Memory Locality Exploitation Strategies for FFT on the CUDA Architecture

Eladio Gutierrez, Sergio Romero, Maria A. Trenas, and Emilio L. Zapata

Department of Computer Architecture
University of Malaga
29071 Malaga, Spain
{eladio,sromero,maria,ezapata}@ac.uma.es

Abstract. Modern graphics processing units (GPU) are becoming more and more suitable for general purpose computing due to its growing computational power. These commodity processors follow, in general, a parallel SIMD execution model whose efficiency is subject to a right exploitation of the explicit memory hierarchy, among other factors. In this paper we analyze the implementation of the Fast Fourier Transform using the programming model of the Compute Unified Device Architecture (CUDA) recently released by NVIDIA for its new graphics platforms. Within this model we propose an FFT implementation that takes into account memory reference locality issues that are crucial in order to achieve a high execution performance. This proposal has been experimentally tested and compared with other well known approaches such as the manufacturer's FFT library.

Key words: Graphics Processing Unit (GPU), Compute Unified Device Architecture (CUDA), Fast Fourier Transform, memory reference locality.

1 Introduction

The Fast Fourier Transform (FFT) nowadays constitutes a keystone 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):

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

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

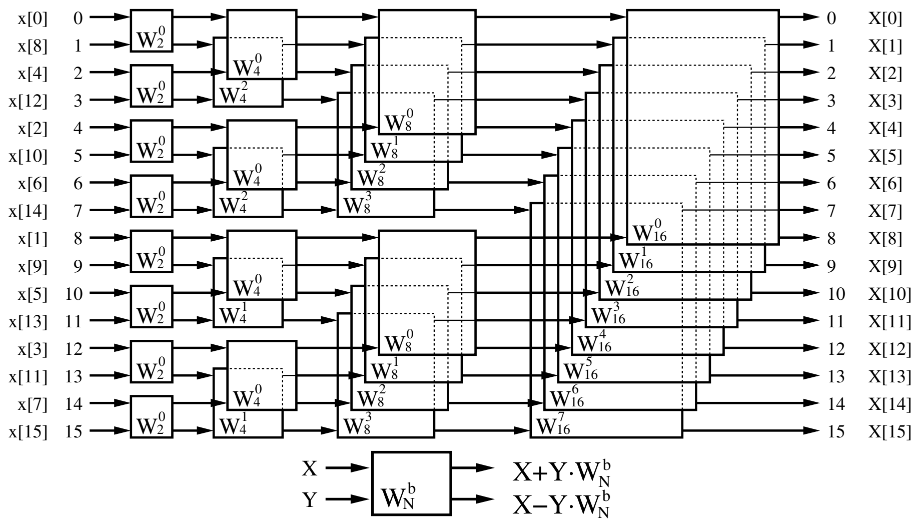

Fig. 1. Radix-2 decimation-in time FFT in terms of butterfly operators.

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. Before the first stage, input coefficients are permuted in bit reversal order 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 $L_{(N,j,i)}(x)$ 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.

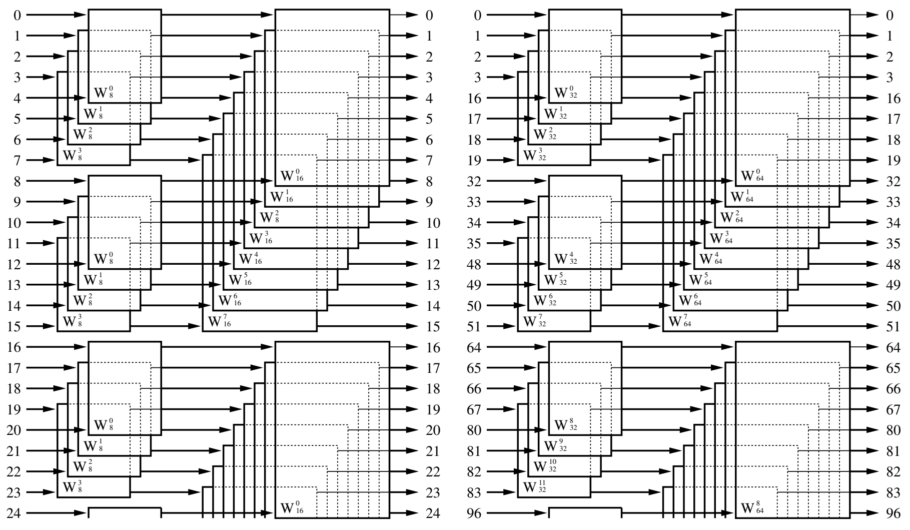

Fig. 2. Computing 3^{rd} and 4^{th} stages of the FFT (a); computing 5^{th} and 6^{th} stages of the FFT using the pattern of 3^{rd} and 4^{th} stages over a properly permuted input (b).

2 CUDA Programming Model

The Compute Unified Device Architecture (CUDA$^{\text{TM}}$) from NVIDIA$^{®}$, 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 SIMD (Single Instruction Multiple Data) multiprocessors each one containing several processing elements (processors), as shown in Fig. 3. Different memory spaces

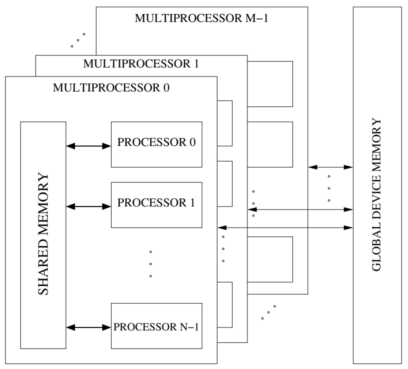

Fig. 3. Organization of processors and memory spaces in CUDA.

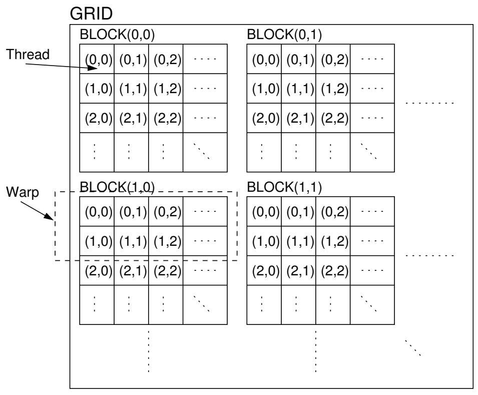

Fig. 4. Thread-based execution model in CUDA.

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, omitted in the figure, for specific purposes: texture and constant memories.

CUDA execution model is based on a hierarchy of abstraction layers: grids, blocks, warps and threads (Fig. 4). The thread is the basic execution unit that is actually mapped onto one processor. A block is a batch of threads cooperating together on one multiprocessor and therefore all threads in a block share the parallel data cache. 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 set of multiprocessors. In turn, a warp is a group of threads executing in an SIMD 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 programming 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. Such kernels are intended to be executed in an SIMD fashion over the processors.

Memory accesses and synchronization scheme are the most important aspects to take into account. Warp addresses issued by SIMD 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. Global synchronization is not provided at the device side, only threads in a block can be waiting one to each other. Thus block synchronization mechanism must be explicitly implemented by the host through consecutive kernel invocations.

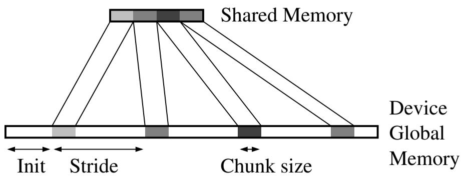

Fig. 5. Data transfer pattern between device and shared memory of `copy_in/copy_out` operations.

3 Implementation Strategies for the FFT

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. 5. It starts from the `ii`-th coefficient and copies `nc` chunks of size `sz` coefficients separated by a stride `st`. Symmetrically, `copy_out(ii,nc,sz,st)` copies coefficients from shared memory back to the device memory. During the SIMD execution of these functions, each thread is in charge of transferring only a pair of coefficients. Observe that threads in a warp must access consecutive memory locations with the purpose of coalescing global memory accesses. Thus, the arguments of such functions describes how coefficients are accessed and hence if this transference fulfills coalescing criteria. In general, the chunk size must be a multiple of the warp size for an optimal transfer. Accesses are serialized when the chunk size is smaller.

The function `fft_level(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 the coefficient vector 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. Executing this function in an SIMD way on a butterfly-per-thread basis, the b-th thread computes the b-th butterfly transformation, so a block performs 2^{r-1} butterflies, one per thread. 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. 6. After a bit reversal permutation, coefficients are transferred to the shared memory, then s invocations of `fft_level` 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. Besides, the original FFT scheme is applied locally (both `fft_level` arguments are equal).

GPU side, one single block

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

Fig. 6. FFT of a signal fitting the shared memory.

The generic case of a signal whose size exceeds the available shared memory of a multiprocessor ($N > 2^r$) is discussed in next subsections. In this case several multiprocessors are involved, meaning that different blocks of threads must collaborate to perform the FFT.

3.1 Straightforward Approach

The first approach to be analyzed is a straightforward solution, but it exploits barely the locality features of the memory access pattern. As threads of different blocks cannot be synchronized within kernel code, required synchronizations 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 `fft_level` 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 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

Host side

```
q=min(r,s);
FirstKernel⟨⟨⟨$2^{s-q}$,$2^{q-1}$⟩⟩⟩(q); --->


if (s>r) {
  q=s-r;
  NextKernel⟨⟨⟨$2^r$,$2^{s-r-1}$⟩⟩⟩(q);--->
}
```

GPU side, block j

```
copy_in(j*$2^q$,1,$2^q$,1);
for (i=1; i$\leq$q; i++)
{
   syncthreads();
   fft_level(i,i);
}
copy_out(j*$2^q$,1,$2^q$,1);

copy_in(j,$2^q$,1,$2^r$);
for (i=1; i$\leq$q; i++)
{
   syncthreads();
   fft_level(i,i+r);
}
copy_out(j,$2^q$,1,$2^r$);
```

FFT mapping

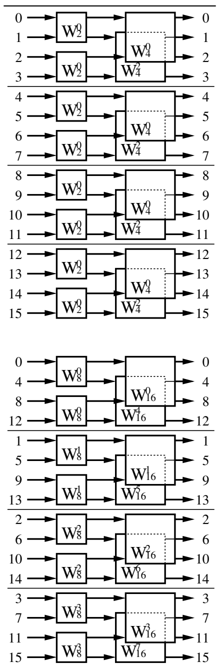

Fig. 7. Straightforward FFT implementation for large signals.

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. For the sake of a simplified discussion, 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, $\lceil \frac{s}{r} \rceil$ kernel invocations will be required. Assigning the 2^r sequences of size 2^{s-r} coefficients with stride 2^r to different blocks (one sequence per block), all the $s-r$ remaining stages can be performed as shown in Fig.7. This pictorial example shows how a 16-samples FFT ($s = 4$) is performed for $r = 2$, that is 4 coefficients per block. Following the notation introduced by CUDA for its extended C language, the numbers enclosed in the triple angle notation (⟨⟨⟨`nB,nTpB`⟩⟩⟩) stand for the total number of blocks and the number of threads per block respectively.

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

```
       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(...);
```

Fig. 8. FFT implementation for large signals using matrix transpositions.

requirement. The approach of Fig. 7 suffers from this lack of coalescing because memory locations accessed by copy operations do not contain chunks of consecutive coefficients. Observe that the third argument (size of chunks) of the copy functions in `NextKernel` invocation is set to one. This way, the first block in the example operates with vector (0, 4, 8, 12).

3.2 Transposition-based FFT

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 [6]. 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. 8. 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.

Broadly, a matrix transposition can be carried out in a block fashion by decomposing it into submatrices of size $m \times n$ fitting the shared memory. Submatrix (i, j) can be copied-in fulfilling the coalescing requirements because the m elements in the same row are consecutive. Once in the shared memory, the submatrix is transposed. Finally, the transposed submatrix is efficiently copied-out in its symmetrical position (j, i) as there are m chunks of n consecutive elements. Source codes for an efficient implementation of the matrix transposition can be found in the manufacturer's website [8].

For signal size larger than $2^r \times 2^r$ this approach uses a higher dimensional matrix representation of the coefficients. In general, for $2^{n \times r}$ coefficients a $n-$dimensional matrix is required. For example, if $2^{2r} < N \leq 2^{3r}$, signal coefficients are arranged in a 3D matrix. In this manner, 1D FFT is needed for each dimension, being necessary to do and undo transpositions not only for the second dimension but also for the third one.

3.3 Locality Improved FFT

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. 9, where the left column corresponds with the host side code for a generic signal size, which has been unrolled to the particular case of two iterations matching a signal up to 2^{2r} samples. Observe that invocations to `NextKernel` are not preceded by any transposition and, what is more important, `copy_in/copy_out` operations meet the 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, in the example of Fig. 9 the host invokes `NextKernel` two times, one half of the higher stages are performed in each invocation. Observe that the third argument (size of chunks) of the copy functions in `NextKernel` invocation is set to 2^{r-q} where q is the number of FFT stages to be computed. Therefore the lower the number of stages the higher the number of kernel invocations and so less reusability of data in the shared memory. Nevertheles if q is less than the warp size the coalescing gets worse.

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 5th and 6th stages

```
     Host side          Host side (unrolled)          GPU side, block j
 ----------------      ------------------      --------------------------------

q=min(r,s);           q=min(r,s);
FirstKernel(q);       FirstKernel(q);   ---> copy_in(j*2^q,1,2^q,1);
                                             for (i=1; i≤q; i++) {
                                                syncthreads();
                                                fft_level(i,i);
p=0;                  p=0;                   }
                      /*1st iteration*/      copy_out(j*2^q,1,2^q,1);
while (q<s){          if (s>r) {
  p=p+q;                p=p+q;
  q=min(r/2,s-p);       q=min(r/2,s-p);
  NextKernel(p,q);      NextKernel(p,q);--->copy_in((j+j/2^q)*2^q,2^q,2^(r-q),2^p);
}                     }                      for (i=1; i≤q; i++) {
                                                syncthreads();
                                                fft_level(r-q+i,p+i);
                                             }
                      /*2nd iteration*/      copy_out((j+j/2^q)*2^q,2^q,2^(r-q),2^p);
                      if (s>r+r/2) {
                        p=p+q;
                        q=s-p-r/2;
                        NextKernel(p,q);--->copy_in((j+j/2^q)*2^q,2^q,2^(r-q),2^p);
                      }                      for (i=1; i≤q; i++) {
                                                syncthreads();
                                                fft_level(r-q+i,p+i);
                                             }
                                             copy_out((j+j/2^q)*2^q,2^q,2^(r-q),2^p);
```

Fig. 9. Improved-locality FFT implementation for large signals.

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`. In order to maximize the coalescing the chunk size should be a multiple of the maximum number of threads per warp. 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. This fact involves that only 5 higher stages can be done in each invocation to `NextKernel`. A lower number of threads per warp allows `NextKernel` to perform more stages, however the degree of fine-grained parallelism will decrease. In Fig. 9 the invocation to `FirstKernel` performs r stages whilst successive invocations to `NextKernel` perform at most $\frac{r}{2}$ stages. Although this technique can double the number of `NextKernel` invocations compared with the straightforward solution, that is, host side synchronizations, the improvement of coalesced global memory accesses is worthwhile because non-coalesced accesses are serialized (up to 32, the number of thread per warp).

4 Experimental 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 [8]. The manufacturer provides a platform–tuned FFT library (CUFFT) which allows the users to easily run FFT transformations on the graphic platform. The CUFFT library offers an API modelled after FFTW [2, 3], for different kinds of transformations and dimensions. We have chosen CUFFT to be used with the purpose of measurement comparisons. Since the manufacturer recommends the transposition strategy for signals exceeding the supported signal size limit, CUFFT for supported signal sizes can be considered an upper limit for the transposition technique.

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 GFLOPS for a N-sample signal will be $GFLOPS = 2\frac{5Nlog_2(N)}{t_{\mathrm{FFT}}+t_{\mathrm{IFFT}}}10^{-9}$. According to the CUFFT/FFTW interface, two dimensionality parameters are taken into consideration: the signal size (N) and the number of signals (b) of the given size to be processed. In literature that is known as a batch of b signals. For example, the transformation of four 1024-sample signals ($N = 1024$) can be permormed by one FFT call using an input vector of 4096 coefficients arranged in a 4-signal batch ($b = 4$). Observe that in this case only 10 FFT levels are carried out. Therefore the measured GFLOPS can be calculated as

$$GFLOPS = 2b\frac{5Nlog_2(N)}{t_{\mathrm{FFT}} + t_{\mathrm{IFFT}}}10^{-9}.$$

Tables 1, 2 and 3 show the experimental results, measured in GFLOPS, by using the previous definition. Table 1 compiles results corresponding to single-signal tranforms in function of the signal size (from 1Ksamples to 64Msamples). The performance of the proposed locality improved FFT implementation is compared with this one of the CUFFT library. A similar comparison is shown

Table 1. Measured GFLOPS for the CUFFT library and the proposed locality improved FFT version (liFFT). N represents the number of coefficients of the transform for a batch of one single signal ($b = 1$). Void entries correspond to unsupported configurations due to memory constraints.

Single signal of N coefficients																	
$\log_2(N)$	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
CUFFT	**2.11**	0.68	1.41	2.88	5.52	8.97	**14.1**	**17.9**	15.8	**21.5**	**25.0**	**20.9**	18.4	18.4	-	-	-
liFFT	0.48	**0.75**	**1.59**	**3.37**	**6.81**	**10.6**	12.3	15.4	**17.9**	20.9	20.5	19.6	**20.0**	**19.1**	**20.7**	**21.1**	**19.4**

Table 2. Measured GFLOPS for the CUFFT library and the proposed locality improved FFT version (liFFT). N represents the number of coefficients of the transform for a batch of 8 signals ($b = 8$). Void entry corresponds to CUFFT unsupported configuration.

Batch of 8 signals, N coefficients per signal														
$\log_2(N)$	10	11	12	13	14	15	16	17	18	19	20	21	22	23
CUFFT	**16.9**	5.0	8.3	12.3	11.8	15.5	**19.9**	**21.9**	17.1	**22.8**	**26.0**	**21.7**	19.1	-
liFFT	3.64	**5.56**	**8.97**	**12.6**	**15.7**	**18.8**	18.3	19.4	**20.2**	20.7	21.1	19.4	**19.3**	**19.2**

in Table 2, where 8-signal batch transforms are considered. Note that for our locality-improved implementation, the upper limit for the total number of coefficients ($b \times N$) is imposed by the size of the device memory, being 2^{26} coefficients.

Results in table 3 show, by means of two series of experiments, the effect of the number of signals in the batch. In both series, the total number of coefficients ($b \times N$) to be processed has been kept constant and equal to 2^{20} and 2^{26} respectively. Observe that, for 2^{26} coefficients, CUFFT library does not support batching with more than 4096 signals although all the coefficients fit in the global memory.

The proposed implementation makes a good exploitation of memory locality, allowing a good scalability with the signal size. In fact, although neither of the two methods exhibits a clear advantage, for several interesting situations the locality improved implementation is able to provide better results than CUFFT. Examples of such situations are when there is a high number of signals in a batch and for very large signal size.

This ability to manage large size signal constitutes an important feature of the proposed implementation. The CUFFT library is unable to perform the transform beyond 8 million elements (2^{23}) [8] whereas our implementation can manage up to 2^{26} coefficients (about 64 million samples), making a better exploitation of the available device memory.

5 Related Work

The FFT represents a computationally intensive floating-point 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

Table 3. Measured GFLOPS for the CUFFT library and the proposed locality improved FFT version (liFFT). N represents the number of coefficients of each signal in the batch. The total number of coefficients to be processed is fixed to 2^{20} coefficients (upper table) and 2^{26} coefficients (lower table). Void entries correspond to unsupported memory configurations due to the large number of signals in the batch.

2^{20} coefficients, batch of $2^{20}/N$ signals											
$\log_2(N)$	10	11	12	13	14	15	16	17	18	19	20
CUFFT	**40.7**	**22.9**	**22.3**	**22.1**	13.7	16.8	**20.6**	**22.0**	16.8	**22.2**	**25.0**
liFFT	18.5	17.4	18.1	18.7	**19.3**	**19.9**	18.9	19.4	**19.8**	20.5	20.4

2^{26} coefficients, batch of $2^{26}/N$ signals											
$\log_2(N)$	10	11	12	13	14	15	16	17	18	19	20
CUFFT	-	-	-	-	14.1	17.3	**21.3**	**22.8**	17.2	**23.1**	**26.2**
liFFT	**19.0**	**18.1**	**18.8**	**19.4**	**20.1**	**20.6**	19.7	20.1	**20.6**	20.9	21.2

the last years focused on porting FFT algorithms to graphics processing units. In [9] very basic ideas of how to implement the FFT algorithm are collected. In [7], implementations of the FFT in the context of image processing applications are presented using GPU shader programming. Also in other specific contexts FFT has been developed on graphics hardware, like [10, 1, 5]. A discussion about the FFT implementation, together with other algorithms, is found in [4]. This last work tries to exploit the GPU memory hierarchy in order to improve the performance of the implementations but using programming models prior to CUDA.

6 Conclusions

Locality features of some implementations of the Fast Fourier Transform using the NVIDIA CUDA programming model are discussed in this work. 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. Moreover, it is able to work with signals of larger size than the manufacturer's implementation.

References

1. Fialka, O., Cadik, M.: 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. Frigo, M., Johnson, S.G.: The Design and Implementation of FFTW3. Proceedings of the IEEE 93, 216–231 (2005)
4. Govindaraju, N.K., Larsen, S., Gray, J., Manocha, D.: A Memory Model for Scientific Algorithms on Graphics Processors. Conference on Supercomputing (2006)
5. Jansen, T., von Rymon-Lipinski, B., Hanssen, N., Keeve, E.: Fourier volume rendering on the GPU using a split-stream FFT. Vision, Modeling, and Visualization Workshop (2004)
6. Moler, C.: HPC Benchmark. Available at: `http://www.hpcchallenge.org/presentations/sc2006/moler-slides.pdf.` Conference on Supercomputing (2006)
7. Moreland, K., Angel., E.: The FFT on a GPU. ACM Conference on Graphics Hardware (2003)
8. NVIDIA CUDA Homepage. Available at: `http://developer.nvidia.com/object/cuda.html`
9. Spitzer, J.: Implementing a GPU-Efficient FFT. SIGGRAPH GPGPU Course (2003)
10. Sumanaweera, T., Liu, D.: Medical Image Reconstruction with the FFT. GPU Gems 2, 765–784 (2005)