Locality-Improved FFT Implementation on a Graphics Processor

SERGIO ROMERO, MARIA A. TRENAS, ELADIO GUTIERREZ, EMILIO L. ZAPATA
Department of Computer Architecture
University of Málaga
29071 Málaga, SPAIN

Abstract: The growing computational power of modern graphics processing units is making them very suitable for general purpose computing. These commodity processors operate generally as parallel SIMD platforms and, among other factors, the effectiveness of the codes is subject to a right exploitation of the underlying memory hierarchy. This paper deals with the implementation of the Fast Fourier Transform on a novel graphics architecture offered recently by NVIDIA. Such an implementation takes into consideration memory reference locality issues, that are crucial when pursuing a high degree of parallelism, that is, a good occupancy of the processing elements. The proposed implementation has been tested and compared to the manufacturer's own implementation.

Key–Words: Fast Fourier Transform (FFT), Graphics Processing Unit (GPU), memory locality, parallel processing

1 Introduction

The Fast Fourier Transform (FFT) constitutes nowadays a cornerstone for many algorithms and applications in the context of signal processing. 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):

$$X = DFT(x)\colon\ X[k] = \sum_{n=0}^{N-1} x[n] W_N^{nk}, \quad 0 \le k < N$$

$$x = IDFT(X)\colon\ x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] W_N^{-kn}, 0 \le n < N$$

where the powers of $W_N = e^{-j\frac{2\pi}{N}}$ are the so-called twiddle factors.

The 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. This way, the computational complexity is reduced to $\mathcal{O}(N \log_2(N))$ instead of $\mathcal{O}(N \times N)$ as inferred directly from the DFT definition.

Several configurational issues have been preset in Fig. 1. This configuration is known as *radix two* because butterflies operate on two inputs generating two transformed coefficients. Also input coefficients are permuted in bit reversal order before the first stage

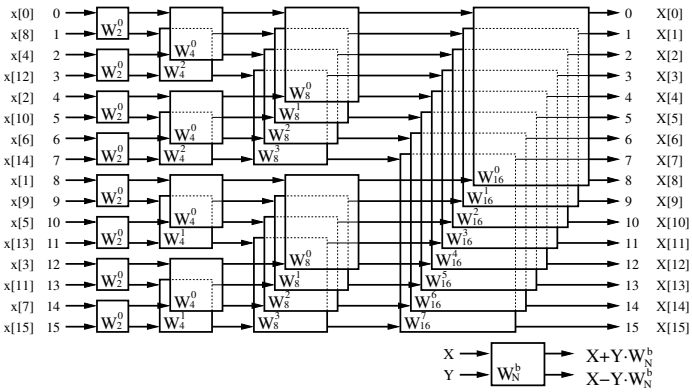

Figure 1: Radix-2 decimation-in time FFT in terms of butterfly operators.

with the purpose of obtaining the right output arrangement. Such a rearrangement in time domain gives rise to the denomination *decimation-in-time* algorithm. This configuration is used the rest of the paper.

From the viewpoint of memory reference locality, we can 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, we must remark that 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} stages can be performed with the access pattern of the 3^{rd} and 4^{th} ones, after permuting the coefficients.

For subsequent use, we will denote $L_{(N,j,i)}(x)$

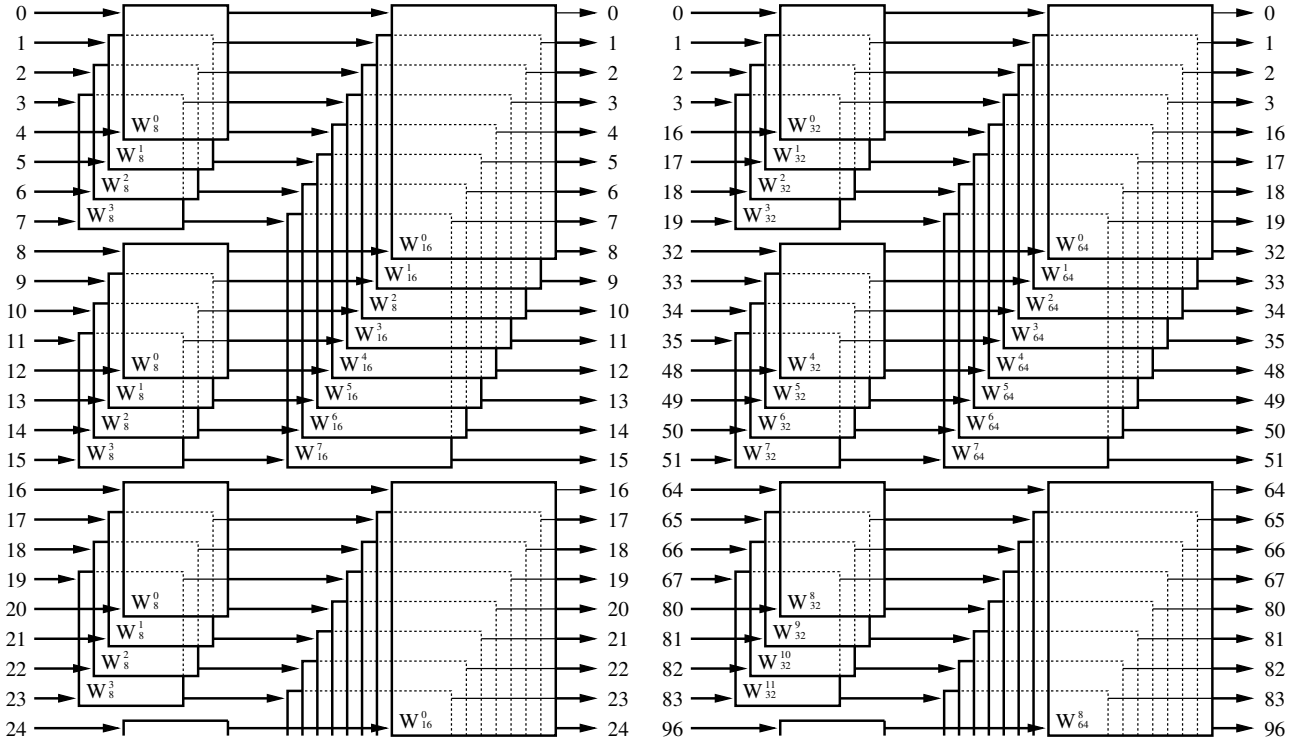

Figure 2: Computing 3^{rd} and 4^{th} stages of the FFT (a); computing 5^{th} and 6^{th} stages of the FFT using the pattern of 3^{rd} and 4^{th} stages over a properly permuted input (b).

as the computation of j-th stage of a N-sample signal, but using the access pattern of the i-th stage (excluding the permutation) and $L_{(N,i)}(x) = L_{(N,i,i)}(x)$ the computation of the i-th stage with the proper pattern and twiddle factors. This way we can write the full FFT computation as $X = FFT(x) = L_{(N,s-1)}(...(L_{(N,1)}(L_{(N,0)}(P(x))))...)$, assuming that the number of samples is $N = 2^s$, and P represents the bit reversal permutation of the signal.

2 Related work

The FFT represents a floating-point computationally intensive algorithm whose generalized application makes it adequate for being accelerated on graphics platforms. Due to its interest, several contributions can be found in the literature of the last years focused on porting FFT algorithms to graphics processing units. In [8] very basic ideas of how to implement the FFT algorithm are collected. In [6], implementations of the FFT in the context of image processing applications are presented using a shadower model. Also in other specific contexts FFT has been developed on graphics hardware, like [9, 1, 4]. A discussion about the FFT implementation, together with other algorithms, is found in [3]. This work tries to exploit the GPU memory hierarchy in order to improve the performance of the implementations.

3 The GPU programming model

A GPU (Graphic Processor Unit) is a device specialized in algorithms such as graphics rendering involving very intensive and highly parallel computations. These devices are nowadays implemented as a set of multiprocessors with a Single Instruction Multiple Data (SIMD) architecture. Due to their high computational power, these GPUs are used both for graphics and general purpose processing. In this scope, they operate as a coprocessor, or hardware accelerator, to the main CPU, or host.

NVIDIA® has recently presented its Compute Unified Device Architecture (CUDA™), as a both hardware and software architecture for issuing and managing computations on the GPU as a truly generic data-parallel computing device with a very high level of parallelism. An extension to the C programming language is provided in order to develop source codes.

CUDA programming model is based on a hierarchy of abstraction layers: grids, blocks, warps and threads. All threads in a block behave as an SIMD, whereas different blocks of a grid are scheduled among the set of multiprocessors by the API runtime in a transparent way. Programmers spec-

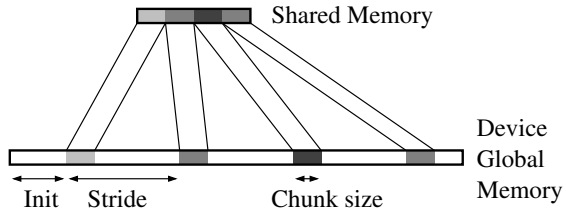

Figure 3: Data transfer pattern between device and shared memory of `copy_in/copy_out` operations.

ify the number and shape (1D, 2D or 3D) of some of such levels, without the additional charge of coding strategies to balance the workload among the actual hardware configuration. However, some limitations exist, among others: the maximum number of threads in a block is 512, a block of threads is executed in one multiprocessor (manufacturer recommends the use of a large number of blocks), memory accesses for all threads in a warp must be coalesced, and only threads in a block can be synchronized at the device side, while the synchronization of different blocks of threads must be explicitly done by the host.

According to this model, an application running on the host invokes a unique kernel code that will be executed for each thread at the device side, but operating over different data sets.

4 FFT implementation

In this section we analyze an FFT implementation using the programming model previously described. The goal 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 coefficients 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 coefficients back to the global memory. This process may be repeated with different mapping functions until all stages are done.

In order to be more precise we firstly introduce some useful functions describing the FFT implementations under study. These functions represent data transfers and transformations accomplished in a single shared memory.

Function `copy_in(ii,nc,sz,st)` copies a subset of signal coefficients from the device memory, into consecutive positions of the shared memory, adding a padding when necessary to avoid memory bank conflicts. Its behaviour is depicted in Fig. 3 starting from the `ii`-th coefficient and copying `nc` chunks of size `sc` coefficients separated by a stride `st` as shown in Fig. 3. Symmetrically,

GPU side, one block

```
copy_in(0,N,1,1);
for (i=0; i<s; i++) {
  syncthreads();
  fftL(i,i);
}
copy_out(0,N,1,1);
```

Figure 4: FFT of a signal fitting the shared memory.

`copy_out(ii,nc,sz,st)` copies coefficients from shared memory back to the device memory.

Function `fftL(i,j)` corresponds to the application of the operator $L_{(N,i,j)}(x)$ as described in section 1. Such a function is intended to be applied to coefficient x, previously transferred to shared memory, and it operates in-place. The number of blocks of threads is $\frac{N}{2^r}$, where 2^r is the number of coefficients copied into shared memory. As this function is executed in a SIMD way, the b-th thread computes the b-th butterfly transformation. Twiddle factors for this case are determined by the b-th butterfly of the j-th FFT stage, whereas the coefficients to be transformed are those involved in the b-th butterfly of the i-th FFT stage.

Let us analyze the naive case when the whole input signal fits into the shared memory of one SIMD multiprocessor ($N \leq 2^r$), whose implementation is shown in Fig. 4. After a bit reversal permutation, coefficients are transferred to the shared memory, then r invocations of `fftL` are executed and finally coefficients are returned to the device memory. Note that as all the threads belong to the only block, global synchronization can be performed among threads. Also, the original FFT scheme is applied locally (`fftL` arguments are equal).

Now, we will consider the generic case consisting on a signal whose size exceeds the available shared memory of a multiprocessor ($N > 2^r$). In this case several multiprocessors are involved, meaning that different blocks of threads must collaborate to perform the FFT. The first approach to be analyzed is a straightforward solution, but it exploits barely the locality features of memory access pattern. As threads of different blocks cannot be synchronized within kernel code, required synchronization points must be carried out in the host side through successive kernel function invocations. Each kernel code invokes `copy_in` and `copy_out`. Between these two invocations, several `fftL` stages need to be performed. The larger the signal size, the larger the number of butterflies operators and also the larger the number of FFT stages. Due to the fixed size of shared memories, the input signal must be distributed among the blocks of

Host side	GPU side, block j
q=min(r,s);	
FirstKernel(q); -->	COPY_IN(j*2^q,1,2^q,1);
	for (i=1; i≤q; i++) {
	syncthreads();
	FFT_LEVEL(i,i);
	}
if (s>r) {	COPY_OUT(j*2^q,1,2^q,1);
q=s-r;	
NextKernel(q); -->	COPY_IN(j,2^q,1,2^r);
}	for (i=1; i≤q; i++) {
	syncthreads();
	FFT_LEVEL(i,i+r);
	}
	COPY_OUT(j,2^q,1,2^r);

Figure 5: Straightforward FFT implementation for large signals.

Host side	GPU side, block j
q=min(r,s);	
FFTKernel(q); -->	COPY_IN(j*2^q,1,2^q,1);
	for (i=1; i≤q; i++) {
	syncthreads();
	FFT_LEVEL(i,i);
	}
	COPY_OUT(j*2^q,1,2^q,1);
Transposition();-->	COPY_IN(...);
	TranspositionCore();
if (s>r) {	COPY_OUT(...);
q=s-r;	
FFTKernel(q); -->	COPY_IN(j*2^q,1,2^q,1);
}	for (i=0; i≤q; i++) {
	syncthreads();
	FFT_LEVEL(i,i+r);
	}
	COPY_OUT(j*2^q,1,2^q,1);
Transposition();-->	COPY_IN(...);
	TranspositionCore();
	COPY_OUT(...);

Figure 6: FFT implementation for large signals using matrix transpositions.

threads. Thus, a number of blocks of threads equal to $\frac{N}{2^r}$ will work with their corresponding set of disjoint coefficients. Let us consider each block with a set of 2^r consecutive complex coefficients. This way the first r FFT stages can be performed independently of the work of other blocks. Nevertheless, threads within the block must be synchronized before every stage in order to ensure that its input coefficients are updated by the previous stage. The remainder FFT stages involve coefficients located at a distance larger than 2^r, that is, their copies are located on different shared memories, on different multiprocessors. In order to proceed forward, coefficients must be properly rearranged. This fact involves a `copy_out` and a host synchronization prior to continue with the next stages.

At this point, all output coefficients of the r-th FFT stage can be found in the device memory. Without loss of generality, let be $2^r \geq N/2^r$ ($2^s \leq 2^{2r}$), that is, the total number of FFT stages s is at most $2r$. As these r levels have been just computed, only r subsequent stages remain at most, and therefore only a new kernel invocation is needed. In general, $\frac{s}{r}$ kernel invocations will be required. Assigning the 2^r sequences of size 2^{s-r} coefficients with stride 2^s to different blocks (one sequence per block), all the $s-r$ remaining stages can be performed as shown in Fig.5.

Observe that an important fact affects adversely the performance of the second kernel call (`NextKernel`). As threads are scheduled in warps behaving like gangs of threads that execute the same SIMD instruction, the memory addressing mode must follow a specific pattern for an efficient execution. In the case of global memory, threads of a same warp must access to consecutive memory locations, otherwise accesses are serialized. This condition is called coalescing requirement. The approach of Fig. 5 suffers from this lack of coalescing because memory locations accessed by copy operations do not contain chunks of consecutive coefficients.

A well known solution to this problem is to store the input signal in a 2D matrix ($2^{s1} \times 2^{s2}$ with $s = s1 + s2$), 1D FFT is applied to every row (first $s1$ stages), then the matrix is transposed and finally 1D FFT is again applied to every row (last $s2$ stages). In order to apply correctly these last stages, a transformation of the transposed matrix is required as described in [5]. This step can be avoided if these 1D FFT stages use the corresponding twiddle factors of the original FFT higher stages as shown in Fig. 6. Note that input coefficients for the second invocation to the `FFTKernel` are now located on consecutive positions satisfying memory access coalescing demands, but this technique requires extra `copy_in/copy_out` operations for each transposition stage. Details of the transposition of a matrix can be found in [7], whose data transfers between device and shared memory also fulfill the coalescing requirements.

With the purpose of improving the data locality in the higher levels of the FFT of large signals, we propose the technique described as follows. The key idea consists of transferring chunks of consecutive coefficients with a given stride among them, allowing the application of higher FFT stages using lower FFT stage access patterns. This technique is depicted in Fig. 7. Observe that invocations to `NextKernel` are not preceded by any transposition and, what is more important, `copy_in/copy_out` operations meet the

Host side	GPU side, block j

```
q=min(r,s);
FirstKernel(q); --> COPY_IN(j*2^q,1,2^q,1);
                    for (i=1; i≤q; i++){
                      syncthreads();
                      FFT_LEVEL(i,i);
                    }
if (s>r) {          COPY_OUT(j*2^q,1,2^q,1);
  q=min(r/2,s-r);
  NextKernel(q); --> COPY_IN((j+j/2^q)*2^q,
}                           2^q,2^q,2^r);
                    for (i=r+1; i≤r+q; i++){
                      syncthreads();
                      FFT_LEVEL(i-q,i);
                    }
                    COPY_OUT((j+j/2^q)*2^q,
if (s>r+r/2) {              2^q,2^q,2^r);
  q=s-r-r/2;
  NextKernel(q); --> COPY_IN((j+j/2^q)*2^q,
}                       2^q,2^q,2^(r+r/2));
                    for (i=r+1; i≤r+q; i++){
                      syncthreads();
                      FFT_LEVEL(i-q,i+r/2);
                    }
                    COPY_OUT((j+j/2^q)*2^q,
                        2^q,2^q,2^(r+r/2));
```

Figure 7: Improved-locality FFT implementation for large signals.

coalescing condition. This way, on avoiding transposition stages, the number of memory transfer operations is significatively reduced. The number of higher FFT stages that can be mapped on lower ones depends on the number of chunks (nC), in particular $log_2(nC)$ stages. Moreover, the number of chunks depends on the size of the chunks, which is determined by the number of threads of a warp (coalescing condition). For this reason, the host must invoke `NextKernel` two times, half of the higher stages are performed in each one.

By way of illustration, let us consider the case of an FFT of an input signal whose size is 256 coefficients (8 FFT radix-2 stages), running on a GPU with 8 threads per block assembled in 4 threads per warp and a shared memory with room for 16 coefficients per block. With this configuration, the whole FFT can be decomposed into 16 block of 16 consecutive coefficients (after a bit reversal permutation) performing the four first FFT stages. Then, coefficients must be rearranged in order to proceed with the next stages. As warps are made of 4 threads, the chunk size is fixed to 4 consecutive coefficients, but pairs of coefficients separated 2^4 are required, so the stride is 16. Function `copy_in(4j,4,4,16)` collects all coefficients for the j-th block, enabling it to perform 5^{th} and 6^{th} stages using the access patterns of 3^{rd} and 4^{th} stages by means of `FFT_LEVEL(3,5)` and `FFT_LEVEL(4,6)`. This is the same example shown in Fig. 2. Note that 5^{th} and 6^{th} stages can not be remapped onto stages 1^{st} and 2^{nd} because of their shared memory access pattern. This involves that stages 7^{th} and 8^{th} must be performed after a new rearrangement of the coefficients (`copy_in(4j,4,4,64); FFT_LEVEL(3,7); FFT_LEVEL(4,8)`).

According to the hardware specifications of the target platform, the maximum number of threads per block is 512, the maximum number of threads per warp is 32 and the shared memory size is 8 Kbytes, so 1024 complex coefficients fit. First 10 FFT stages (r= 10) are performed in the invocation of `FirstKernel`, while only 5 higher stages can be done in each invocation to `NextKernel`. This fact is imposed by the chunk size. Since there are 32 chunks of 32 coefficients in 1024 coefficients, the 6^{th} stage is the first one onto which a higher stage can be mapped. A lower number of threads per warp allows `NextKernel` to perform more stages, however the degree of parallelism will decrease. In Fig. 7 the invocation to `FirstKernel` performs r stages whilst invocations to `NextKernel` perform at most $\frac{r}{2}$ stages.

5 Results

The locality improved strategy for the 1D complex FFT above discussed has been implemented and tested. Experiments have been conducted on a NVIDIA GeForce®8800GTX GPU, which includes 16 multiprocessors of eight processors, working at 1.35GHz with a device memory of 768MB. Each multiprocessor has a 8KB parallel data cache (shared memory). The latency for global memory is about 200 clock cycles, whereas the latency for the shared memory is only one cycle.

Codes have been written in C using the version 1.0 of NVIDIA® CUDA™, recently released [7]. The manufacturer provides this version with the CUFFT library, which allows the users to easily run FFT transformations on the graphic platform. CUFFT library offers an API modelled after FFTW [2], for different kinds of transformation and dimensions. We have chosen CUFFT to be used with the purpose of measurement comparisons.

We have executed the forward and inverse FFT measuring the number of GigaFLOPS obtained in these two operations, including the scale factors of inverse FFT. A common metric [2] considers that the number of floating point operations required by a radix-2 FFT is $5Nlog_2(N)$. Thus, if the number of seconds spent by the forward and inverse FFT are t_{FFT} and t_{IFFT}, the number of GigaFLOPS for a N-sample

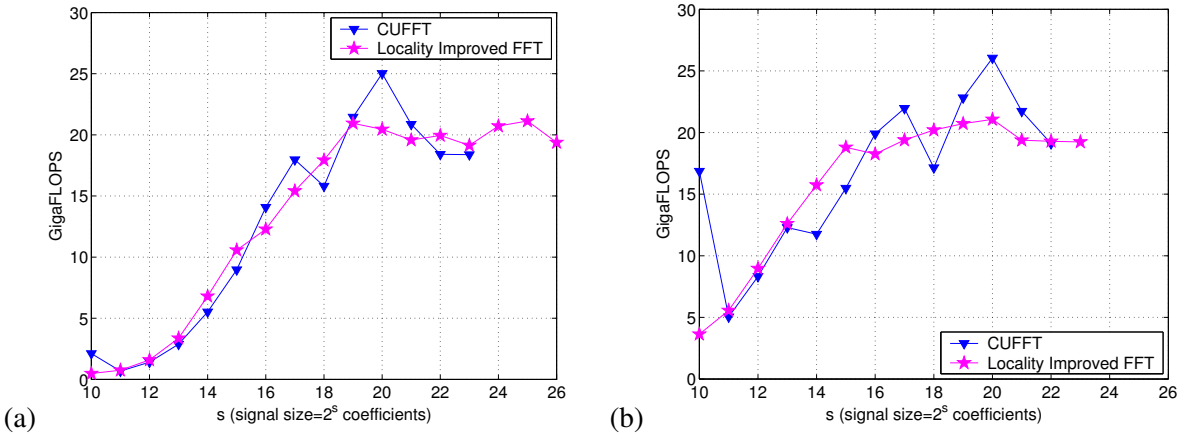

Figure 8: Performance of FFT implementations for 1 single signal (a) and 8 batches of signals (b).

signal will be $GFLOPS = 2\frac{5Nlog_2(N)}{t_{\text{FFT}}+t_{\text{IFFT}}(\text{sec})}10^{-9}$.

Results are shown in Fig. 8 for different signal sizes. Fig. 8(a) corresponds to the execution of one single signal and (b) to the execution of eight batches of signals. The performance of the proposed locality improved FFT implementation is compared to this one of the CUFFT library. Note that the upper limit in the number of coefficients is imposed by the memory size of the device. In fact, for high signal sizes the CUFFT library is unable to perform the transform, because its memory requirements exceed the available global memory.

Some facts can be pointed out. First, the proposed implementation makes a good exploitation of memory locality, allowing a good scalability with the signal size that is comparable to this one of the CUFFT library. In fact, the locality improved implementation is able to outperform CUFFT for some particular signal sizes. Another important feature of the proposed implementation is its ability to manage large size signals. The CUFFT library is limited to 2^{23} coefficients (about 8 million samples), even less when several batches are executed at once. Nevertheless our implementation can manage up to 2^{26} coefficients (about 64 million samples), making a better exploitation of the available device memory.

6 Conclusions

This work discuss the locality features of some implementations of the Fast Fourier Transform on modern graphics processing units. A radix-two decimation-in-time FFT implementation is proposed, that can take advantage of the GPU memory organization. With this purpose, the proposed implementation intends to exploit memory reference locality, making an optimized use of the parallel data cache of the target device. Compared to the FFT library provided by the graphics processor manufacturer, our proposal exhibits a good scalability and it is able to achieve a better performance for certain signal sizes. In addition, it is able to work with signals of larger size than the manufacturer's implementation.

References:

[1] O. Fialka, M. Cadik. FFT and Convolution Performance in Image Filtering on GPU, *Information Visualization*, 2006.

[2] Fastest Fourier Transform in the West (FFTW) Available at: http://www.fftw.org/

[3] N. K. Govindaraju, S. Larsen, J. Gray, D. Manocha. A Memory Model for Scientific Algorithms on Graphics Processors. *Conference on Supercomputing*, Tampa, Florida, USA, 2006.

[4] T. Jansen, B. von Rymon-Lipinski, N. Hanssen, E. Keeve. Fourier volume rendering on the GPU using a split-stream FFT. *Vision, Modeling, and Visualization Workshop*, 2004.

[5] C. Moler. HPC Benchmark. Available at: http://www.hpcchallenge.org/presentations/sc2006/moler-slides.pdf. *Conference on Supercomputing*, Tampa, Florida, USA, 2006.

[6] K. Moreland, E. Angel. The FFT on a GPU. *ACM Conference on Graphics Hardware*, San Diego, California, USA, 2003.

[7] NVIDIA CUDA Homepage. Available at: http://developer.nvidia.com/object/cuda.html

[8] J Spitzer. Implementing a GPU-Efficient FFT. *SIGGRAPH GPGPU Course*, 2003.

[9] T. Sumanaweera and D. Liu. Medical Image Reconstruction with the FFT. *GPU Gems 2*, pp. 765-784, 2005.