Li Y, Zhang YQ, Liu YQ *et al.* MPFFT: An auto-tuning FFT library for OpenCL GPUs. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 28(1): 90–105 Jan. 2013. DOI 10.1007/s11390-013-1314-8

MPFFT: An Auto-Tuning FFT Library for OpenCL GPUs

Yan Li[1,2] (李　焱), *Student Member, CCF, ACM*, Yun-Quan Zhang[1,*] (张云泉), *Member, CCF, ACM, IEEE*
Yi-Qun Liu[1,2] (刘益群), *Student Member, CCF, ACM*, Guo-Ping Long[1] (龙国平), and Hai-Peng Jia[3] (贾海鹏)

[1] *Institute of Software, Chinese Academy of Sciences, Beijing 100190, China*

[2] *Graduate University of Chinese Academy of Sciences, Beijing 100049, China*

[3] *School of Information Science and Engineering, Ocean University of China, Qingdao 266000, China*

E-mail: liyan08@iscas.ac.cn; zyq@mail.rdcps.ac.cn; {liuyiqun.jlu, longguoping, jiahaipeng95}@gmail.com

Received November 18, 2011; revised September 25, 2012.

Abstract Fourier methods have revolutionized many fields of science and engineering, such as astronomy, medical imaging, seismology and spectroscopy, and the fast Fourier transform (FFT) is a computationally efficient method of generating a Fourier transform. The emerging class of high performance computing architectures, such as GPU, seeks to achieve much higher performance and efficiency by exposing a hierarchy of distinct memories to software. However, the complexity of GPU programming poses a significant challenge to developers. In this paper, we propose an automatic performance tuning framework for FFT on various OpenCL GPUs, and implement a high performance library named MPFFT based on this framework. For power-of-two length FFTs, our library substantially outperforms the *clAmdFft* library on AMD GPUs and achieves comparable performance as the CUFFT library on NVIDIA GPUs. Furthermore, our library also supports non-power-of-two size. For 3D non-power-of-two FFTs, our library delivers 1.5x to 28x faster than FFTW with 4 threads and 20.01x average speedup over CUFFT 4.0 on Tesla C2050.

Keywords fast Fourier transform, GPU, OpenCL, auto-tuning

1 Introduction

The fast Fourier transform (FFT) is one of the most critical computational kernels which has broad applicability across a wide range of disciplines in audio signal processing, image processing, spectral methods for solving partial differential equation (PDE) and so on. Many algorithms have been proposed for solving FFT efficiently since 1965[1]. However, the FFT is only a good starting point if an efficient implementation exists for the architecture at hand, and optimizing memory accesses and scheduling computational operations for the FFT on modern platforms is a serious challenge.

The gap between processor performance and memory latency has increasingly widened over the last decade. The resulting increased complexity of memory systems to ameliorate this gap has made it increasingly harder for compilers to optimize arbitrary code within an acceptable amount of time. Although GPUs are promising platforms for general-purpose high-performance computing, the state-of-the-art numerical libraries suffer tremendously from the new memory design and organization. We have to explicitly orchestrate efficient data transfers among massive threads from memory hierarchy to processing elements. Managing data locality in GPUs requires trade-offs in performance, code complexity and optimization effort. Accordingly, its programming complexity poses a significant challenge to programmers.

Due to the fast product cycles in hardware development and the complexity of today's execution environments, evolution of hardware technology is not accompanied with innovative software technology that makes the computational capability of the hardware unavailable to scientists and engineers. It gets more and more complicated to design algorithms that are able to utilize modern computer systems to a satisfactory degree. Algorithms which were optimized for a specific architecture several years ago, fail to perform well on current and emerging architectures. Software automatic tuning is considered to be the most promising paradigm that will meet such a demand of software technology.

Regular Paper

This work is supported in partial by the National Natural Science Foundation of China under Grant Nos. 61133005, 61272136, 61100073, 61100066, the National High Technology Research and Development 863 Program of China under Grant Nos. 2012AA010902, 2012AA010903, and the Chinese Academy of Sciences Special Grant for Postgraduate Research, Innovation and Practice.

*Corresponding Author

There has been significant progress over the last decades on FFT in the development of auto-tuning technique and many FFT libraries have been built on CPUs and GPUs. On the GPU side, high performance vendor FFT libraries are provided, such as CUFFT① on NVIDIA GPUs, *clAmdFft*② on AMD GPUs, and several other research FFT libraries[2-5]. Although the researches they did demonstrated significant performance improvement against prior state-of-the-art FFT algorithms on GPUs, they just took the NVIDIA GPU into account and ignored the AMD GPU. Hence all the developed libraries are not cross-platform adaptive. Furthermore, there has been a substantial body of work on FFT, but most of it is not extended to non-power-of-two size.

The major contributions of our work can be summarized as follows. First, an auto-tuning framework to optimize 3D FFT algorithms on multiple platforms with OpenCL is proposed. To the best of our knowledge, it is the first auto-tuning framework for FFT supporting AMD GPUs. Second, a high performance library named MPFFT based on the framework is implemented and analyzed. Last but not least, the differences of features of architectures between AMD and NVIDIA GPUs that affect program performance are enumerated. Our analysis and structured memory optimization techniques are based on the Kronecker product that captures the memory access pattern and other performance effects of major GPU microarchitecture features in our library. Furthermore, it can also account for the differences in the underlying hardware and programming language constructs of these two platforms.

The remainder of the paper is organized as follows. Section 2 presents the background and motivation. Section 3 elaborates the proposed auto-tuning framework on GPUs. Section 4 illustrates the performance results and analysis of our library in detail. Section 5 discusses the related work. Section 6 provides conclusions and future implications for this work.

2 Background and Motivation

In this section, we provide a brief overview of the GPU architecture and OpenCL programming model. Our auto-tuning framework is constructed based on this model and well-suited to AMD and NVIDIA GPUs. Furthermore, FFT algorithms with their representations we used as well as the motivation of our work are presented.

2.1 OpenCL Programming Model

OpenCL (Open Computing Language)[6-7] is an open royalty-free standard for general purpose heterogeneous parallel programming across CPUs, GPUs and other processors, giving software developers portable and efficient access to the power of these processing platforms. OpenCL consists of a programming language used to write computational kernels that execute on compute devices, and runtime API functions used to coordinate executions of the kernels, configure and manage OpenCL objects. Vendor-specific implementations of OpenCL are provided by almost all major CPU and GPU vendors, in particular NVIDIA, AMD, Intel and IBM. These implementations allow for using OpenCL with a limited set of devices types.

Fig.1(a) shows an overview of the GPU architecture and thread execution model in OpenCL. OpenCL provides a framework for coordinating parallel computation across heterogeneous processors, and a cross-platform programming language with a well specified computation environment. It is similar in style to the single program multiple data (SPMD) parallel programming model. The OpenCL platform consists of a host processor connected to multiple compute devices, and the devices are most easily defined as a collection of compute units (CUs), each containing multiple processing elements (PEs) of which the functionality equals to streaming processor core (cuda core in Fermi architecture) in NVIDIA GPU or stream core in AMD GPU. The PE executes the computation commands submitted from an OpenCL application. All PEs within a CU execute a single stream of instructions as single instruction multiple data (SIMD) or as SPMD.

An OpenCL application consists of a host program which executes on the host processor and kernels that are functions for being accelerated on compute device using OpenCL API. The host program provides command-queue for performing computation in-order or out-of-order on the PEs, and also defines a multi-dimensional abstract index space. Each point within its index space is associated with an execution instance of the kernel, which is defined as work-item. Work-items are further grouped as work-groups. Work-items in a work-group execute concurrently on the PEs of a single CU, and all work-items in a work-group can cooperate, whereas work-items from different work-groups cannot. Furthermore, they are grouped into multiple warps, which are the granular multi-threading scheduling units. A warp is conceptually equivalent to a wavefront in AMD GPUs. If work-items within a wavefront/warp diverge, such as branching, all execution paths are executed serially. This phenomenon is called thread divergence[8] which would degrade the performance greatly.

①http://developer.nvidia.com/cuda/cufft, Sept. 2012.

②http://developer.amd.com/libraries/appmathlibs/pages/default.aspx, Sept. 2012.

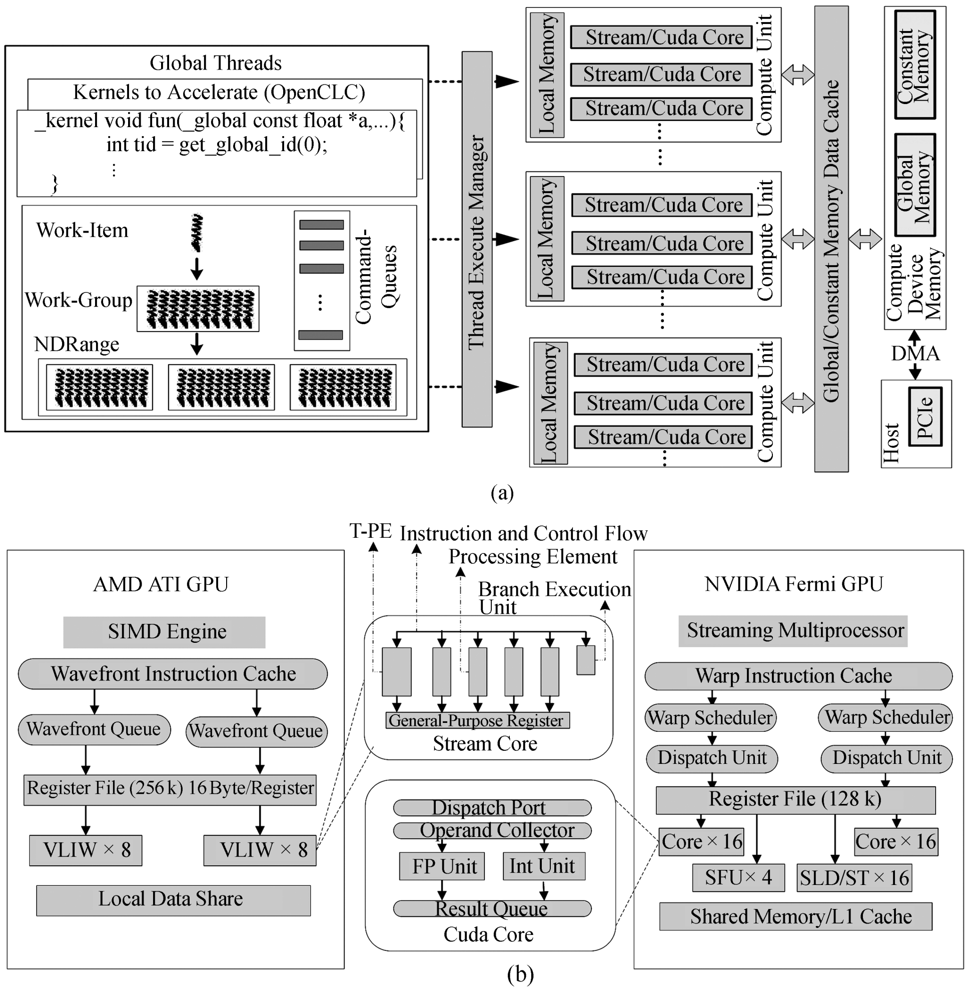


Fig.1. Overview of the OpenCL framework and GPU architectures. (a) OpenCL programming model. (b) Differences of architecture between AMD and NVIDIA GPU.

Different architectures among the underlying platforms pose various challenges in memory optimization and parallelism management, resulting in different performances. The architectures of the NVIDIA and AMD GPU are shown in Fig.1(b). The NVIDIA GPU employs a scalar architecture in the sense that computation and memory operations are not necessarily executed in a vector fashion. The thread execution model is called single instruction multiple threads (SIMT). Nonetheless, due to its simplicity, it is easier for application and game developers to program. Furthermore, global memory coalescing and bank conflict avoidance in the local memory access are the key optimization to achieve high memory bandwidth and high performance. The AMD GPU architecture is a bit different. In each block of six stream processing units, four are identical, the 5th carries different FP/INT arithmetic functions, and the 6th keeps things in check. Essentially, each block of five stream processors (ignoring the special unit) is comparable to one NVIDIA stream processor. Accordingly, the major difference is probably that AMD GPU is vector-based, whereas NVIDIA GPU is scalar-based. Because of AMD GPU's architecture, developers have a tougher time programming to take full advantage of every stream processor on board. Furthermore, the benefits of vectorization far overweigh the penalties of local memory bank conflicts in AMD GPUs, whereas the benefits from vectorization are limited in NVIDIA GPUs[9].

2.2 FFT Algorithms

In fact, the FFT algorithm is reported to be one of the top ten algorithms in the 20th century. A considerable research effort has been devoted to optimization of FFT codes over the past four decades. The algorithm presented by Cooley and Tukey[10] reduces the algorithm complexity of computing the naïve DFT (discrete

Fourier transform) from $O(n^2)$ to $O(n \log n)$, which is viewed as a turning point for applications of the Fourier transform.

The DFT of a sequence $x = x_0, \ldots, x_{n-1}$ is defined in summation form as follows:

$$y_j = \boldsymbol{DFT}_N x = \sum_{k=0}^{n-1} \omega_N^{jk} x_k, \tag{1}$$

where $k \in [0, n-1]$ and $\omega_N = e^{\frac{-2i\pi}{n}}$.

DFT can be represented in many different forms. The butterfly is extracted from a signal flow graph implementing an FFT algorithm for simplicity. In this paper, we adopt Kronecker product to design and implement FFT algorithms. The property of the formalism facilitates verification of the correctness in the code generation. Furthermore, it assists us with implementing the GPU kernel and optimize its performance.

If $\boldsymbol{A}$ is an $m \times n$ matrix and $\boldsymbol{B}$ is a $p \times q$ matrix, then the Kronecker product $\boldsymbol{A}$ by $\boldsymbol{B}$ is an $mp \times nq$ matrix denoted by $\boldsymbol{A} \otimes \boldsymbol{B}$ and defined by

$$\boldsymbol{A} \otimes \boldsymbol{B} = [a_{ij}\boldsymbol{B}]_{i,j} \tag{2}$$

with $\boldsymbol{A} = [a_{ij}]_{i,j}$.

The direct sum of $\boldsymbol{A}$ and $\boldsymbol{B}$ is an $(m+q) \times (n+s)$ matrix denoted by $\boldsymbol{A} \oplus \boldsymbol{B}$ and defined by

$$\boldsymbol{A} \oplus \boldsymbol{B} = \begin{pmatrix} A & \mathbf{0} \\ \mathbf{0} & B \end{pmatrix}, \tag{3}$$

where the $\mathbf{0}$'s denote blocks of zeros with appropriate size.

The mathematical identities for FFT algorithms can be obtained by referring to [11-13]. Historically, FFT algorithms were obtained by applying breakdown rules recursively. The rule we use is expressed in (4), and it is applied to exhibit the parallel Kronecker product structure which leads to the so-called 6-step or parallel FFT algorithm. The more detailed description of FFT algorithms and their variants can be found in [1, 11, 14], and the notation we use here mostly coincides with the notation in [11-12].

$$\begin{aligned}\boldsymbol{DFT}_{n_2 n_1} = &\boldsymbol{L}_{n_2}^{n_2 n_1}(\boldsymbol{I}_{n_1} \otimes \boldsymbol{DFT}_{n_2})\boldsymbol{L}_{n_1}^{n_2 n_1} \cdot \\ &\boldsymbol{T}_{n_1}^{n_2 n_1}(\boldsymbol{I}_{n_2} \otimes \boldsymbol{DFT}_{n_1})\boldsymbol{L}_{n_2}^{n_2 n_1}.\end{aligned} \tag{4}$$

The twiddle factor matrix, denoted by $\boldsymbol{T}_{n_1}^{n_1 n_2}$, is the diagonal matrix (abbreviated as $\boldsymbol{diag}$):

$$\begin{aligned}\boldsymbol{T}_{n_1}^{n_1 n_2} &= \bigoplus_{i=0}^{n_2-1} \bigoplus_{j=0}^{n_1-1} \omega_{n_1 n_2}^{ij} \\ &= \bigoplus_{j=0}^{n_2-1} \boldsymbol{diag}(1, \omega_{n_1 n_2}^{j}, \ldots, \omega_{n_1 n_2}^{j(n_1-1)})\end{aligned} \tag{5}$$

and moreover, the notation $\boldsymbol{L}_n^{mn}$ denotes the stride permutation which indicates that a vector of size mn is reordered by loading into n segments at stride n.

2.3 Motivation

The programming models and interfaces in traditional graphics processors were highly specialized and limit the ability of developers to map general-purpose applications to these platforms. With the advent of CUDA[15] and OpenCL, researchers now have the programming and architectural features to quickly port programs to a platform with a massively parallel, GPU-based coprocessor[16-17]. However, the amount of effort required to maximize the performance of applications on GPU architectures is relatively large. Due to resource restrictions and the threading model of the GPU, the optimization space can also be discontinuous[18]. Chasing performance gains through manual tuning becomes a complex and time-consuming process that is neither scalable nor portable. Automatic performance tuning or auto-tuning is a promising and viable approach to address the limitations of manual tuning. In recent years, it has emerged as an effective approach to tune scientific kernels for both multi-core processors and GPUs. It can handle the increasing complexity in GPU computation and memory subsystems effectively. Recent research has demonstrated that GPUs can significantly accelerate the performance of FFTs as compared to CPUs. Nukada and Matsuoka[3] used auto-tuning for optimizing 3D FFT performance on GPUs, but their approach has limited search space and is unlikely to be efficient for arbitrary FFT configurations. Yuri Dotsenko *et al.*[4] presented a complete FFT framework to optimize an FFT library for arbitrary dimensions and sizes.

However, the methodology of auto-tuning techniques in MPFFT is very different from all the above-mentioned work. We employ two-stage adaptation methodology in different levels, namely installation time and runtime. At installation time, there is a code generator that could automatically generate FFT code for arbitrary size called by GPU kernel. The code generator also could generate high optimized code for GPU kernel according the auto-tuning techniques at runtime. Hence MPFFT is more adaptive to various styles of architecture. The framework of MPFFT is fully discussed in the following sections. Furthermore, both of the libraries they developed are not cross-platform adaptive. The intent of our work is to model the GPU organization and features for constructing an auto-tuning performance model of FFT on the GPU architecture. Our model is well-suited both on AMD and NVIDIA GPUs.

2.4 Mapping FFTs to GPUs

Due to the various features of different architectures, many variants of FFT algorithms and their implementation are required. The explicit digit reversal permutations required by the Cooley-Tukey algorithm are avoided in the Stockham auto-sort algorithm (shown in (6)) by changing the permutations across stages[4,11,19]. The transformations of 4-D arrays which illustrate the Stockham algorithm are shown in Fig.2. The row-column algorithm is yielded by applying the definition of 2D DFT and properties of Kronecker product ((7)). Higher-dimensional DFT algorithms are derived similarly.

$$\begin{aligned}\boldsymbol{DFT}_N =&\prod_{i=1}^{n}(\boldsymbol{DFT}_{n_i}\otimes \boldsymbol{I}_{N/n_i})(\boldsymbol{T}_{n_i}^{m_in_i}\otimes \boldsymbol{I}_{l_i})\cdot\\ &(\boldsymbol{L}_{n_i}^{m_in_i}\otimes \boldsymbol{I}_{l_i}),\\ &\quad l_1=1, l_{i+1}=n_il_i, m_i=\frac{N}{l_{i+1}},\end{aligned}\tag{6}$$

$$\begin{aligned}\boldsymbol{DFT}_{m\times n}=&(\boldsymbol{I}_m\otimes \boldsymbol{DFT}_n)(\boldsymbol{DFT}_m\otimes \boldsymbol{I}_n)\\ =&(\boldsymbol{I}_m\otimes \boldsymbol{DFT}_n)\boldsymbol{L}_m^{mn}(\boldsymbol{I}_n\otimes \boldsymbol{DFT}_m)\boldsymbol{L}_n^{mn}.\end{aligned}\tag{7}$$

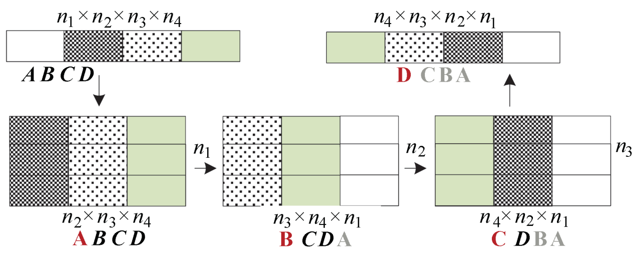

$n_1\times n_2\times n_3\times n_4$
A B C D
$n_4\times n_3\times n_2\times n_1$
D C B A
n_1
n_2
n_3
$n_2\times n_3\times n_4$
A B C D
$n_3\times n_4\times n_1$
B C D A
$n_4\times n_2\times n_1$
C D B A

Fig.2. Stockham FFT algorithm using 4-D partitioning (A B C D). The partition being transformed in each stage is shown in red. Untransformed partitions are shown in italicized bold and transformed partitions in gray.

For a given size and certain dimensional FFT, we first partition it into multiple dimensions without violating on-chip resource usage, i.e., the amount of local memory and the number of registers used by a kernel along with other hardware limits. To begin with, the threads load data from global memory to registers for computing FFT along one dimension, then multiply with twiddle factors and shuffle data for the subsequent transforms along other dimensions through local memory. Finally, the results are written back to global memory.

Fig.3 illustrates how to compute large-size FFT on the GPU. The factorization and computation for local memory is also similar to it. Assume that the input size is 16 M, the library factorizes it according to the value $\#T$ because the size is larger than $\#L$. We use term $\#T$ to indicate the threshold for multi-dimensional decomposition in the global memory while term $\#L$ stands for the maximum size supported in the local memory. If $\#T$ equals 512, then $16\,\text{M} = 512\times 512\times 64$. Hence the library needs to factorize 512 and 64 along each dimension. Sequently, the code generator module generates multiple GPU code versions based on these various factorizations. Then the library employs the search algorithm to prune the search space. Furthermore, if the called codelet size is too small, each thread executes several batched FFTs to improve the occupancy of resources on the CU.

3 Auto-Tuning Framework

GPUs have recently become popular as general computing devices due to their relatively low costs, massively parallel architectures, and improving accessibility provided by programming environments such as the CUDA and OpenCL framework. There are two levels of parallelism to be extracted to exploit parallelism at the thread block level and the thread level for GPUs. Furthermore, GPUs have a deep memory hierarchy needed to be orchestrated explicitly. Effectively exploiting both GPU computational resources and memory bandwidth is critical to achieve peak per-node performance. Hence we employ a two-stage adaptation methodology to map FFT algorithms to the GPU architecture and memory hierarchy, as shown in Fig.4. At installation time, the codelets library consists of many small size DFTs that

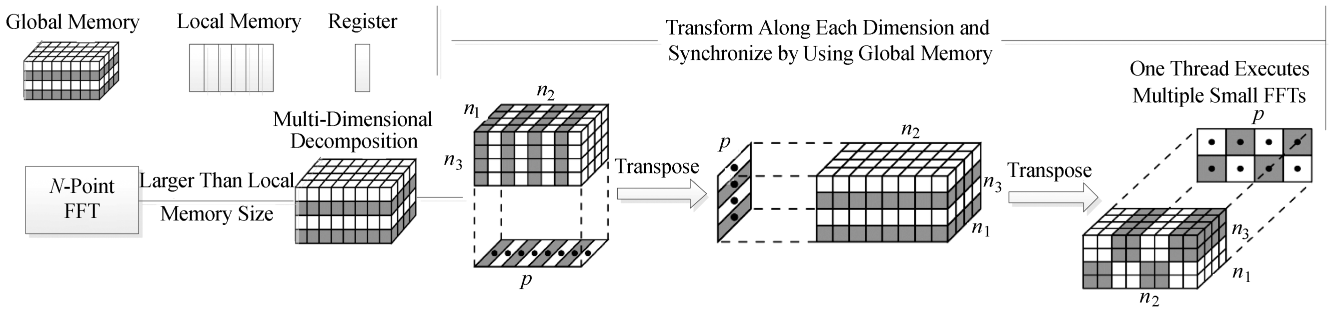

Global Memory
Local Memory
Register
Transform Along Each Dimension and Synchronize by Using Global Memory
One Thread Executes Multiple Small FFTs
Multi-Dimensional Decomposition
N-Point FFT
Larger Than Local Memory Size
Transpose
Transpose
n_1
n_2
n_3
p

Fig.3. Overview of the workflow for computing large-size FFT.

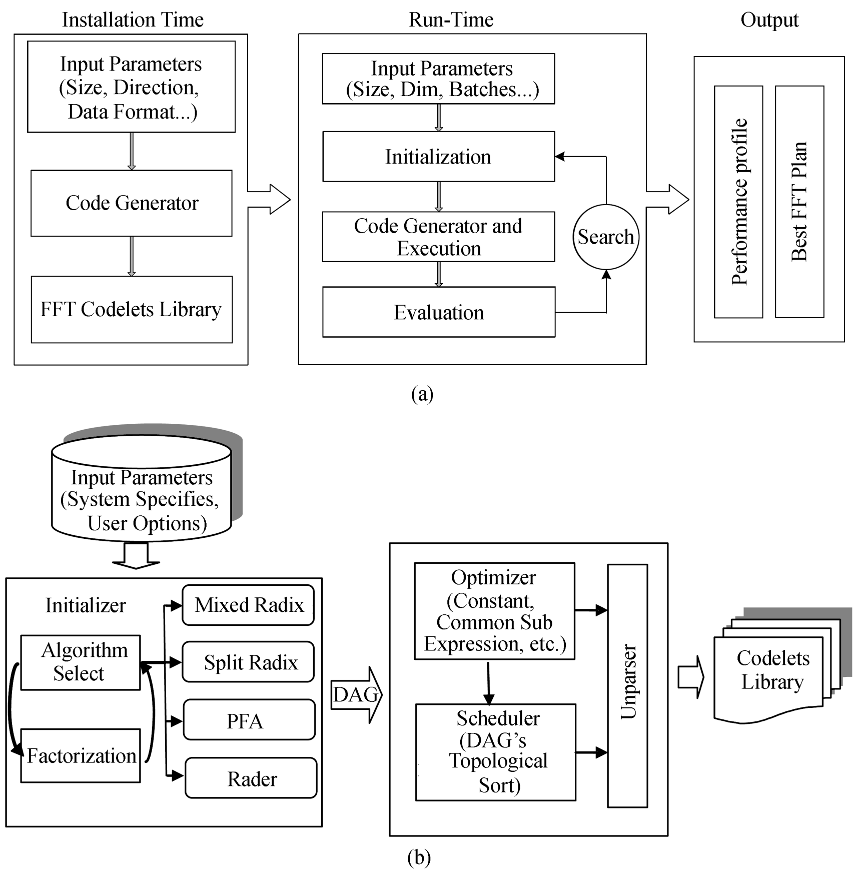

Installation Time
Input Parameters (Size, Direction, Data Format...)
Code Generator
FFT Codelets Library
Run-Time
Input Parameters (Size, Dim, Batches...)
Initialization
Code Generator and Execution
Evaluation
Search
Output
Performance profile
Best FFT Plan
(a)
Input Parameters (System Specifies, User Options)
Initializer
Algorithm Select
Factorization
Mixed Radix
Split Radix
PFA
Rader
DAG
Optimizer (Constant, Common Sub Expression, etc.)
Scheduler (DAG's Topological Sort)
Unparser
Codelets Library
(b)

Fig.4. Overview of the adaptive optimization framework for our FFT algorithms. (a) Framework of our MPFFT framework. (b) Block diagram of code generator module.

are generated by a lightweight code generator module named *clfftgen*. The codelets are straight-line code which deliver optimal performance by reducing the number of register usage, operations, precomputation of constants and so on. The optimal FFT's factorization and performance are obtained by exploring the optimization space at runtime. The proposed framework is fully described in the following subsections.

3.1 Code Generator Module

The base components of MPFFT library are its codelets, which are long blocks of highly optimized straight-line codes. At runtime, the given FFT problem will be factorized until the size is small enough to be directly computed by a codelet. Thus, a large FFT problem is computed by a series of codelets which are composed in a specific way, and the codelets library plays a critical role in MPFFT's performance. Since it is very tedious and complicated to implement and tune the codelets by hand when the transform size is larger than 5, we choose an automatic code generation approach. Our code generator is a special-purpose FFT compiler, which produces the codelets automatically from a simple script file at installation time. Similar to UHFFT[26] and FFTW's[20] code generator, *clfftgen* consists of four components: initializer, optimizer, scheduler and unparser. The overall framework of *clfftgen* is given in Fig.4(b). The function of each component is described in detail as follows.

• *Initializer*. The main work of initialization is to produce an internal representation of the codelet in the form of an expression list, or an expression directed acyclic graph (DAG). For the given parameters, the best algorithm and factorization policy is chosen to reduce the number of floating-point operations. The DAG is produced according to well-known FFT algorithms: *mixed-radix*, *split-radix*, *prime factor*, and *rader*. In fact, *clfftgen* is able to produce codelet of arbitrary size.

• *Optimizer*. In most cases, the minimal number of arithmetic operations means the least execution time. The optimizer applies local rewriting rules to each node of DAG, mainly to do some arithmetic optimizations, such as constant folding (multiplications by 0 or 1, adds by 0, multiplications and adds of two constants), common subexpression elimination, and so on.

• *Scheduler*. All large-size FFT transforms are solved by factorizing them into smaller transforms. Those smaller transforms correspond to simple *function* type nodes in the DAG. According to target system's specific parameters, we can set the nesting depth of blocks, which determines whether to unfold the inner block to the outer. The efficiency of generated code depends greatly on register usage. Enhancing the blocking of codelet can reduce the number of temporary variables, which will improve register's usage. Without violating the constraints of data dependency, the scheduler applies a topological sort of the DAG, then transforms the DAG to an equivalent expression list, whose node corresponds to an instruction exactly.

• *Unparser*. According to the sorted expression list, the unparser is to translate them to straight-line code without any branch and jump. The generated code of *clfftgen* is implemented totally with built-in *float*2 data type in OpenCL, thus it can run on any machine which enables OpenCL programming. For improving performance, we define each codelet as a macro to avoid extra overhead of function call. Take size 12 as an example, the chosen factorization policy and its codelet code are shown in Fig.5

```
#define fftKernel12(a, dir)                                   \
{                                                             \
 float2 t12_0;                                                \
 float2 C12_1 = (float2)(0.86602540f, 0.5f*(dir));            \
 float2 C12_2 = (float2)(0.5f, 0.8660254f*(dir));             \
 float2 C12_3 = (float2)(0.0f, 1.0f*(dir));                   \
 float2 C12_4 = (float2)(0.5f, -0.86602540f*(dir));           \
 fftKernel4S((a)[0], (a)[3], (a)[6], (a)[9], dir);            \
 fftKernel4S((a)[1], (a)[4], (a)[7], (a)[10], dir);           \
 fftKernel4S((a)[2], (a)[5], (a)[8], (a)[11], dir);           \
 (a)[4] = complexMul(C12_1, (a)[4]);                          \
 (a)[7] = complexMul(C12_2, (a)[7]);                          \
 (a)[10] = complexMul(C12_3, (a)[10]);                        \
 (a)[5] = complexMul(C12_2, (a)[5]);                          \
 (a)[8] = -complexMul(C12_4, (a)[8]);                         \
 (a)[11] = -(a)[11];                                          \
 fftKernel3((a), dir);                                        \
 fftKernel3((a)+3, dir);                                      \
 fftKernel3((a)+6, dir);                                      \
 fftKernel3((a)+9, dir);                                      \
 bitreverse12(a);                                             \
}
```

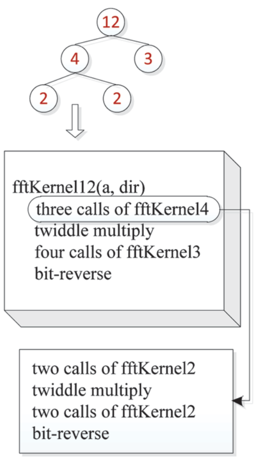


Fig.5. Size 12 and its factorization in codelets library.

Fig.6 describes the performance of codelets library generated by our code generator module on ATI 5 850 with 64 threads per work-group in the GPU kernel. The performance of a codelet is dominated by the number of registers in a CU as the size increases. Larger radices reduce the total number of iterations and temporary variables required to combine the smaller size FFTs, which in turn consume more on-chip resources, especially excessive use of registers. Due to high register pressure, there is substantial performance degradation when the size is larger than 16. Accordingly, we define the maximum base radix as 16, and any size larger than it would be split.

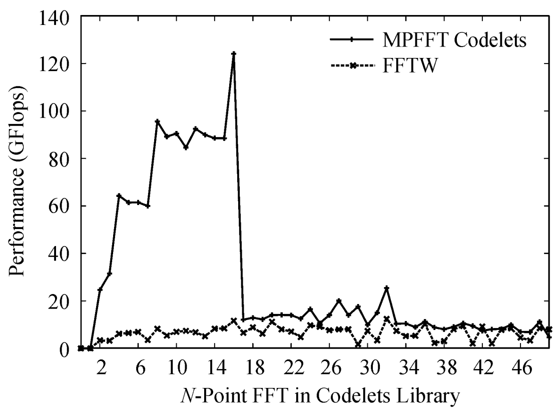


Fig.6. Performance of codelets library on ATI 5850.

3.2 Runtime Module

At runtime, the initialization module first accepts the required parameters. Some parameters come from the input, such as DFT length, dimension, batch size, transform direction and complex data format (planar or interleaved storage), and others are derived from microbenchmarks, such as the amount of banks interleaved in local memory and register file capacity. Then it constructs many FFT plans and each represents a factorization for the given transform size. The runtime code generator produces GPU kernels to be executed in the next step. Subsequently, the search module evaluates the performance of executed plans to select the optimal plan. Furthermore, we provide an empirical value for the performance of a given size. While the performance of the evaluated plan is below that value, the search engine would change some parameters in the initialization stage, such as work-group size, work-group number and the maximum radix in a plan, then repeats aforementioned process. The optimal parameters can be assembled by iteratively compiling and evaluating the various plans. Finally, the search module generates the performance data that contains the information of all evaluated plans besides the selected plan and reuses it in the later sessions. The detailed memory access pattern and optimization techniques in the code generator are presented in the following subsections, and the search algorithm in the runtime module is also specified.

$$N = \prod_{k=1}^{n} n_k, \quad N(k) = n_1 n_2 \dots n_k,$$
$$P(k) = \prod_{j=1}^{k-1} n_j, \quad R(k) = \prod_{j=k+1}^{n} n_j. \tag{8}$$

3.2.1 *Implementation of FFT Kernels*

Assume that N-point FFT is computed by n FFT kernels with radix n_k as shown in (8).

For each fixed $1 \leqslant i \leqslant n$,(6) consists of three computational steps listed as below.

$$\begin{aligned} A_1 &: \boldsymbol{x}_1 \mapsto (\boldsymbol{L}_{n_i}^{m_i n_i} \otimes \boldsymbol{I}_{l_i})x, \\ A_2 &: \boldsymbol{x}_2 \mapsto (\boldsymbol{T}_{n_i}^{m_i n_i} \otimes \boldsymbol{I}_{l_i})x_1, \\ A_3 &: \boldsymbol{x}_3 \mapsto (\boldsymbol{DFT}_{n_i} \otimes \boldsymbol{I}_{N/n_i})x_2. \end{aligned} \quad (9)$$

We describe how to map the formula $(\boldsymbol{L}_{n_i}^{m_i n_i} \otimes \boldsymbol{I}_{l_i})$ to a GPU kernel. The input vector $\boldsymbol{x}$ is regarded as an $m_i \times r_i$ $(r_i = n_i \times l_i)$ matrix stored in the row-major layout. The effect of this stride permutation on $\boldsymbol{x}$ is to perform the following reordering:

$$\boldsymbol{x} = \begin{pmatrix} a_0 & \cdots & a_{n_i} - 1 \\ \vdots & \ddots & \vdots \\ a_{(m_i-1)n_i} & \cdots & a_{m_i n_i - 1} \end{pmatrix}$$

$$\xrightarrow{A_1} \begin{pmatrix} a_0 & \cdots & a_{(m_i-1)n_i} \\ \vdots & \ddots & \vdots \\ a_{n_i-1} & \cdots & a_{m_i n_i - 1} \end{pmatrix}, \quad (10)$$

where a_i denotes the elements of $\boldsymbol{x}$ with indices from il_i to $(il_i + l_i - 1)$. The output matrix $\boldsymbol{x}_1$, of size $n_i \times m_i$, is the transposed matrix of $\boldsymbol{x}$. According to its definition, $\boldsymbol{T}_{n_i}^{m_i n_i}$ is a diagonal matrix of size $m_i n_i$ and thus $(\boldsymbol{T}_{n_i}^{m_i n_i} \otimes I_{l_i})$ is also a diagonal matrix of size N, with each diagonal element repeated I_{l_i} times. Accordingly, step A_2 simply scales $\boldsymbol{x}$ with powers of the primitive root of unity ω. On the other hand, step A_3 is a list of basic butterflies with stride size N/n_i.

3.2.2 Global Memory Access Pattern

With so many parallel threads accessing memory simultaneously, effective utilization of the memory hierarchy has significant impact on performance. Global memory is an off-chip memory space with latencies on the order of hundreds of cycles. To improve bandwidth of global memory, the memory controller will coalesce accesses to multiple data into a single memory transfer if the accessed data has spatial reuse within the size of a memory transfer.

The requirements for coalesced access are different across devices of different compute capabilities in NVIDIA GPUs (refer to [15] for detailed information), and this architectural features affect optimization decisions. The Tesla C2050 is based on the new Fermi architecture (show in Fig.1). Compared with older GPUs, Fermi introduces more banks in local memory (32 banks vs 16 banks), 64 KB on-chip configurable local memory and L1 cache (48 KB vs 16 KB), more coalescing threads (32 threads vs 16 threads) and fast atomic memory operations with ECC memory support. Those new features are incorporated into the optimization through the parametrization of the algorithm. Concurrent kernel execution is not applicable to FFT because of data dependency among the computation in each dimension. Fermi's newly added $L1$ and $L2$ caches are not really helpful for FFT because of FFT's highly irregular and non-repeating data access pattern.

We evaluate the performance of global memory copies with different strides and offsets, and the result is shown in Fig.7 (the word accessed by a thread is 8-byte

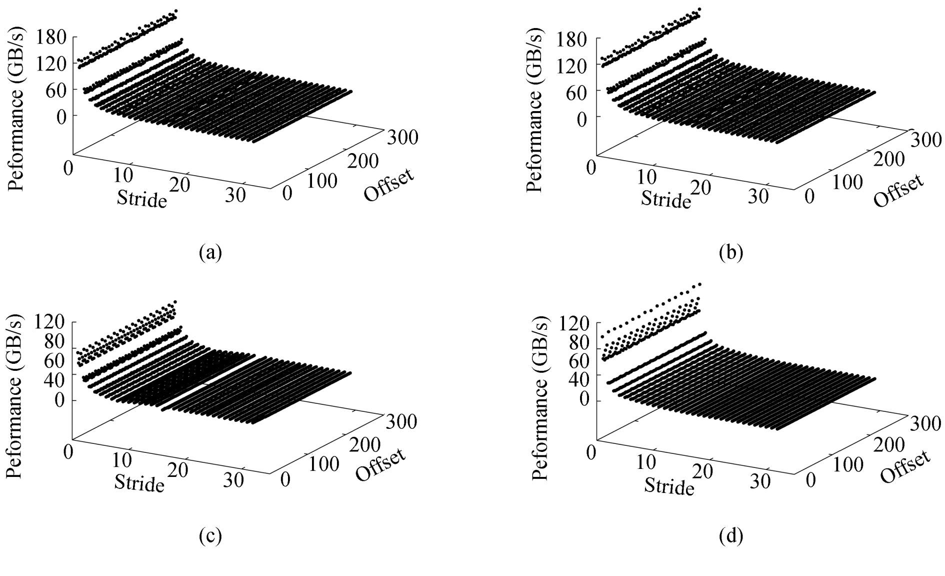


Fig.7. Performance comparison of global memory copies with various strides and starting address offsets on GPUs. (a) ATI 5850 GPU. (b) ATI 6870 GPU. (c) Tesla C1060 GPU. (d) Tesla C2050 GPU.

wide). It is observed that unaligned starting addresses and non-unit strides across threads lead to significant degradation in performance. As shown in Fig.8, we also estimate the efficiency of data transfers with different vector types on both AMD and NVIDIA GPUs. For NVIDIA GPUs, the vector types that achieve peak performance are *char*4, *short*4, *int*2, *float*2 and *double*1. Global memory instructions support reading or writing 1-, 2-, 4-, 8-, 16-byte wide words[24] and the size of vector data type that is larger than 16-byte, such as *double*4, *float*8 and *short*16, has extremely poor performance.

Due to not taking memory access alignment into account, the performances of *short* and *char* vector types are also very poor. Fig.8(a) and Fig.8(b) shows that there are few performance differences in all vector lengths except *char*1, *char*2 and *short*1 on AMD GPUs. There are two independent memory paths between the compute units and the memory on AMD GPUs, namely *fastpath* and *completepath* respectively. When a kernel has atomic operations or sub 32-bit data transfers, the kernel will use the completepath. The maximum bus utilization between the shader unit and the memory unit for the *completepath* is 25% compared to the 100% for the *fastpath*[25]. Consequently, the global memory access efficiency of *char*1, *char*2 and *short*1 is very low.

The formulas that specify the data access from global memory at the beginning and store back the results at the end are $\boldsymbol{L}^{N}_{N/n_1}$ and $\boldsymbol{L}^{N}_{n_1}$ respectively. Each work-item deals with one of N/n_1 and n_1 segments respectively, accesses each point of data at the stride N/n_1 and n_1. The symbols and their definitions are listed at Table 1. If $128\,\text{byte}/(N/N(1) \times$ *sizeof*(*accessed_word*)$) < 1$ or the multiplicative product of the number of work-items in a work-group and *sizeof*(*accessed_word*) is less than 128 byte, the accessed region will not satisfy the above coalescing requirements. The local memory is used to rearrange data properly for assisting coalesced access. Furthermore, if *coalesce_width*$|(N/n_1)$ or *coalesce_width*$|n_1$ is satisfied, the accesses will be properly aligned.

3.2.3 Local Memory Access Pattern

In GPUs, large interleaved local memory is used for inter-thread communication within a work-group. Because of the interleaved design of memory banks, multiple threads fetch or store concurrently with unit stride operates at full speed, since each word resides on a different bank. Namely, any memory load or store of n addresses that spans n distinct memory banks can be served simultaneously, yielding an effective bandwidth that is n times as high as the bandwidth of a single bank.

To avoid bank conflicts, CW consecutive threads are required to access different banks. Tesla C2050 (Fermi architecture) has 32 banks with 4-byte width, and local memory accesses are issued per warp ($CW = 32$) not half-warp ($CW = 16$) in GPUs prior to Fermi architecture, such as Tesla C1060 with just 16 banks. However,

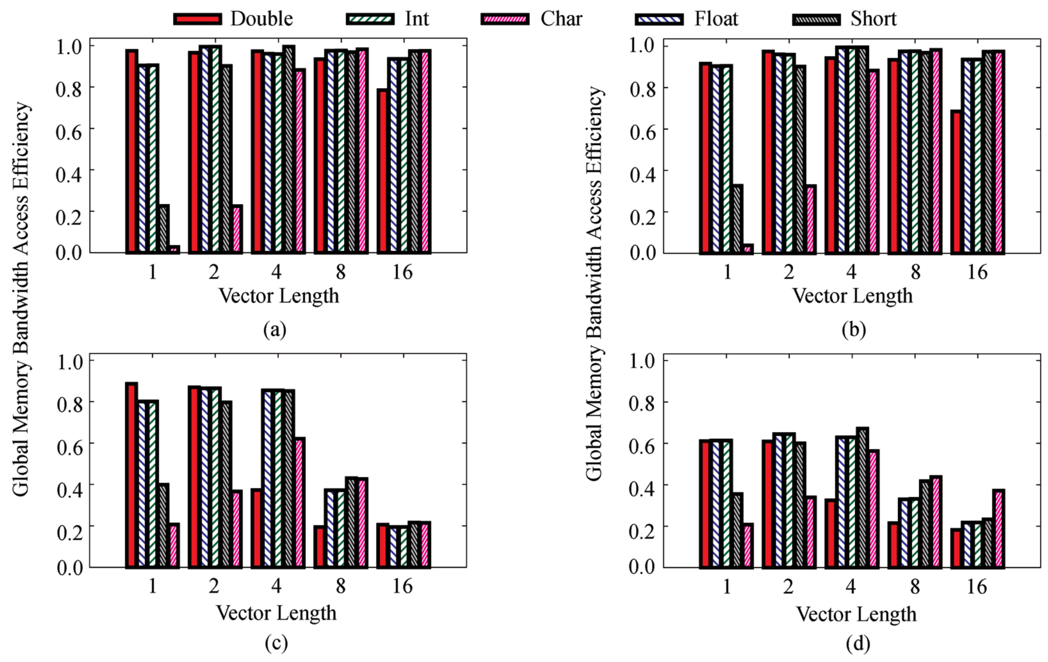


Fig.8. Comparison of global memory bandwidth efficiency with different vector types. (a) ATI 5850 GPU. (b) ATI 6870 GPU. (c) Tesla C1060 GPU. (d) Tesla C2050 GPU.

Table 1. Symbols and Their Definition

Symbol	Definition	GPU			
		Tesla C1060	Tesla C2050	ATI 6870	ATI 5850
accessed_word	Word size of each thread accessed (byte)				
$\vert$	Divisible (if $b = 3a$, then $a\vert b$)				
CW	Work-item size issued in a local memory accesses	16	32	16	16
coalesce_width	Work-item size requested in a global memory access	16	32	16	16
bank_num	Bank size in a local memory	16	32	32	32
workgroup_size	Work-item size in a workgroup				
workgroup_num	Workgroup size in a kernel				
$\boldsymbol{L}$	Stride permutation, $\boldsymbol{L}_{n_i}^{m_i n_i}$				
$\boldsymbol{S}$	Stride permutation, $\boldsymbol{S}_{m_i}^{m_i n_i} = \boldsymbol{L}_{n_i}^{m_i n_i}$				
threadId	Thread Id				
$\#T$	Threshold for multi-dimensional decomposition in the global memory if the size is larger than $\#L$				
$\#L$	Maximum size supported in the local memory, such as 2 048, 4 096.				

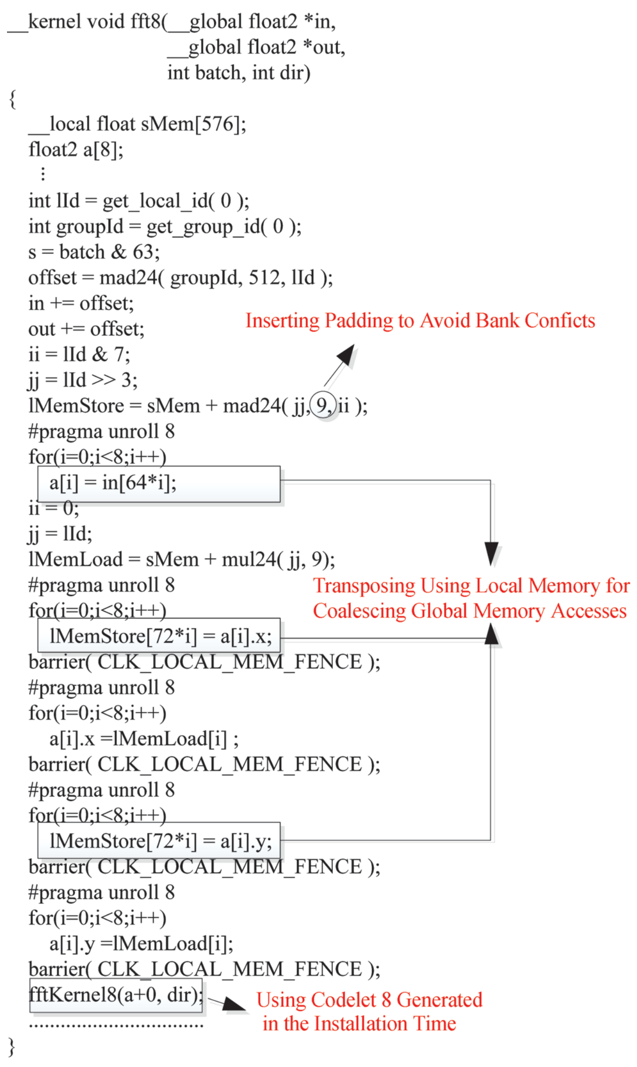
CW is just a quarter-wavefront ($CW = 16$) on both ATI 5850 and 6870 with 32 banks. The feature that each thread can access two banks simultaneously benefits the vector-based applications on its platforms.

Each thread loads one data from every $P(k) \times R(k)$ n_k-tuple sub-array to compute n_k-point FFT and stores back to local memory. The formula to store the results in local memory is $\boldsymbol{S}_{N/n_k}^{N} = (\boldsymbol{L}_{N/n_k}^{N})^{-1}$ (Table 1 shows the definitions of all symbols). The indices are $threadId + N/n_k \times r$ ($r \in [0, n_k)$, $threadId \in [0, N/n_k - 1)$). The indices are consecutive, and if the size of work-group is larger than N/n_k, the kernel computes multiple FFTs at a time. We need to partition the multiple batches FFT in work-items using local memory for exchanging data. Bank conflicts will arise if n_k is power-of-two, and the problem can be solved by inserting appropriate padding. $\boldsymbol{I}_{p(k)} \otimes \boldsymbol{L}_{R(k)}^{n_k R(k)}$ is the formula to load the data from local memory for computing n_k-point FFT and $P(k) \otimes n_k \otimes R(k)$ indicates the indices for fetching data along the n_k dimension. The threads read one $R(k)$-tuple vector with $R(k)$ stride while skipping $R(k) \times (n_k - 1)$ elements. This may suffer from bank conflicts, and the code generator inserts padding after every $R(k) \times (n_k - 1)$ elements to deal with them.

$$T_{\min}(N) = \begin{cases} 0, & \text{if } N = 1, \\ \min(T_{\min}(N/n_i) + T(n_i)), & \text{if } \forall n_i | N. \end{cases} \tag{11}$$

3.2.4 GPU Kernel Generated in the Runtime

Fig.9 presents the sample of final kernel executed on GPUs. Among the low-level optimization techniques, we highlight loop unrolling, constant propagation, branching and divergence. Constant propagation can avoid unnecessary arithmetic instructions, especially when computing padding functions, and the controlling condition is configured to align with CW.



```
__kernel void fft8(__global float2 *in,
                   __global float2 *out,
                   int batch, int dir)
{
  __local float sMem[576];
  float2 a[8];
    ⋮
  int lId = get_local_id( 0 );
  int groupId = get_group_id( 0 );
  s = batch & 63;
  offset = mad24( groupId, 512, lId );
  in += offset;
  out += offset;
  ii = lId & 7;
  jj = lId >> 3;
  lMemStore = sMem + mad24( jj, 9, ii );
  #pragma unroll 8
  for(i=0;i<8;i++)
    a[i] = in[64*i];
  ii = 0;
  jj = lId;
  lMemLoad = sMem + mul24( jj, 9);
  #pragma unroll 8
  for(i=0;i<8;i++)
    lMemStore[72*i] = a[i].x;
  barrier( CLK_LOCAL_MEM_FENCE );
  #pragma unroll 8
  for(i=0;i<8;i++)
    a[i].x =lMemLoad[i] ;
  barrier( CLK_LOCAL_MEM_FENCE );
  #pragma unroll 8
  for(i=0;i<8;i++)
    lMemStore[72*i] = a[i].y;
  barrier( CLK_LOCAL_MEM_FENCE );
  #pragma unroll 8
  for(i=0;i<8;i++)
    a[i].y =lMemLoad[i];
  barrier( CLK_LOCAL_MEM_FENCE );
  fftKernel8(a+0, dir);
  ................................
}
```

Inserting Padding to Avoid Bank Conficts
Transposing Using Local Memory for Coalescing Global Memory Accesses
Using Codelet 8 Generated in the Installation Time

Fig.9. GPU kernel for size 8 from code generator in the runtime.

Besides those techniques, we also maximize the use of build-in functions, such as *native_sincos* and *mad*, where possibly we also use bit-wise operations to implement integer multiply, divide, and modulus for power-of-two radices.

3.2.5 Runtime Search

The space of FFT schedules consists of algorithms and their factorizations. The size of the search space depends on the number of codelets present in the library. Fig.10 shows the search space of the 16-point FFT. Each of the eight branches of the tree represents a possible factorization for computing FFT. In a naive empirical search scheme, the best schedule has the smallest execution time out of all the possible factorizations after executing the factorizations for the given size. However, the time required to exhaustively search an exponential space of factorizations may be quite large. Furthermore, the exhaustive search scheme is also costly in terms of the workspace requirement. To avoid the drawbacks of an exhaustive search scheme, we take advantage of the recursive structure of the FFT to avoid re-evaluating common subsequences (branches with identical strides) in the factorization tree. The optimal substructure of the problem is reused to characterize it in a recursion, given by (11). The cost of a codelet $T(n_i)$ is represented by its execution time with appropriate input and output strides. The costs of evaluated subsequences for specific strides are stored in a lookup table.

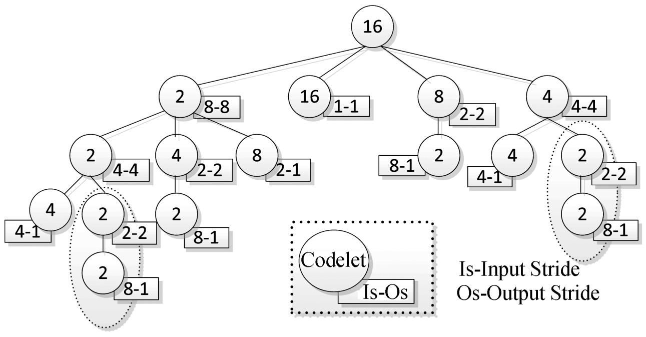


Fig.10. Search space for FFT of size 16.

We use dynamic programming to evaluate the tree of factorizations in a bottom up fashion. The best factorization can be found by selecting the module that yields the $T_{\min}(N)$ for computing the FFT. The pseudo-code of the search scheme is given in Fig.11. Compared with our hand-tuning FFT, the auto-tuned library delivers higher performance using the heuristic algorithm to search the optimal schedule. Fig.12 shows that there is a significant rise in performance by using our auto-tuning technique.

```
void Search($N, is, os$)
begin
  if $S \leftarrow$ TableLookup $(N, is, os)$ then return $S$;
  $S \leftarrow \varnothing$;
  $mincost \leftarrow MAX\_INT$;
  foreach $r_i \in$ {module library} do
    if $r_i = N$ then $S \leftarrow r_i$;
    else $S \leftarrow$ Search $(N/r_i, is, os) + r_i$;
    $cost \leftarrow$ Evaluate $(S)$;
    if $cost = mincost$ then
      TableInsert $(S, N, is, os, cost)$;
      $mincost \leftarrow cost$;
    end
  end
  return TableLookup$(N, is, os)$;
end
```

Fig.11. Search algorithm for the optimal FFT factorization.

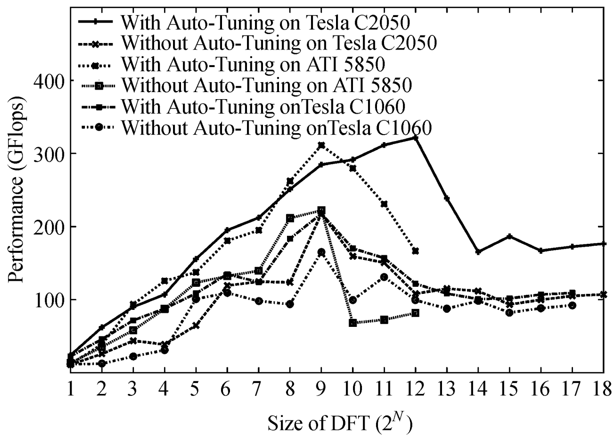


Fig.12. Performance improvement using auto-tuning technique.

4 Performance Results and Analysis

In this section we estimate the performance of our FFT library on different GPUs. Our library supports both in-place and out-of-place FFT computation, batched execution for all dimensional FFTs, and the maximum batch size is limited by global memory. To validate our methodology, the performances of our library are compared with those of FFTW 3.2.2 on all host CPUs, CUFFT 4.0.1 on NVIDIA GPUs and *clAmdFft* 1.4.182 on AMD GPUs. No planning or pre-calculation is needed for our library and the twiddle factors are computed on the fly. Meanwhile, our library assumes that data resides entirely within GPU memory and the auto-tuning framework does not account for memory transfers and out-of-core FFT computations.

Experiments are conducted to evaluate our approach for out-of-place complex to complex FFTs on these platforms with different GPUs which are shown in Table 2 and Table 3 in detail. For a three dimensional (3-D) FFT with the total size $N = N_x \times N_y \times N_z$ and

Table 2. Evaluation Platforms

Platform	CPU	RAM (GB)	GCC	Linux	GPU	GPU SDK	OpenCL
System 1	AMD Phenom II X4 940 with 0.8 GHz	8	4.3.3	Ubuntu 9.04	ATI HD 5850	ATI SDK 2.4	1.1
System 2	Two Intel Xeon X5472 with 3.0 GHz	16	4.4.5	Ubuntu 10.10	Tesla C1060	NVIDIA SDK 4.0.1	1.1
System 3	Two Intel Xeon X5550 with 2.66 GHz	16	4.4.5	Ubuntu 10.10	Tesla C2050	NVIDIA SDK 4.0.1	1.1
System 4	Two Intel Xeon X5550 with 2.66 GHz	16	4.5.2	Ubuntu 11.04	ATI HD 6870	ATI SDK v2.5	1.1

Table 3. Configuration of the GPUs

GPU	Clock Rate	PE	CU	Peak Perf.	Memory	Bus Width	Peak BW	Registers	Local Memory	Driver
	(GHz)			(GFlops)	(GB)	(bits)	(GB/s)	per CU (K)	(KB)	
ATI HD 5850	0.73	288	18	2 088	1.0	256	128.0	16	32	8.80
Tesla C1060	1.30	240	30	933	4.0	512	102.0	16	16	280.13
Tesla C2050	1.15	448	14	1 030	3.0	384	144.0	16	48	280.13
ATI HD 6870	0.90	224	14	2 016	1.0	256	134.4	16	32	11.70

Note: PE: processing element, CU: compute unit, Peak Perf.: peak performance of single floating-point, BW: bandwidth.

with execution time of t seconds, its performance is calculated in GFlops defined by the following equation.

$$\text{GFlops} = \frac{5N_xN_yN_z(\log_2 N_x + \log_2 N_y + \log_2 N_z) \times 10^{-9}}{t}. \tag{12}$$

Fig.13 shows the performance results of 1D batched FFT of size 2^N on these platforms and our automatic performance tuning OpenCL FFT library is called MPFFT. The performance of our library is 1.5 to 4 times the performance of *clAmdFft* library, closes to CUFFT library, and 6 to 18 times the performance of FFTW library with four threads. Many bank conflicts of local memory accesses in *clAmdFft* library result in suboptimal performance, for example, the percentage of GPU time that local memory is stalled by bank conflicts in the library reaches to 24.7% on ATI 5 850 by profiling the program when FFT size is 1 024 with 1 024 batches and the value of that in our library is 0. For the size larger than what can be computed using local memory FFT, global transposes for data synchronization are needed which result in launching more kernels.

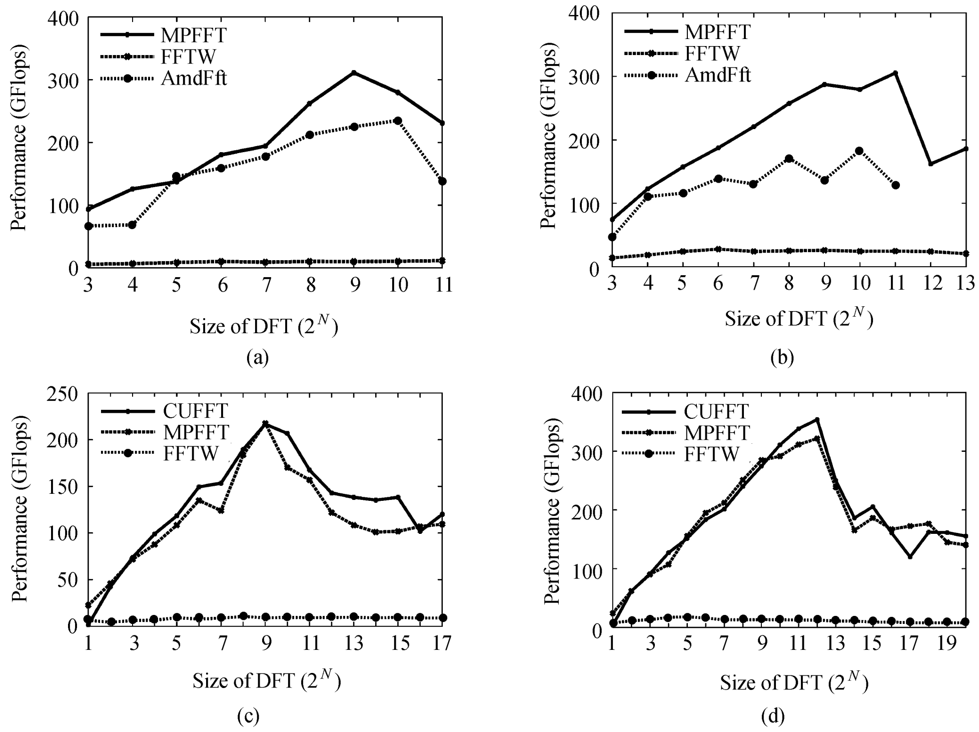


Fig.13. Performance of 1D FFTs on GPUs. (a) ATI 5850 GPU. (b) ATI 6870 GPU. (c) Tesla C1060 GPU. (d) Tesla C2050 GPU.

For these sizes, n is decomposed and we choose an appropriate larger base radix for local memory computation according to (4) for amortizing the cost of the increased memory access. When N is larger than 2^{10} on ATI 5850, 2^9 on Tesla C1060 and 2^{12} on Tesla C2050, the performance begins to degrade respectively because of the limited local memory capacity and the register numbers. In that case, N is divided into several smaller sizes using global memory to exchange data, resulting in multiple kernels are generated during the runtime. Hence, the latency from global memory access leads to the performance decreases. Furthermore, when the size is larger than 4 096 with multiple batches, the execution of *clAmdFft* library on ATI 6 870 fails, so the performance is not shown in Fig.13(b).

The performance of batched 2D FFT of sizes $N_1 \times N_2$ on these four platforms are presented in Fig.14. For multidimensional FFTs, we use row-column FFT algorithm to compute each dimension with multiple batches using (7). Due to the increased number of transpose operations and stride accesses, it is crucial to use the coalesced memory access and bank-conflict free to maximize efficient memory bandwidth. Our auto-tuner can effectively discover the hotspot by analyzing the behavior of memory access, then inserts appropriate padding to avoid reduced bandwidth. Further, a higher occupancy is obtained through increasing the computation ratio in each thread without registers and local memory overflowed to hide the cost of communication. As shown in Fig.14, the performance of our 2D FFT on ATI 5850 is 1.5 to 36.81 times the performance of *clAmdFft* library, and the average performance speedup achieved up to 2.6x over *clAmdFft*, 15.27x over FFTW with 4 threads. Moreover, we achieve average 2.4x, 8.33x maximum speedup over *clAmdFft* on ATI 6870, and 6.52x over FFTW with 8 threads. On NVIDIA GPU, including Tesla C1060 and C2050, the overall performance is comparable to CUFFT library in less than 10%. In addition, the results outperform CUFFT on Tesla C1060 when the FFT size is large enough, such as $2\,048 \times 2\,048$.

Fig.15 presents the performance comparison of 3D FFTs of size $N_1 \times N_2 \times N_3$. Our 3D FFT achieves an average of 1.81x on ATI 5850 and 1.37x on ATI 6870 over *clAmdFft* library. Meanwhile the overall performance is within 90% of CUFFT 4.0 on two NVIDIA GPUs. Furthermore, Fig.16 shows that our library also supports non-power-of-two sizes, and the performance of our 3D FFT with those sizes on ATI 5850 is 1.5 to 28 times over that of FFTW with 4 threads, 20.01x average speedup over CUFFT on Tesla C2050, 31.87x average speedup over CUFFT on Tesla C1060.

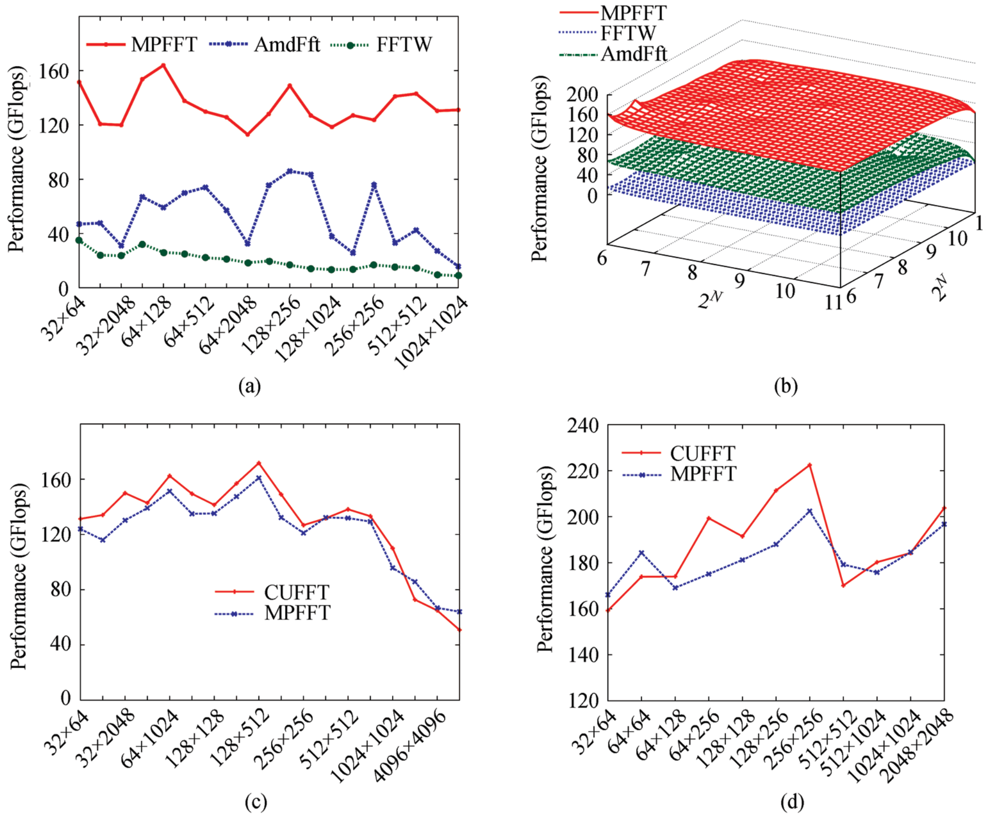


Fig.14. Performance of 2D FFTs on GPUs. (a) ATI 6870 GPU. (b) ATI 5850 GPU. (c) Tesla C1060 GPU. (d) Tesla C2050 GPU.

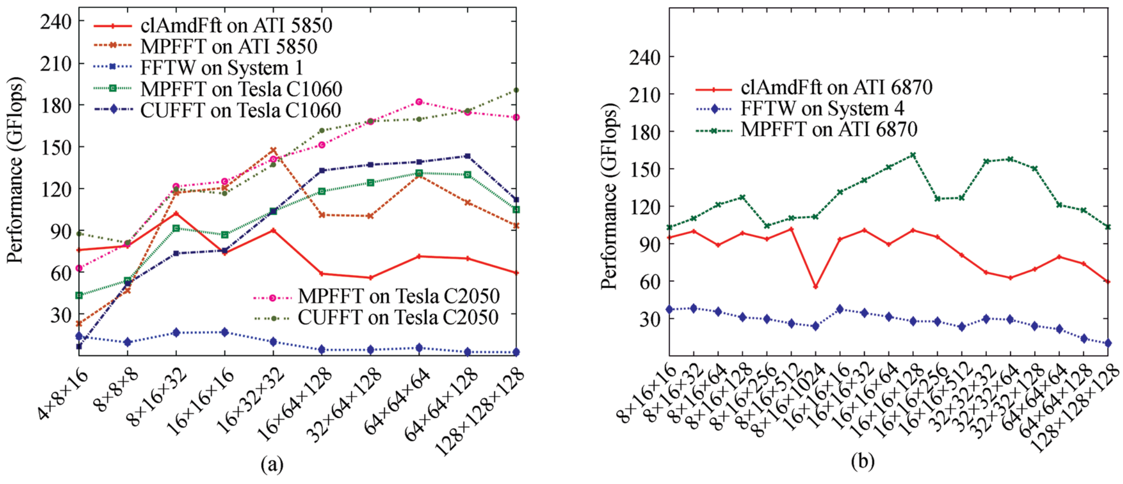


Fig.15. Performance of 3D FFTs on GPUs.

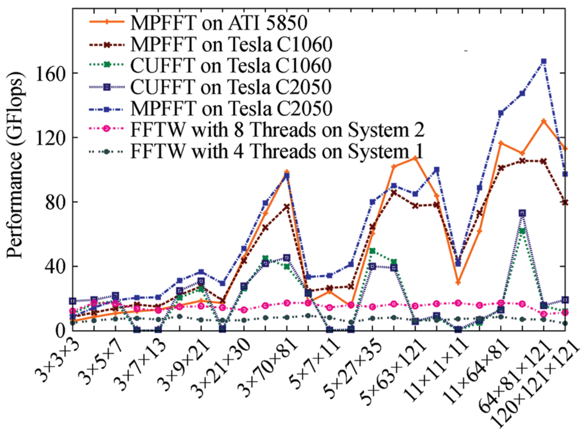


Fig.16. Performance of non-power-of-two FFTs.

5 Related Work

FFT is one of the most widely used algorithms for scientific and engineering computation especially in the fields of signal processing, image processing and data compression. Several FFT libraries on CPUs with automatic performance tuning have been proposed, e.g., FFTW[20-21], SPIRAL[③][22-23] and UHFFT[④][26-28]. Automatic tuning in FFTW is performed in two different levels, namely installation time and runtime. At installation time, the code generator generates highly optimized straight-line FFT code blocks called *codelets*. At runtime, the pre-generated codelets are assembled in a plan to compute large FFT problem size. The methodology of auto-tuning techniques in the UHFFT is similar to FFTW. SPIRAL is a program generation and optimization system which generates optimized codes for Digital signal processing (DSP) transforms. It employs 3-stage adaptation methodology to adapt to various styles of architecture. In the first stage, the mathematical rules and identities have been used for the formula generator in virtue of a special purpose pseudo-mathematical language called SPL (Signal Processing Language) to expand and optimize the FFT formula to a given transform. Then, the optimized SPL formula is translated into source code on a specific platform. Finally, the source code is compiled and evaluated to generate the best code by guiding the code generation process.

Graphics processors traditionally have high specialized programming models and interfaces that limit the ability of developers to map general-purpose applications to these platforms with a massively parallel. The advent of CUDA and OpenCL have led to decrease the complexity of programming on GPUs, and there has been growing research in exploring auto-tuning techniques for improving performance of algorithms such as SpMV (Sparse Matrix-Vector Multiplication), GEMM (General Matrix Multiply) and FFTs on CUDA GPUs, and moreover, several studies have been conducted to optimize the performance manually. Nukada and Matsuoka[3] presented an auto-tuning algorithm for optimizing 3D FFT algorithms on CUDA GPUs. Their algorithm optimizes the number of threads and resolves bank conflicts on local memory especially. However, the larger size of FFTs may leads to suboptimal performance as a result of restricting the search space severely. Yuri Dotsenko, Sara S.Baghsorkhi, *et al.*[4] also presented an auto-tuning framework for automatically generated optimized FFT kernels by pruning

[③]http://www.spiral.net/index.html, Sept. 2012.

[④]http://www2.cs.uh.edu/~ayaz/uhfft/, Sept. 2012.

heuristics significantly to reduce the optimization search space. Although the work they did demonstrated significant performance improvement against prior state-of-the-art FFT algorithms on GPUs, they just took the NVIDIA GPU into account and ignored the AMD GPU. Hence the libraries they developed were not adaptive to various GPU platforms. Furthermore, there has been a significant body of work on FFT, but most of the work is not extended to non-power-of-two sizes.

6 Conclusions and Future Work

In this paper, we have presented an automatic performance tuning framework for generating optimized FFT kernels, and the framework is well-suited to both AMD and NVIDIA GPUs. Our library named MPFFT (Massively Parallel FFT) based on this framework was implemented and achieves a high performance. For power-of-two length FFTs, our library achieves an average speedup of 1.3x for 1D FFT, 2.6x for 2D FFT and 1.57x for 3D FFT on ATI 5850, 1.6x for 1D FFT, 2.4x for 2D FFT, and 1.37x for 3D FFT on ATI 6870 over *clAmdFft* 1.4. The overall performance is within 90% of CUFFT 4.0 on Tesla C1060 and Tesla C2050. Furthermore, our library also supports non-power-of-two sizes. For 3D non-power-of-two FFTs, our library delivers 1.5x to 28x times faster than FFTW with 4 threads and 20.01x average speedup over CUFFT 4.0 on Tesla C2050. Our analysis and structured memory optimization techniques are based on the Kronecker product, which captures the memory access pattern and other performance effects of major GPU micro-architecture features in our library. Meanwhile it can also account for the differences in the underlying hardware and programming language constructs of these two platforms.

As future prospects, we intend to apply the framework on other OpenCL devices, such as Cell, APU and Intel processors with Sandy Bridge architecture, and continue to optimize the performance of our FFT library. We also plan to integrate other existing adaptive approaches to our framework to benefit well from the power of these techniques, and construct a novel performance model with data training or machine learning to attain a good trade-off between performance and search time. Finally, we would like to extend our library for datasets that do not fit into GPU memory.

References

[1] Duhamel P, Vetterli M. Fast fourier transforms: A tutorial review and a state of the art. *Signal Processing*, 1990, 9(14): 259-299.

[2] Govindaraju N K, Lloyd B, Dotsenko Y, Smith B, Manferdelli J. High performance discrete Fourier transforms on graphics processors. In *Proc. SC*, Nov. 2008, Article No.2.

[3] Nukada A, Matsuoka S. Auto-tuning 3-D FFT library for CUDA GPUs. In *Proc. SC*, Nov. 2009, Article No.30.

[4] Dotsenko Y, Baghsorkhi S S, Lloyd B, Govindaraju N K. Auto-tuning of fast Fourier transform on graphics processors. In *Proc PPoPP*, Feb. 2011, pp.257-266.

[5] Gu L, Li X M, Siegel J. An empirically tuned 2D and 3D FFT library on CUDA GPU. In *Proc. the 24th ICS*, June 2010, pp.305-314.

[6] Gaster B, Howes L, Kaeli D R, Mistry P, Schaa D. Heterogeneous Computing with OpenCL. San Fransisco, USA: Morgan Kaufmann, 2011.

[7] Munshi A, Gaster B, Mattson T G, Fung J, Ginsburg D. OpenCL Programming Guide. Boston, USA: Addison-Wesley Professional, 2011.

[8] Zhang E Z, Jiang Y L, Guo Z Y, Shen X P. Streamlining GPU applications on the fly: Thread divergence elimination through runtime thread-data remapping. In *Proc. the 24th ICS*, June 2010, pp.115-126.

[9] Yang Y, Xiang P, Kong J F, Zhou H Y. A GPGPU compiler for memory optimization and parallelism management. In *Proc. PLDI*, June 2010, pp.86-97.

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

[11] Van Loan C. Computational Frameworks for the Fast Fourier Transform. Philadelphia, USA: SIAM, 1992.

[12] Johnson J, Johnson R W, Rodriguez D, Tolimieri R. A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures. *Circuits, Systems and Signal Processing*, 1990, 9(4): 449-500.

[13] Franchetti F, Püschel M, Voronenko Y, Chellappa S, Moura J M F. Discrete Fourier transform on multicore. *IEEE Signal Processing Magazine*, 2009, 26(6): 90-102.

[14] Tolimieri R, An M, Lu C. Algorithms for Discrete Fourier Transforms and Convolution. Berlin: Springer-Verlag, 1989.

[15] NVIDIA Corporation. NVIDIA CUDA compute unified device architecture — Programming guide version 4.2. 2008, http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation, Sept. 2012.

[16] Ji F, Ma X S. Using shared memory to accelerate MapReduce on graphics processing units. In *Proc. IPDPS*, May 2011, pp.805-816.

[17] Jiao Y, Lin H, Balaji P, Feng W. Power and performance characterization of computational kernels on the GPU. In *Proc. GreenCom-CPSCOM*, Dec. 2010, pp.221-228.

[18] Baghsorkhi S S, Delahaye M, Patel S J, Gropp W D, Hwu W W. An adaptive performance modeling tool for GPU architectures. In *Proc. the 15th PPoPP*, May 2010, pp.105-114.

[19] Schwarztrauber P N. Multiprocessor FFTs. *Parallel Computing*, 1987, 5: 197-210.

[20] Frigo M, Johnson S G. The design and implementation of FFTW3. In *Proceedings of the IEEE*, 2005, 93(2): 216-231.

[21] Frigo M. A fast Fourier transform compiler. In *Proc. PLDI*, May 1999, pp.169-180.

[22] Mesmay F, Franchetti F, Voronenko Y. Encyclopedia of Parallel Computing. Berlin: Springer, 2011.

[23] de Mesmay F, Voronenko Y, Püschel M. Offline library adaptation using automatically generated heuristics. In *Proc. IPDPS*, Apr. 2010, pp.1-10.

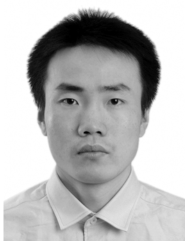
[24] Kirk D B, Hwu W W. Programming Massively Parallel Processors: A Hands-on Approach. San Fransisco, USA: Morgan Kaufmann, 2010.

[25] Purnomo B, Rubin N, Houston M. ATI stream profiler: A tool to optimize an OpenCL kernel on ATI radeon GPUs. In *Proc. SIGGRAPH*, July 2011, pp.26-30.

[26] Mirkovic D, Johnsson S L. Automatic performance tuning in the UHFFT library. In *Proc. ICCS*, May 2001, pp.71-80.

[27] Ali A, Johnsson L, Mirkovic D. Empirical auto-tuning code generator for FFT and trigonometric transforms. In *Proc. ODES*, Mar. 2007.

[28] Mirkovic D, Mahasoom R, Johnsson L. An adaptive software library for fast Fourier transforms. In *Proc. the 14th ICS*, May 2000, pp.215-224.



Yan Li is currently pursuing the Ph.D. degree in Institute of Software, Chinese Academy of Sciences, Beijing. His research interests focus on the development of portable libraries for applications on GPU-based heterogeneous computing platforms, such as FFT and data-parallel algorithms.

Yun-Quan Zhang is currently a professor in Institute of Software, Chinese Academy of Sciences, Beijing. His research interests are in the areas of high-performance parallel computing (HPC), with particular emphasis on large-scale parallel computation and programming models, high-performance parallel numerical algorithms, and performance modeling and evaluation for parallel programs. Recently he served as chair of Program Committee of IEEE CSE 2010, vice-chair of Program Committee of High-Performance Computing China (2008∼2012), member of Steering Committee of International Supercomputing Conference 2012, member of Program Committee of IEEE ICPADS 2008, ACM ICS 2010, IEEE IPDPS 2012, and IEEE CCGRid 2012. He also organizes and distributes China's TOP100 List of High Performance Computers, which traces and reports the development of the HPC system technology and usage in China.

Yi-Qun Liu now is pursuing the Ph.D. degree in Institute of Software, Chinese Academy of Sciences, Beijing. Her research mainly concentrates on the design and optimization of parallel FFT algorithms, code generation and software adaption.

Guo-Ping Long received his Ph.D. degree in Institute of Computing Technology, Chinese Academy of Sciences, Beijing, in 2010. He is currently an assistant professor in Institute of Software, Chinese Academy of Sciences. His major research interests include parallel algorithms and programming on multi/many-core architectures.

Hai-Peng Jia now is pursuing the Ph.D. degree in School of Information Science and Engineering, Ocean University of China, Qingdao. His research mainly concentrates on the design and optimization of parallel algorithms, computer vision and computer graphics.