Experiences with Mapping Non-linear Memory Access Patterns into GPUs

Eladio Gutierrez, Sergio Romero, Maria A. Trenas, and Oscar Plata

Department of Computer Architecture
University of Malaga, Spain
{eladio,sromero,maria,oplata}@uma.es

Abstract. Modern Graphics Processing Units (GPU) are very powerful computational systems on a chip. For this reason there is a growing interest in using these units as general purpose hardware accelerators (GPGPU). To facilitate the programming of general purpose applications, NVIDIA introduced the CUDA programming environment. CUDA provides a simplified abstraction of the underlying complex GPU architecture, so as a number of critical optimizations must be applied to the code in order to get maximum performance. In this paper we discuss our experience in porting an application kernel to the GPU, and all classes of design decisions we adopted in order to obtain maximum performance.

1 Introduction

Driven by the huge computing demand of the graphics applications, Graphics Processing Units (GPU) have become highly parallel, multithreaded and many-core processors. Modern GPUs deliver a very large amount of raw performance that have drawn attention to the scientific community, with a growing interest in using these units to boost the performance of their compute-intensive applications. That is, to use the GPUs as general-purpose hardware accelerators (General-Purpose Computation on GPUs, or GPGPU [2]).

Developing GPGPU codes using the conventional graphics programming APIs is a very hard task and with many limitations. This situation motivated the development of general parallel programming environments for GPUs [11,12]. NVIDIA CUDA (Compute Unified Device Architecture) [11], one of the most widespread models, is built around a massively parallel SIMT (Single-Instruction, Multiple-Thread) execution model, supported by the NVIDIA GPU architecture [7], and provides a shared-memory, multi-threaded architectural model for general-purpose GPU programming [10].

CUDA provides a convenient and successful model at programming scalable multi-threaded many-core GPUs, across various problem domains [5]. However, the simplified abstraction that CUDA model provides does not permit to extract maximum performance from the underlying GPU physical architecture without applying a set of optimizations to the parallel code [8,13]. We can distinguish two classes of optimizations. The first class corresponds to techniques that fall within

G. Allen et al. (Eds.): ICCS 2009, Part I, LNCS 5544, pp. 924–933, 2009.

the programming model, that is, those that improve the use of the architectural resources defined at CUDA level. The second class includes those optimizations that fall outside the programming model. We consider in this class techniques at a level lower than CUDA, that is, that must be included in the parallel execution implementation of the programming model.

This paper discusses our experience in porting application kernels to a GPU accelerator, with the aim of obtaining maximum performance. In order to have enough room for optimization, we have selected as a working example a kernel showing non-linear access patterns to memory, the fast Fourier transform (FFT). We will show that if the optimization efforts are only within the CUDA model, the obtained performance is much lower than the expected peak one. We have to resort to additional low-level techniques in order to improve significantly the resulting performance. An important issue is that these techniques are hard to apply and very dependent on the kernel computational structure. A final issue we also analyzed refers to the algorithm chosen to implement the kernel.

Due to its interest, several contributions can be found in the literature focused on porting FFT algorithms to graphics processing units [1,3,6,9,14]. More recently works about CUDA implementations report higher performance [4,15]. This is accomplished by a much more efficient use of GPU resources, through the application of many optimization techniques (CUDA level and low level). The implementation described in [4] behaves specially well. They use a different algorithm for FFT, a hierarchical Stockham.

2 CUDA Programming Model

NVIDIA CUDA is both a hardware and software architecture for issuing and managing computations on the GPU, making it to operate as a truly generic data-parallel computing device. An extension to the C programming language is provided in order to develop source codes.

From the hardware viewpoint, the GPU device consists of a set of SIMT multiprocessors each one containing several processing elements. Different memory spaces are available. The global device memory is a unique space accessible by all multiprocessors, acting as the main device memory with a large capacity. Besides, each multiprocessor owns a private on-chip memory, called shared memory or parallel data cache, of a smaller size and lower access latency than the global memory. A shared memory can be only accessed by the multiprocessor that owns it. In addition, there are other addressing spaces for specific purposes.

CUDA execution model is based on a hierarchy of abstraction layers: grids, blocks, warps and threads. The thread is the basic execution unit that is actually mapped onto one processor. A block is a batch of threads cooperating together in one multiprocessor and hence all threads in a block share the shared memory. A grid is composed by several blocks, and because there can be more blocks than multiprocessors, different blocks of a grid are scheduled among the multiprocessors. In turn, a warp is a group of threads executing in an SIMT way, so threads of a same block are scheduled in a given multiprocessor warp by warp.

Two kinds of codes are considered in the CUDA model: those executed by the CPU (host side) and those executed by the GPU, called kernel codes. The CPU is responsible of transferring data between host and device memories as well as invoking the kernel code, setting the grid and block dimensions. Memory accesses and synchronization schemes are the most important aspects to take into account. Warp addresses issued by SIMT memory access instructions may be grouped thus obtaining a high memory bandwidth. This is known as coalescing condition. Otherwise, access will be serialized and the resulting latency will be difficult to hide with the execution of other warps of the same block.

3 Experiences in Optimizing the FFT in CUDA

We have selected as a benchmark a kernel code showing non-linear access patterns to memory: the Fast Fourier Transform (FFT). Basically, the FFT follows a *divide and conquer* strategy in order to reduce the computational complexity of the discrete Fourier transform (DFT), which provides a discrete frequency-domain representation $X[k]$ from a discrete time-domain signal $x[n]$. For a 1-dimensional signal of N samples, DFT is defined by the following pair of transformations (forward and inverse): $X = DFT(x)\colon\ X[k] = \sum_{n=0}^{N-1} x[n]W_N^{nk}, \quad 0 \leq k < N$, and $x = IDFT(X)\colon\ x[n] = \frac{1}{N}\sum_{k=0}^{N-1} X[k]W_N^{-kn}, 0 \leq n < N$, where the powers of $W_N = e^{-j\frac{2\pi}{N}}$ are the so-called twiddle factors.

The design decisions to develop an efficient GPU implementation of a kernel code like FFT may be classified into three levels:

- *Algorithm level*: It refers to the algorithm chosen to implement the kernel. The basic strategy is the well-know Cooley-Tukey decomposition, but some other strategies, like the Stockham approach, have been shown to behave better in SIMD architectures. In addition, the selection of the radix parameter has strong influence in the performance.
- *CUDA level*: Once the algorithm has been configured, it must be mapped into the CUDA architecture. The resulting performance depends strongly on two main issues: parallelism extracted and memory hierarchy exploitation. Both issues are closely related to platform features, and are clearly influenced by the problem memory access patterns.
- *Code level*: The CUDA architecture hides many low-level details of the underlying hardware platform. This way, an important fraction of the overall performance may depend on a series of low level tricks that would help the compiler to generate an efficient object code.

Algorithm level. In this paper we have considered a radix-2, DIT (Decimation In Time), Cooley-Tukey implementation. Although this algorithm requires an initial bit-reversal stage, however it could be used in signal transformations where such bit-reversal operation is not required (e.g., Walsh). This is not the case of the Stockham method, which is auto-sorted.

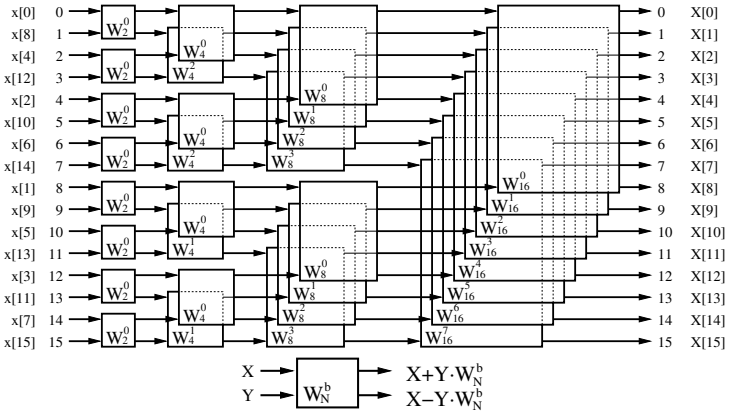

Fig. 1. Radix-2 decimation-in time FFT in terms of butterfly operators

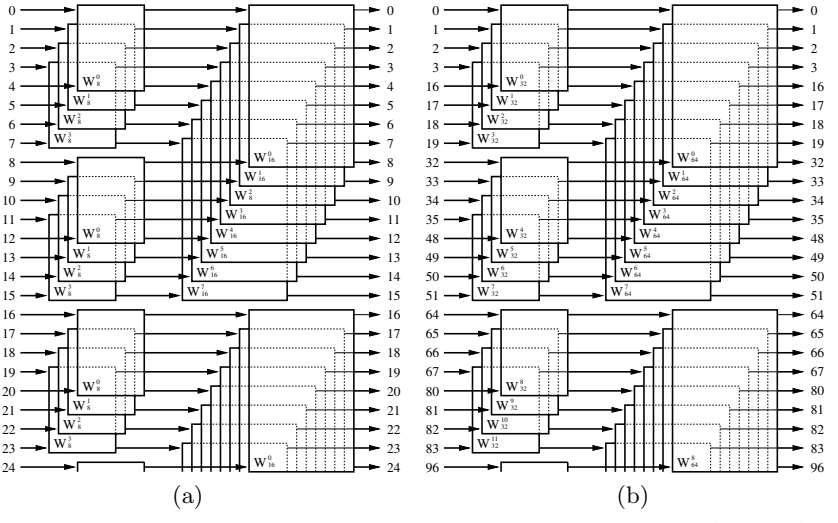

Fig. 2. (a) Computing 3^{rd} and 4^{th} stages of the FFT; (b) computing 5^{th} and 6^{th} stages of the FFT using the pattern of 3^{rd} and 4^{th} stages over a properly permuted input

The radix-2, DIT, Cooley-Tukey FFT organizes the DFT computations, as shown in Fig. 1, in terms of basic blocks, known as butterflies. The computation is carried out along $\log_2 N$ stages being computed N coefficients per stage. Before the first stage, input coefficients must be bit reversed (omitted in the figure).

Focusing on memory locality, we observe that if the input coefficients are located into consecutive memory positions, the reference patterns of higher stages will exhibit poorer locality features than the lower ones. In addition, if the input coefficients are permuted properly, it is possible to carry out one of the stages using the access pattern of another, simply by using the corresponding twiddle factors. Such an equivalence is depicted in Fig. 2 showing how 5^{th} and 6^{th}

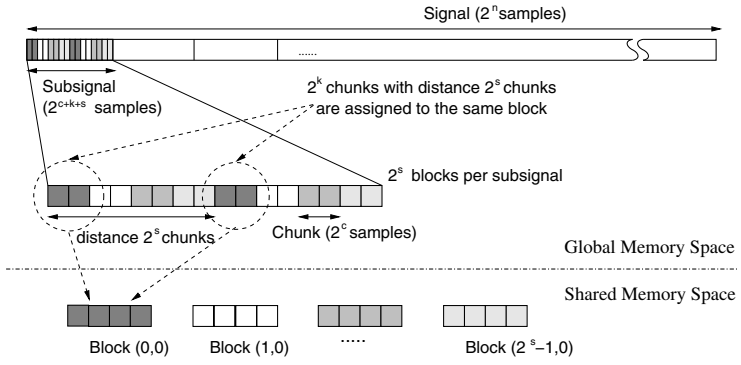

Fig. 3. Mapping of the input signal from global to shared memory spaces

stages can be performed with the access pattern of the 3^{rd} and 4^{th} ones, after permuting the coefficients. This mapping is denoted as: $[5:6] \rightarrow [3:4]$.

CUDA level. The goal in the CUDA version is to obtain a high degree of parallelism taking into account system constrains, specifically those related to the memory hierarchy. The basic idea consists of mapping input signal samples placed in global (device) memory into the data parallel cache (shared memory), performing all possible computations with these local data and then copying the updated samples back to the global memory. This process may be repeated with different mapping functions until all stages are done.

Fig. 3 depicts how the input signal is repeatedly mapped from global memory into shared memory spaces. This mapping try to maximize coalesced accesses to global memory. The figure also shows a number of parameters defined to describe our CUDA version, named as ctFFT (from Cooley-Tukey FFT) from now on. First, we consider that the input signal size is a power of two (2^n samples). This signal is subdivided into equal-sized subsignals, which are further subdivided into fixed-size chunks of 2^c samples. A set of 2^k chunks, separated among themselves a distance of 2^s chunks, are grouped and assigned to the same CUDA block. So, each subsignal contains a total of 2^s chunk blocks, or a total of $2^{(s+k+c)}$ samples. Hence, the size of each CUDA block is of $2^{(c+k)}$ samples, and the complete input signal contains a total of $2^{n-(c+k)}$ such blocks. There is a size restriction for CUDA blocks, as each one will be processed in parallel in a single GPU multiprocessor, so it must be fitted completely in the shared memory.

With the above data mapping strategy, ctFTT proceeds as a series of synchronized phases, as follows:

- **Initial Phase**: Processing of CUDA blocks composed of consecutive chunks ($s = 0$). The first $k + c$ FFT stages can be accomplished with these data blocks ($[0 : (k + c - 1)] \rightarrow [0 : (k + c - 1)]$).
- **Intermediate Phase 1**: After finishing these first $k + c$ stages, we can continue with the remaining FFT stages. These stages should not overlap

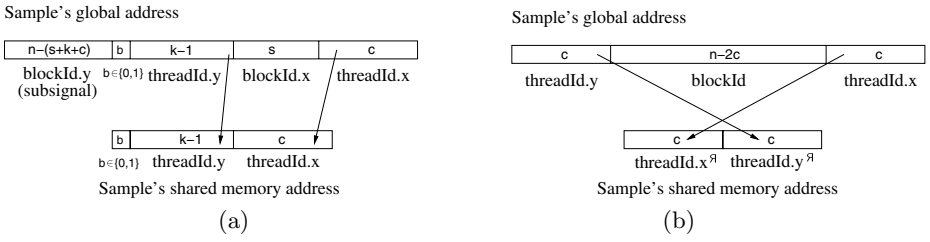

Fig. 4. (a) Bit-block addressing defining the mapping of the input signal from global to shared spaces, and (b) addressing for the bit reversal operation

the previous ones already processed, so we must select the suitable CUDA blocks to be transferred to shared memory. Stage overlapping is avoided if we select blocks corresponding to values of s that are integer multiple of k. So, in the first intermediate phase, we process CUDA blocks composed of chunks separated k chunks, that is, $s = k$. These data allow to compute the following k FFT stages: $[s + c : s + c + k - 1] \rightarrow [c : c + k - 1] = [k + c : 2k + c - 1] \rightarrow [c : c + k - 1]$.

- **Intermediate Phase i**: In general, the above procedure is repeated. Now, we take CUDA blocks corresponding to $s = ik$, that allow to compute a bunch of k FFT stages: $[s + c : s + c + k - 1] \rightarrow [c : c + k - 1] = [ik + c : (i + 1)k + c - 1] \rightarrow [c : c + k - 1]$.
- **Final Phase**: The last FFT stages to be computed could be less than k. If the total number of intermediate phases is P, in this final phase the next FFT stages are computed: $[s + c : n] \rightarrow [c : n - s] = [(P + 1)k + c : n] \rightarrow [c : n - (P + 1)k]$. The number of intermediate phases can be calculated as $P = \lfloor (n - (k + c))/k \rfloor$, if this number is positive.

ctFFT organizes parallel execution by assigning to each thread two main tasks: (i) copying of two signal samples from global to shared spaces, and (ii) processing of a single FFT butterfly, accessing two signal samples stored in shared memory. So, the total number of threads is $2^n/2 = 2^{n-1}$ (radix-2). These threads are organized in a grid of $(nBlock.x, nBlock.y)$ thread blocks, and each of these thread blocks groups $(nThreads.x, nThreads.y)$ threads. With this grouping of threads, the data mapping shown if Fig. 3 can be defined by mapping bit-blocks of the memory addresses, as depicted in Fig. 4 (a). The b bit in both addresses allows to distinguish between the two signal samples assigned to the same thread, and it is in use during the copy-in and copy-out operations (global to shared and shared to global). During the computation of a butterfly in the i FFT stage, the thread accesses the corresponding signal samples in shared memory by inserting a bit 0 or 1 in the i bit of the shared memory address composed of the bit-blocks $threadIdx.y|threadIdx.x$. Note that if c is larger than the number of threads in a single warp, then coalescing condition is completely fulfilled.

As a DIT implementation is considered here, the initial bit reversal operation applied to the input signal is required. Fig. 4 (b) shows how this operation is implemented in ctFFT. Basically, it consists in a coalesced copy of the input

signal samples to the shared memory, storing them in positions given by the bit reversal of the thread bit-block identifiers. Afterwards, these samples are copied back to the corresponding positions in global memory (not in-place).

Code level. Among the low-level optimization techniques, we can highlight four with high impact on the code performance: loop unrolling, padding, constant propagation and thread synchronization. Padding is used to reduce shared memory bank conflicts. Constant propagation avoids unnecessary arithmetic instructions, specially when computing padding functions. Additional non-mandatory thread barrier synchronizations are beneficial. For example, after completing global memory accesses. We consider that most of these optimizations should be implemented in the CUDA compiler, but our experience shows that without the help of the programmer, the compiler fail to apply many of these techniques.

4 Experimental Evaluation

In this section we experimentally evaluate ctFFT. All experiments were conducted on a NVIDIA GeForce 280 GTX GPU, which includes 30 multiprocessors of eight processors each (240 cores in total), working at 1.3GHz with a device (global) memory of 1 GB. Each multiprocessor has a 16KB parallel data cache (shared memory). Codes were written in C using the version 1.0 of NVIDIA CUDA [11]. NVIDIA provides its own FFT library (CUFFT), that we take as a reference in order to assess the quality of our optimized CUDA version (ctFFT).

Fig. 5 (top) shows the performance of ctFFT compared to CUFFT. These plots correspond to the CUDA version discussed in the previous section, considering only CUDA level optimizations, specially, coalescing (no low-level). According to the CUFFT interface, two dimensionality parameters are taken into consideration: the signal size and the number of signals of the given size to be processed (known as a batch of signals). The number of FLOPS is calculated using the equation $5bN \log_2 N$, for a batch of b signals of N samples per signal.

From the plots it can be seen that in cases of batches of large signals, our ctFFT outperforms CUFFT. However, CUFFT is better for a large number of batches of small signals. In addition, ctFFT allows to process larger signals than CUFFT. The CUFFT library is unable to perform the transform beyond 2^{23} samples [11] whereas our implementation can manage up to 2^{26} samples, making a better exploitation of the available device memory. Fig. 5 bottom summarizes all these results (GFLOPS in the plot at the left, relative GFLOPS in the plot at the right). The bottom-right plot allows to determine for which signal configurations ctFFT outperforms CUFFT.

The above results show that the best performance attained is almost 40 GFLOPS, which is much lower than the peak performance of the GPU platform. And there are no other relevant performance strategies at CUDA level that we can use to further improve the parallel code. So, only two alternatives remain, either change the original FFT algorithm, that maps better into the

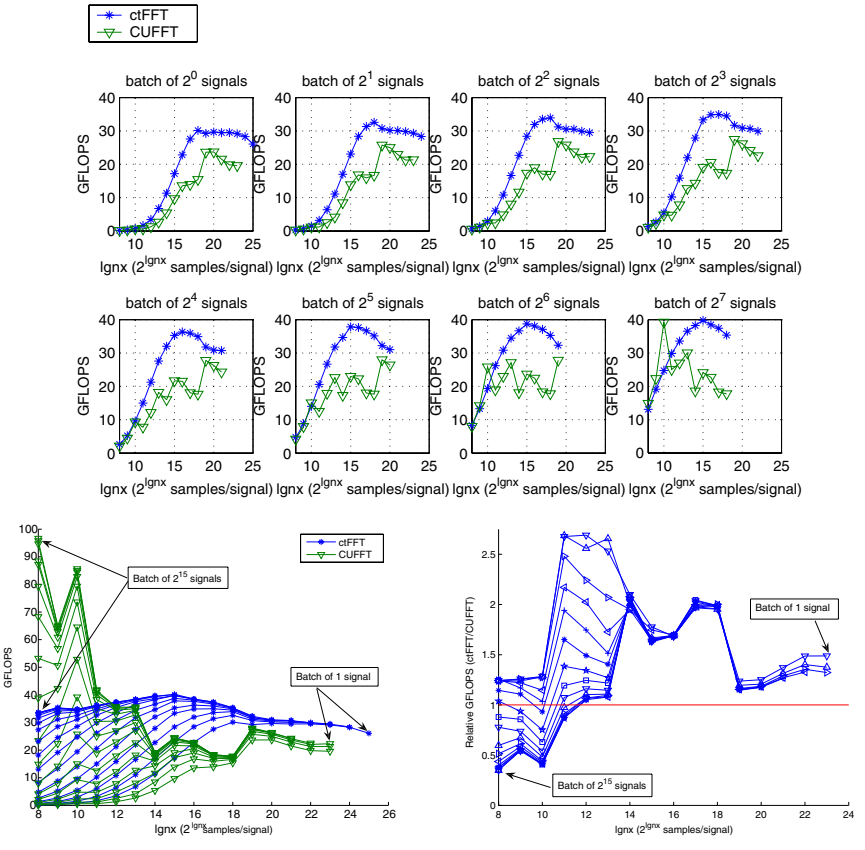

Fig. 5. Performance in GFLOPS of ctFFT compared to CUFFT

GPU architecture, or apply low-level optimization techniques. These optimizations fall outside the CUDA programming model, are dependent on the specific CUDA code at hand, and represent a hard effort to apply.

Table 1 (a) illustrates the change in performance that ctFFT underwent when an incremental series of low-level optimization techniques were applied. These figures were obtained for a batch of 2^{15} signals of 2^9 samples per signal, in such a way that each signal fits completely in the shared memory of a CUDA block. This table shows a broad range of achieved performance results, from 17.5 to 229 GFLOPS, depending on different optimizations and algorithms used. At present, the best known FFT implementation on CUDA [4] performs a peak of 300 GFLOPS (based on considerable hand-coded low-level optimizations). The first column in the table corresponds to a ctFFT version where all computations are carried out over the global memory, using coalesced accesses (shared memory is not used). The second column is the CUDA version analyzed in the previous section. This version uses the shared memory as a cache of the global one, resulting in about 2× performance improvement. The next two columns

Table 1. Incremental performance improvements: (a) applying low-level optimizations, (b) for different memory access patterns (batch of 2^{15} signals of 2^9 samples)

(a)

	ctFFT (global)	ctFFT	+ unrolling	+ padding	+ radix-4	+ radix-8	+ constant propagation
GFLOPS	17.5	35.1	59.5	63.5	106	116	130
Incoherent ld/st	0	0	0	0	0	0	0
Warp serialize	0	$806{\cdot}10^3$	$668{\cdot}10^3$	$649{\cdot}10^3$	$428{\cdot}10^3$	$255{\cdot}10^3$	$221{\cdot}10^3$

(b)

	DIT, no bit reversal	DIF, wo/ synch	DIF, w/ synch	Stockham
GFLOPS	142	167	199	229
Incoherent ld/st	0	0	0	0
Warp serialize	$189{\cdot}10^3$	$211{\cdot}10^3$	$170{\cdot}10^3$	$95{\cdot}10^3$

add loop unrolling and padding low-level optimizations to ctFFT. The performance improvement due to padding is lower than expected because the padding function must be simple in order to not introducing too much overhead. The use of a higher radix (than 2), as shown in the next two columns, allows to increase the performance even more. The reason for this behaviour is a more intensive (re)use of the processor registers, that represent the fastest level of the memory hierarchy. For radices beyond 8, the performance degrades due to two effects. First, the fixed number of registers limits the number of active threads per multiprocessor (this is called occupancy). Second, for a higher radix, the number of threads per block decreases, reducing the opportunities of hiding memory latency via warp scheduling. Constant values resulting from padding functions can be precomputed and propagated directly in the code. This optimization has an important impact in the performance, as shown in the corresponding column.

Table 1 (b) corresponds to other four implementations we have developed for the same signal configuration, and also including all the previously discussed low-level optimizations. This table show the impact in performance of various memory access patterns. The first one is ctFFT but with the bit reversal stage omitted. The second and third columns correspond to the radix-8, DIF (Decimation In Frequency) version of the Cooley-Tukey algorithm. The difference between them is the inclusion or not of additional non-mandatory thread barrier synchronizations. Finally, the fourth column corresponds to the Stockham algorithm, developed using similar strategies than discussed for ctFFT.

Basically, the performance of ctFFT is modest due to two main reasons: the cost of the bit reversal operation, and the loss of parallelism due to conflicts in shared memory banks. Other interesting measurements included in tables are the number of incoherent loads/stores and the warp serialize parameter (obtained by activating `CUDA_PROFILE`). The first one shows the number of non-coalesced accesses to global memory. In all evaluated cases, coalescing was perfect. Warp serialize represents the number of threads serialized due to shared memory bank conflicts. Note that the low-level optimizations reduces significantly this number, with a clear effect in performance. But also the memory access pattern associated to the algorithm (Cooley-Tukey/Stockham, DIT/DIF, radix-2/-4/-8) has an important impact. The lowest number of warp conflicts correspond to Stockham version (radix-8, DIF).

5 Conclusions

This paper discusses our experience in porting application kernels to GPU accelerators using CUDA. In particular, a FFT benchmark was chosen due to its non-linear memory access patterns that allow to play with many design issues when mapping it to the complex GPU architecture. According to our experience, developers should take into account three classes of design issues: at algorithm level, at CUDA level and at code level (low level). In any level we may observe strong impact in performance. We specially highlight low-level optimizations, that may increase the performance of the CUDA version by one order of magnitude. However, the bad part is that these techniques are very dependent on the specific CUDA code and represent a hard effort to apply. New programming environments should include technology to automatize, at least partially, these essential and high-impact low-level optimizations.

References

1. Fialka, O., Cadik, M.: FFT and Convolution Performance in Image Filtering on GPU. In: 10th Int'l. Conf. on Information Visualization (2006)
2. General-Purpose Computation Using Graphics Hardware, http://www.gpgpu.org
3. Govindaraju, N.K., Larsen, S., Gray, J., Manocha, D.: A Memory Model for Scientific Algorithms on Graphics Processors. In: ACM Int. Conf. Supercomputing (2006)
4. Govindaraju, N.K., Lloyd, B., Dotsenko, Y., Smith, B., Manferdelli, J.: High Performance Discrete Fourier Transforms on Graphics Processors. In: Int'l. Conf. for High Performance Computing, Networking, Storage and Analysis (SC 2008) (2008)
5. Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., Volkov, V.: Parallel Computing Experiences with CUDA. IEEE Micro. 28(4), 13–27 (2008)
6. Jansen, T., von Rymon-Lipinski, B., Hanssen, N., Keeve, E.: Fourier volume rendering on the GPU using a split-stream FFT. In: Vision, Modeling, and Visualization Workshop (2004)
7. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro. 28(2), 39–55 (2008)
8. Manikandan, M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In: ACM Int'l. Conf. on Supercomputing (2008)
9. Moreland, K., Angel, E.: The FFT on a GPU. In: ACM Conf. Graph. Hardware (2003)
10. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable Parallel Programming with CUDA. ACM Queue 6(2), 40–53 (2008)
11. NVIDIA CUDA (2008), http://developer.nvidia.com/object/cuda.html
12. The OpenCL Specification, Ver. 1.0.29, Khronos OpenCL Working Group, http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf
13. Petit, E., Matz, S., Bodin, F.: Data Transfer Optimization in Scientific Applications for GPU based Acceleration. In: Workshop Compilers for Parallel Computers (2007)
14. Sumanaweera, T., Liu, D.: Medical Image Reconstruction with the FFT. GPU Gems 2, 765–784 (2005)
15. Volkov, V., Kazian, B.: Fitting FFT onto the G80 Architecture (2008), http://www.cs.berkeley.edu/~kubitron/courses/cs258-S08/projects/reports/project6_report.pdf