allows for up to 179 GB/s for reading.

4.3 Clovertown

The Intel Quad-Core (Clovertown) is two Intel Woodcrest chips fused together on a single package. This results in a total of 4 cores, with each pair sharing a 4MB L2 cache and all cores communicating via a single 1333 MHz FSB. The addition of 2 cores to the same basic architecture as Woodcrest results in a decrease of the per core bandwidth. This single point of communication between all cores can prove to be a bottleneck for memory-intensive applications.

5 Design of the GPU Algorithm

We have a few general guidelines for designing efficient GPU algorithms. First, optimal vector length is 64, which is the smallest that permits high throughput. Any longer vector parallelism in a program should be strip-mined into short vectors. Second, the primary on-chip data storage is the register file. Each scalar thread keeps as much data as possible. In our implementation we chosen 8. Thus, each scalar thread can perform FFT for $N = 8$ in-registers following the matrix formula in Fig. 1. In-register FFTs imply very fast communication and thus high throughput. The second communication level is via shared memory. After 64 threads in a block did their in-register DFTs, they exchange data and do next in-register DFTs. This allows N as large as 64*8 = 512 and requires two transpositions inside a kernel. The third communication level is via global memory and is not currently implemented. This requires running few kernels or using barrier to synchronize. The goal of this hierarchy of communications is to amortize communication as much as possible, as it's usually the bottleneck. Conceptual sketch of the algorithm is in Fig. 3.

This design was implemented for N =512, $N = 64$ and $N = 8$. Smaller N were implemented for debugging purposes as stages in developing the case of $N = 512$. Our prototype implementation doesn't work with different N. Instead we concentrated on getting best results at at least one N to reveal the hardware potential.

Due to specifics of the GPU memory access (non-SIMD accesses run at an order of magnitude lower bandwidth) data in the cases $N = 8$ and $N = 64$ is laid out in DRAM in a special order to facilitate high-bandwidth memory access. This restriction can be easily overcome by extra two reshufflings of data using shared memory that would incur low overhead. However, it was not our concern in this paper and was not done.

We created custom kernels, where N is hardcoded. This is not unusual technique and is used in many other high-performance FFT algorithms, such as FFTW and hardware processors.

We tried to tabulate trigonometric functions in twiddle factors using constant memory, texture cache and shared memory. However, these techniques failed to get speedup versus a naïve approach that uses intrinsics to compute them.

Load data from DRAM
Compute small FFT in-registers
Local shuffle via shared memory
Compute in-registers
Local shuffle via shared memory
Compute in-registers
Global shuffle via DRAM
Compute
Local shuffle
Compute
Local shuffle
Compute
Global shuffle
<…>
Save data to DRAM

Figure 3: Scheme of the hierarchic communication in FFT. The purpose is make global communication as rare as possible and spend most of the time in local compute. In our particular GPU implementation we didn't implement global shuffles. Computation stages operate on 8 elements of local data stored in registers. Shuffles correspond to the transposes in the Cooley-Tukey framework.

5.1 Experimental Methodology on Niagara and Core2

For benchmarking FFT's on both Niagara II and the Intel Clovertown, FFTW3 was used. The library was compiled on each system for single precision with pthreads enabled. The FFTW_MEASURE flag was used. This increases the amount of runtime performance monitoring that occurs for the transform to potentially improve performance. The transforms were 1D Complex to Complex and were performed in-place.

The benchmark itself involved transforming a series of FFT's increasing in size. Smaller-sized FFTs were batched together simultaneously for a more accurate view of the available parallelism. FFTW allows for this batching of multiple FFTs simultaneously. However, this approach did not work correctly on the Niagara II, causing any series of batched FFTs larger than 64 elements to produce very poor results. The values were consistently less than a 0.01 GFlop/s. An alternate method was used wherein pthreads were created for each FFT that was to be run in a batch. Threads were enabled in both cases to also perform on each batched FFT transform in parallel.

6 Performance Results

Fig. 4 shows the performance of our GPU FFT implementation. It achieves up to 144 Gflop/s on GeForce 8800 GTX. This is 2.9× better than the best rate achieved in NVIDIA CUFFT 1.1, which is 50 Gflop/s. Also, it is ~2× faster than the best unreleased code that NVIDIA currently has [Nickolls 2008]. Same graph shows the rates achieved in the CUFFT 1.1 source codes of radix-2 FFT that are released by NVIDIA ("original"). It runs at about the same rate as CUFFT 1.1. The differences at $N = 256$ and $N = 1024$ might be due to radix-4 code that CUFFT 1.1 also uses but we didn't compile individually. Another curve on the plot titled "optimized" is the performance of the CUFFT 1.1 code that includes basic optimization did by us.

Optimizations include unrolling the loop, hard-coding the value of N and other little tweaks such as in bit operations and integer arithmetic. This gave up to 1.77× speedup. So large speedup achieved by basic optimizations done within a couple of hours highlight the little amount of effort applied by vendor's developers in programming these routines. However, our own radix-8 design is still 2.7× faster than these optimized codes.

Figure 5 shows the performance of our best code and CUFFT 1.1 matched versus the machine peaks. It is inspired by roofline figures by Williams et al. [2008]. The bandwidth peak curve in the plot corresponds to the lower bound on the algorithm runtime as dictated by the bandwidth requirements due to reading the input and writing the output. We assumed that these memory accesses run at 70GB/s, which is a peak sustained bandwidth number and number of flops done is 5 N log N just as used in measuring the algorithm performance. According to the Figure, our code runs nearly at the bandwidth bound, i.e. close to optimal. At $N = 512$ it runs at 73% of the bound that indicates further opportunities for designing a faster algorithm. The same figure shows two arithmetic peaks. One is in operations such as multiply-and-add (MAD), another is in adds and multiplies, which is twice as low since runs at the same instruction throughput. The latter bound may be more realistic, since radix-8 algorithm does most of the flops in additions and subtractions. However, it should be noted that the actual number of flops done by our FFT is less then 5 N log N due to the arithmetic features of the radix-8 computation. From other point, 5 N log N figure does not include trigonometric functions that we use in twiddle factors. Other source of the slowdown is the permutations that we do using shared memory and some pointer arithmetic.

The figure also shows the local storage bound. If N is larger than fits into local storage (the registers), multiple smaller FFT kernels must be run. This bounds the performance by the performance of the small FFTs that fit into the local storage. In other words, any FFT performance curve cannot grow past this line. This effect is observed for example with CUFFT 1.1.

The figure also highlights what changes to the hardware might improve the performance of the algorithm. Clearly, improving bandwidth will increase the performance of the algorithm at small N. At larger N the performance seems to be bound by instruction throughput instead. Improving it might allow higher peaks in FFT. Example of improvement may be dual issue, such as in VLIW, e.g. by co-issuing floating point operations and shared memory load/stores. Note that increasing size of the local storage is unlikely to yield higher performance, as the performance growth is already slowing down at $N = 512$.

Table 1 compares performance of our FFT with several earlier implementations. Note that the performance of FFT on GPUs has grown by two orders of magnitude in past few years. This is due to increased programmability, such as introduction of shared memory in 8800GTX.

It is also may be interesting to put this performance into context of other modern chips. For example, Chow et al. [2005] reports 46.8 Gflop/s on Cell processor. Arithmetic peak of 2.4GHz Core2 Quad in multiply-and-add operations is ~77 Gflop/s that bounds the performance of FFT. Thus, GPUs provide substantial performance benefits due to their higher throughput.

6.1 Results on Other Architectures

The FFTW results for Niagara II (see Fig. 6) were not outstanding. The performance was seen to max out at around 4.7 GFlop/s for a transform size of 32768. One issue with the

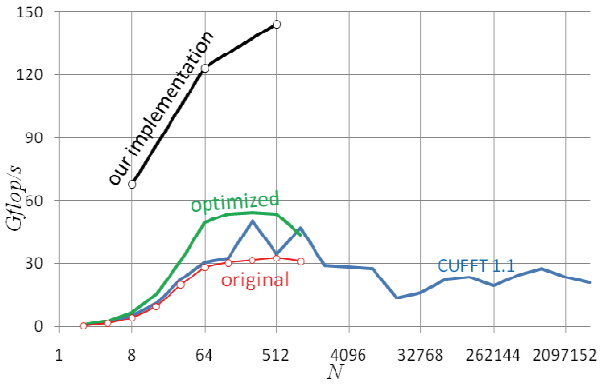

Figure 4: Performance of our GPU implementation versus vendor's codes. "Original" is our compilation of the vendor's radix-2 code. "Optimized is basic optimization of that code. The platform is GeForce 8800GTX.

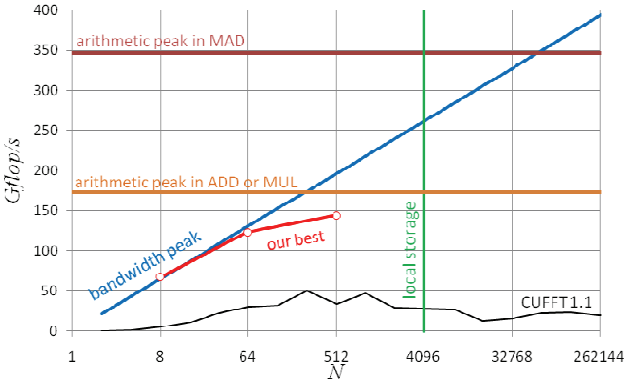

Figure 5: performance of our and NVIDIA codes compared versus the arithmetic, bandwidth and local store bounds.

Reference	Rate	GPU
Spitzer [2003]	1.1 Gflop/s	5900 Ultra
Moreland and Angel [2003]	2.5 Gflop/s	5800 Ultra
Govindaraju et al. [2006]	6.1 Gflop/s	7900 GTX
McCool et al. [2006]	7.5 Gflop/s	7900 GTX
Segal and Peercy [2006]	12 Gflop/s	X1900 XTX
CUFFT 1.1 [2007]	50 Gflop/s	8800 GTX
Lloyd et al. [2008]	18 Gflop/s	8800 GTX
This paper, 2008	144 Gflop/s	8800 GTX

Table 1: Historical comparison of performance of FFTs on GPUs. X1900 XTX is from ATI (now AMD). Others are GeForce solutions from NVIDIA.

Niagara II was the placement of data in the buffers. For the batching of multiple transforms simultaneously, successive calls to malloc added padding to the end of each buffer to ensure better distribution in the cache. Without the padding, performance degradation became more severe at certain points. By placing these buffers into different banks in the cache, performance became more stable at certain powers of two. The other caveat with the Niagara II is that it is built to harness a larger number of threads, similar in practice to a GPU. Without a large number of threads running simultaneously, the memory latency cannot be hidden and the performance degrades significantly. Since prefetching occurs only into the L2 cache, it is necessary for enough threads to execute to offset this latency. In terms of the FFTW performance on the Niagara, the routines are not optimized for a SPARC processor in that they cannot

take advantage of the VIS instruction set. The VIS instruction set enables the UltraSPARC processors to utilize SIMD instructions which would provide a substantial benefit for the benchmark performance. Furthermore, the fact that the advanced API for FFTW do not produce proper results when a batch of FFT's are processed is disappointing. It seems the internal threading in FFTW may not be optimal for the Niagara, in terms of either the placement of threads or the number applied to a given problem. Another possibility is that FFTW changes the layout of the buffered transforms after it reads them in. This is a concern with Niagara as padding can have an impact on the performance in terms of cache effects. The Niagara architecture is well suited for large transforms or batches of transforms. Without large datasets or transform sizes, the performance is very poor when compared to less threaded architectures, such as the Clovertown.

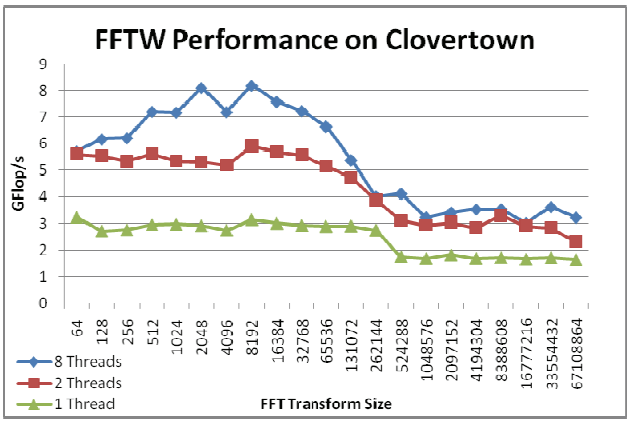

Fig. 6. GFlop/s from FFTW3 on Clovertown

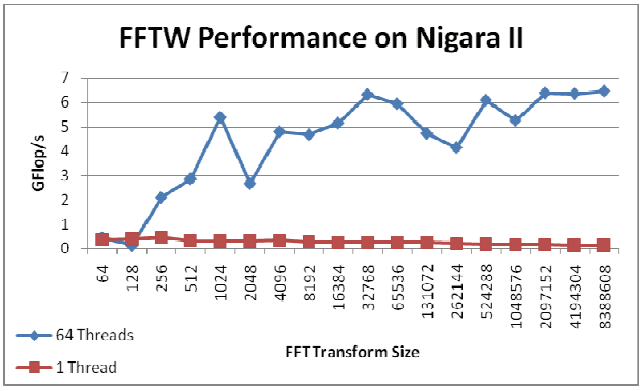

Fig. 7. GFlop/s from FFTW3 on Niagara II

The Clovertown results (see Fig. 7) were more in line with published results for the FFT. The tests were performed with a new configuration of FFTW for single precision. The previous benchmark from the Niagara with batching was also used for this architecture. The icc compiler was used to compile the benchmark and various numbers of threads were run. The benefits of parallelization were obvious when going from one thread to two as the performance doubled in most cases. This is most likely the fact that two threads are running on the same chip and sharing the L2 cache. Once the transform grows too large for the cache itself, the performance severely degrades. Increasing the number of threads up to 8 did not have a huge impact on the performance, most likely due to the contention on the bus due to the shuffling of the transform to the different threads. The Clovertown differs from the Niagara in the sense that parallelism is not as imperative to receive decent performance. One thread on the Clovertown was able to achieve a reasonable amount of GFlop/s and the speedup was almost linear when the number of threads was increased to 2.

6.2 Architecture Improvements

There are some aspects of the above architectures that could be altered to allow for better performance of these algorithms. In the case of the Niagara II, it really showed how poor performance is in terms of intra-core communication.

When multiple threads were working on a single FFT transform, the performance only broke the 1 GFlop/s mark at a size of 2^{18} with a thread count of 64. However, the interesting point is that performance did not degrade rapidly as the thread count increased for this size of FFT. This could be attributed to poor memory management or codelet selection in terms of FFTW's chosen approach. If the memory addresses of a given thread's data conflicts with another in the cache, this could lead to the severe performance degradation. The small L1 cache also poses problems for multiple threads working on rather large sets of data. This would cause performance degradation due to data being ejected from the cache.

One method of improving the cache thrashing issue with Niagara II in hardware is to either increase the associativity of the cache or to split the L2 cache into separate ones for a subset of the cores. Both of these options would increase the complexity and increase overhead within the system. However, padding concurrent memory accesses would not be as imperative with such a change. This would presumably lead to better native performance from the FFTW routines.

The Clovertown architecture suffers from the single interconnect between the chips. As seen, the scaling is good from one to two threads but does not exhibit speedup near this as the number of threads is increased to eight. Once the transform overflows the size of the cache, the performance of eight threads becomes almost identical to that of two threads. One method of fixing this would be to increase the amount of memory bandwidth available to every core. As it stands now, accessing data from the DRAM is an expensive operation when all cores are fighting for it.

The Clovertown and Niagara II are seen to offer very different performance patterns. The Niagara sees very poor single threaded performance. This is due to the lower-powered cores and also the need to hide latency with threading. However, it scales very well as both the problem size and the thread count are increased. On the opposite end, the Clovertown has respectable single-threaded performance. While it does scale well to two threads, additional threading does not have as much of a positive performance impact. The added contention on the bus to main memory proves to be very damaging as the problem size begins to overflow the shared L2 cache.

7 Conclusion

We have shown unprecedented performance results in FFT on GPU by exploiting Cooley-Tukey framework to fit the Fourier transform algorithm to the hardware capabilities.

References

COOLEY, J. W., AND TUKEY, J. W. 1965. An algorithm for the machine calculation of complex Fourier series, *Mathematics of Computation 19*, 90, 297–301.

BAILEY, D. H. 1988. A High-Performance FFT Algorithm for Vector Supercomputers, *International Journal of Supercomputer Applications 2*, 1, 82–87.

BAILEY, D. H. 1990. FFTs in External or Hierarchical Memory, *Journal of Supercomputing 4*, 1, 23–35.

BLUESTEIN, L. 1970. A linear filtering approach to the computation of discrete Fourier transform, *IEEE Transactions on Audio and Electroacoustics 18*, 4, 451–455.

CHOW, A.C., FOSUM, G.C., AND BROKENSHIRE, D. A. 2005. A Programming Example: Large FFT on the Cell Broadband Engine, *Proc. 2005 Global Signal Processing Expo.*

DUHAMEL, P., AND VETTERLI, M. 1990. Fast fourier transforms: a tutorial review and a state of the art, *Signal Processing 19*, 4, 259–299.

EDELMAN, A., MCCORQUODALE, P., AND TOLEDO, S. 1999. The Future Fast Fourier Transform? *SIAM Journal on Scientific Computing 20*, 3, 1094–1114.

GOOD, I. J. 1958. The interaction algorithm and practical Fourier analysis, *Journal of the Royal Statistical Society*, *Series B (Methodological) 20*, 2, 361–372.

GOVINDARAJU, N., LARSEN, S., GRAY, J., AND MANOCHA, D. 2006. A memory model for scientific algorithms on graphics processors, *SC'06*.

LLOYD D. B., BOYD C., AND GOVINDARAJU, N. 2008. Fast Computation of General Fourier Transforms on GPUs.

RADER, C. M. 1968. Discrete Fourier transforms when the number of data samples is prime, *Proc. IEEE 56*, 1107–1108.

NICKOLLS, J. 2008. *Personal communication.*

WILLIAMS, S., PATTERNSON, D., OLIKER, L., SHALF, L. AND YELICK, K. 2008. The Roofline Model: A pedagogical tool for auto-tuning kernels on multicore architectures, *Hot Chips,* to appear.

MORELAND, K., AND ANGEL, E. 2003. The FFT on a GPU, *Graphics Hardware'03*.

MCCOOL, M., WADLEIGH, K., HENDERSON, B., AND LIN, H.-Y. 2006. Performance Evaluation of GPUs Using the RapidMind Development Platform, RapidMind White Paper.

NVIDIA, 2007. *CUDA CUFFT Library, V1.1*, October 2007.

SEGAL, M., AND PEERCY, M. 2006. A Performance-Oriented Data Parallel Virtual Machine for GPUs, *SIGGRAPH 2006 Sketch*.

SPITZER, J. 2003. Implementing a GPU-Efficient FFT, *SIGGRAPH GPGPU Course*, 2003.

VOLKOV V., AND DEMMEL, J. 2008. LU, QR and Cholesky factorizations using vector capabilities of GPUs, Technical Report No. UCB/EECS-2008-49, EECS Department, University of California, Berkeley, May 13, 2008. (Also LAPACK Working Note 202.).