FFTs of Arbitrary Dimensions on GPUs

Xiaobai Sun and Nikos Pitsianis
Department of Computer Science, Duke University, Durham, NC 27708

Introduction

We present the fast Fourier transform (FFT), of arbitrary dimensions, on the graphics processing unit (GPU). The FFT on GPUs exploits the architecture in its image processing capability, as well as its particular graphics/image rendering capacity. It also couples the processing and rendering furthermore. We view the GPU as a special architecture that supports fine-granularity, two-dimensional (2D) memory accesses at the level of application programming interface (API). The unique architectural features are utilized by mathematical and algorithmic means richly associated with the FFT, which has an important role in signal and image processing and in scientific computing in general.

At the kernel of the FFT on GPUs, i.e., at the level innermost to the the native architecture, are the primitive array operations for the 2D FFT, instead of the 1D FFT. Basically, the 2D array operations have natural mappings to the architecture by their joint potential in performance. A lower or higher dimensional FFT is described in terms of the kernel operations, in order to exploit the architecture at the application programming level. This algorithmic abstraction of the operation primitives and their compositions enables, especially, the 2D twiddle scaling, which uses less memory space, and the 2D bit-reversal permutation, which manifests the unique GPU feature in memory access. The 2D FFT on GPUs is detailed in [3], where mixed-radix factorizations are also used to further utilize the memory resource. In this paper we turn the focus onto FFTs of other dimensions on GPUs. We describe the FFT reformulation and data mappings. We provide experimental results to demonstrate that the 2D FFT performance is conveyed to the other FFTs as well.

The One-Dimensional FFT

Consider the 1D FFT, $y = F_m\, x$, where F_m is the $m\times m$ discrete Fourier transform (DFT). Assume that m is a composite number, $m = p * q$, for some non-trivial natural numbers p and q. Otherwise, augment m to the least composite number above. Let X_{pq} be the $p \times q$ array created by folding the input vector x row-wise. The k-th row of X_{pq} is the k-th segment of consecutive q elements in x, $k = 1, \cdots, p$. Let Y_{pq} be the $p \times q$ array from which the transformed (output) vector y is read out column-wise. Then, we have the following mathematical reformulation of the 1D FFT,

$$Y_{pq} = [\,(F_p X_{pq}) \odot W_{pq}\,]\, F_q$$

where $\odot$ denotes the Hadamard (or element-wise) product, W_{pq} is the $p\times q$ scaling array formed by folding column-wise the twiddle vector between the factor associated with F_p and that with F_q. When the elements of the scaling array W_{pq} are all set to 1, Y is the 2D FFT of X. In other words, a simple flag can be set to invoke or skip the twiddle factor for the 1D FFT or the 2D FFT, respectively.

There are additional advantages in the reformulated FFT. The twiddle array W_{pq} is a submatrix of F_m itself. Therefore, there are many ways to represent the twiddle array compressively in space and expand them efficiently during the computation. Furthermore, a row-wise permutation of $(F_p X_{pq}) \odot W_{pq}$ can be applied to $F_p X_{pw}$ and W_{pq} separately. This means that a row-wise permutation, such as the bit-reversal permutation, can be pre-extracted from F_p and W_{pq}. Consider the following arrangement,

$$Y_{pq} = P_r[\,(P_r^{\mathrm{T}} F_p X_{pq}) \odot (P_r^{\mathrm{T}} W_{pq})\,]\,(F_q P_c^{\mathrm{T}}) P_c,$$

where the twiddle array is permuted in rows, the outer permutations P_r and P_c may be the bit reversals associated with p and q, respectively. On GPUs, they can be done simultaneously and efficiently, see [3] for detail.

Multi-dimensional FFTs

Consider the 3D FFT first. There are more varieties in reformulating a 3D FFT in terms of 2D array operations. For example, we may reformulate the $\ell\times m\times n$ FFT in terms of $(\ell m)\times n$ array operations. The input and output data cubes $X_{\ell,m,n}$ and $Y_{\ell,m,n}$ are mapped

to the 2D arrays $X_{(\ell m),n}$ and $Y_{(\ell m),n}$, respectively, on the GPU. Specifically, the k-th cross section of the input cube along the last dimension, becomes the k-th column of $X_{(\ell m),n}$. The output cube is mapped in the same way. It is then computed by the following reformulation

$$Y_{(\ell m),n} = \tilde{F}_{\ell m} X_{(\ell m),n} F_n,$$

where $\tilde{F}_{\ell m} = F_m \otimes F_\ell$ is the DFT matrix $F_{\ell m}$,

$$F_{\ell m} = (F_m \otimes I_\ell)\, D_w\, (I_m \otimes F_\ell)\, P_{\ell,m},$$

with the twiddle scaling factor D_w and the stride-m permutation $P_{\ell,m}$ replaced by the identity matrix $I_{\ell m}$. Similar to the reformulation of the 1D FFT, the setting of the twiddle scalars distinguishes the 3D FFT from the 2D FFT and associates them at the same time in the framework of 2D array operations.

The scheme of cross-dimension aggregation alone may result in 2D data arrays that are not well balanced dimensionwise. For instance, a 192×6144 data array may not be accommodated by the available buffer space as the reshaped 1152×1024 array may. The complementary scheme is dimensional splitting as used for the 1D FFT in the previous section. For instance, when the input and output data cubes are well balanced dimensionwise, we may factor the middle dimension m by m_1 and m_2, i.e., $m = m_1 m_2$, and then aggregate (ℓ, m_1, m_2, n) into $(\ell m_1, m_2 n)$.

The general scheme may be described as dimensionwise splitting, $\ell = \ell_1 \ell_2$, $m = m_1 m_2$, and $n = n_1 n_2$, followed by cross-dimension aggregation, $(\ell_1 m_1 n_1, \ell_2 m_2 \ell_2)$. The general specification of the data mapping is as follows. Any row or column in the 2D array, $X_{\ell_1 m_1 n_1, \ell_2 m_2 n_2}$, corresponds to a subcube of the data cube $X_{\ell,m,n}$. We note also that stride permutations can be incorporated into the scheme so that a subarray corresponds to the data cube at a different granularity level.

The same scheme of splitting, aggregation and permutations can be straightforwardly extended to any other multi-dimensional FFT.

Experiments

The FFT on GPUs is implemented at the application programming level using the language `Cg` and the graphics library `OpenGL`. In Table 1 we show the execution time in milliseconds of three FFTs on square data arrays. The 3D FFT arises from data analysis in a particular hyper-spectral imaging application. The performance of the 2D FFT is determined by and remains the same as the primitive array operations on GPUs. And it is transported to the other FFTs as well.

Table 1: Execution time in milliseconds for three FFTs on two GPUs, in single-precision floating point arithmetic

GPUs / FFTs	nVIDIA 7900GT	ATI X1900XTX
$1 \times 262,144$	8	8
1024×1024	35	36
$512 \times 512 \times 16$	153	158

Finally, we add a few comments. Since its first report in [1], the FFT is implemented and optimized in performance almost on every computer architecture, see [2, 4] for instance. There exist other 1D or 2D FFT implementations on GPUs [5]. Here, we use the GPU as a prototype for architectures supporting 2D memory access at fine granularity. We consider the FFT on GPUs from both the application user's viewpoint and the developer's viewpoint. We do not fix the number of dimensions to 1 or 2, nor limit each dimensional size to a power of 2. The code is developed quickly at the API level, easily readable, and portable to any GPU that supports `Cg` and `OpenGL`. Moreover, we consider it very important to specify clearly and systematically the input and output data mappings.

References

[1] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, April 1965.

[2] M. Püschel *et. al.* SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232–275, 2005.

[3] T. Fridrich, N. Pitsianis, and X. Sun. Mixed-Radix 2D FFT on GPUs. Technical Report CS-2007-02, Duke University, 2007.

[4] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, February 2005.

[5] Kenneth Moreland and Edward Angel. The FFT on a GPU. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 112–119, 2003.