A Hybrid GPU/CPU FFT Library for Large FFT Problems

Shuo Chen
Dept. of Electrical and Computer Engineering
University of Delaware
Newark, DE 19716
schen@udel.edu

Xiaoming Li
Dept. of Electrical and Computer Engineering
University of Delaware
Newark, DE 19716
xli@udel.edu

***Abstract*—Graphic Processing Units (GPU) has been proved to be a promising platform to accelerate large size Fast Fourier Transform (FFT) computation. However, GPU performance is severely restricted by the limited memory size and the low bandwidth of data transfer through PCI channel. Additionally, current GPU based FFT implementation only uses GPU to compute, but employs CPU as a mere memory-transfer controller. The computing power of CPUs is wasted. This paper proposes a hybrid parallel framework to use both multi-core CPU and GPU in heterogeneous systems to compute large-scale 2D and 3D FFTs that exceed GPU memory. This work introduces a flexible partitioning scheme that enables concurrent execution of CPU and GPU and integrates several FFT decomposition paradigms to tailor computation and communication. Moreover, our library exposes and exploits previously overlooked parallelism in FFT. Optimal load balancing is automatically achieved from effective performance modeling and empirical tuning process. On average, our large FFT library on GeForce GTX480, Tesla C2070, C2075 is 121% and 145% faster than 4-thread SSE-enabled FFTW and Intel MKL, with max speedups 4.61 and 2.81, respectively.**

I. INTRODUCTION

Fast Fourier Transform (FFT) is one of the most widely used numerical algorithms in science and engineering domains. It is not rare that large scientific and engineering computation, such as large-scale physics simulations, signal processing and data compression, spend majority of execution time on large size FFTs. Such FFT implementations require large amount of computing resources and memory bandwidth. Compared with current multi-core CPUs, Graphical Processing Units (GPUs) have been recently proved to be a more promising platform to solve FFT problems since GPUs have much more parallel computing resources and can often achieve an order of magnitude performance improvement over CPUs on compute-intensive applications [1].

Efforts have been focused on solving in-card FFT problems whose sizes can fit into the device memory of GPU. It means that only two simple data transfers are needed in the solving of one FFT problem, one copying all the source data from CPU memory to GPU memory using the PCI bus, and the other copying all the results back. Since the data transfer does not have much to optimize, the prior works focused on the decomposition of FFT problems for the two-level organization of processing cores on GPU and the efficient usage of GPU on-device memory hierarchy. Libraries such as CUFFT from NVIDIA [2], Nukada's work on 3D FFT [3], [4], Govindaraju's [5], [6] and Gu's work on 2D and 3D FFT [1] can be classified into this group.

Recently, Gu et.al. [7] demonstrated a GPU-based out-of-card FFT library that can solve FFT problems larger than GPU device memory. Since one data transfer cannot move all data between CPU and GPU, multiple data transfers are needed. Gu et.al. proposed a joint optimization paradigm that co-optimizes the communication and the computation phases of FFT, and an empirical searching method to find the best tradeoff between the two factors. For even larger FFT problems, Chen et.al. presented a GPU cluster based FFT implementation [8]. However, the work has been almost exclusively focused on the optimization of communication over inter-node channels.

In spite of highly influential results in prior FFT work on GPUs, no matter the on-card FFT libraries, the out-of-card FFT libraries or the GPU-cluster based solutions, the real performance of GPUs is not significantly higher than that of current high-performance CPUs as expected, since GPU performance is severely restricted by the limited GPU memory size and the low bandwidth of data transfer between CPU and GPU through PCIe channel. In addition, in the prior FFT research on GPUs, CPU is only used as a memory or communication controller, that is, managing memory transfer requests between CPU and GPU, or between nodes. The computing power of CPUs is wasted. Therefore, the objective of this paper is to propose a hybrid parallel FFT framework to concurrently execute both CPU and GPU for computing large-scale FFTs that exceed GPU memory. Incorporating CPU has several advantages: (1) Multi-core CPU is capable of computing partial work concurrently with GPU to release the pressure which is originally assigned to the GPU, and to make full use of available underlying computing resources for performance improvement, (2) CPU helps increase data transfer bandwidth remarkably by enabling efficient utilization of PCIe bus and save much GPU memory resource since partial work is kept into local CPU memory to execute without being transferred to GPU, and (3) CPU has much larger memory size than that of GPU in general.

Ogata et.al. [9] recently attempted to divide the computation to both CPU and GPU, though targeting at problems whose sizes can fit into the GPU memory. The small problem assumption makes the optimization of data communication between CPU and GPU trivial because all data can be copied to GPU in one data copying, which largely avoids the challenges of co-optimizing both computation and communication between two different types of devices. In this paper, we present a hybrid FFT library that engages both CPU and GPU in the solving of large FFT problems that can not fit into the GPU memory. The key problem we solve is to engage heterogeneous computing resources and newly improved optimization strategies in the acceleration of such large FFTs.

Making FFT run concurrently on CPU and GPU comes with significant challenges. First of all, CPU and GPU are two computer devices with totally different performance characteristics. Even though FFT can be decomposed in many different ways, not a single method can arbitrarily divide a problem into subtasks with two different performance patterns. In FFT, a simple change to the division of computation will lead to global effects on the data transfers, because ultimately any single point in the output of a FFT problem is mathematically dependent on *all* input points. We cannot just optimize for CPU or just optimize for GPU. In simpler words, the first problem we need to solve is to divide a FFT workload between two types of computing devices that are connected by a slow communication channel, and to determine the optimal load distribution among such heterogeneous resources.

The second challenge is the magnitude of the vast space of possible hybrid implementations for one FFT problem. In addition to the large number of possible algorithmic transformations, as outlined in the first challenge, CPU and GPU architectural features also need to be considered in the search. Reconciling CPU and GPU architectures is hard because they simply like different styles of computation/communication mix. Moreover, the decision of workload assignment needs to be put into a search space that consists of many different ways of decomposition and different ways of data transfer. In particular, computation and communication can be efficiently overlapped, an important performance booster, only if the data dependency between the CPU parts and the GPU parts is appropriately arranged. In other words, even if we already find the best algorithm for a FFT problem, i.e., the best division of computation, the implementation of the algorithm still needs to be co-tuned for two different architectures.

This paper is the first effort to propose a hybrid implementation of FFT that concurrently executes both multithreaded CPU and GPU in a heterogeneous computer node to compute large FFT problems that cannot fit into GPU memory. The paper makes four main contributions: (1) a hybrid large-scale FFT decomposition framework that enables the extraction and the tailoring of different workload and data transfer patterns appropriate for the two different computing devices, (2) an empirical performance modeling to determine optimal load balancing between CPU and GPU, which estimates performance based on several key parameters, and replaces an exhaustive walk-through of the vast space of possible hybrid implementations of FFT on CPU/GPU with a guided empirical search, (3) an optimizer that exploits substantial parallelism for both GPU and CPUs, and (4) effective heuristics to purposefully expose opportunities of overlapping communication with computation in the process of decomposing FFT. Overall, our hybrid FFT implementation outperforms several latest and widely used large-scale FFT implementations. For instance, our double-precison Tesla C2070 performance is 29% and 1.4× faster than that of Gu et.al.'s [7] and Ogata et.al.'s [9] work, and 1.9× and 2.1× faster than 4-thread SSE-enabled FFTW [10], [11], [12] and Intel MKL [13], with max speedups 4.6× and 2.8×, respectively.

II. Overview of FFT Algorithm

FFT algorithms recursively decompose a N-point DFT into several smaller DFTs [14], and the divide-and-conquer approach reduces the operational complexity of a Discrete Fourier Transform (DFT) from $O(N^2)$ into $O(NlogN)$. There are many FFT algorithms, or in other words, different ways to decompose DFT problems. In this section we briefly introduce the FFT algorithms used in this paper and overview how they are incorporated into our hybrid approach. The DFT transform of an input series $x(n), n = 0, 1, ..., N-1$ of size N is presented as $Y(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk}$. We can map the one dimensional input into two dimensions indexed by l in L dimension and m in M dimension, respectively. The Cooley-Tukey FFT [15] decomposes the original DFT into three sub-steps: (1) Perform M DFTs of size L, $A(p,m) = \sum_{l=0}^{L-1} x(l,m) W_L^{lp}$; (2) Multiply twiddle factors, $B(p,m) = A(p,m) W_N^{pm}$; and (3) Perform L DFTs of size M, $Y(p,q) = \sum_{m=0}^{M-1} B(p,m) W_M^{mq}$; where $W_c^{ab} = e^{-j2\pi ab/c}$ is the twiddle factor introduced by [15]. Therefore, $Y(k) = Y(pM+q)$. In essence, Cooley-Tukey introduces a decomposition approach that divides one dimensional computation into two. Moreover, the Radix algorithm is a special case of the Cooley-Tukey algorithm for power-of-two FFT problems.

In this paper, we extend the I/O tensor representation introduced in FFTW [12] to represent the algorithmic transformation of our hybrid FFTs. An original I/O tensor $d(C, Si, So, I, O)$ denotes FFTs along a data dimension where C is the size of one dimensional FFT, Si and So represent the stride of input and output, and I and O are the addresses of input and output array. t_M^L represents multiplication of twiddle factors with size $L \times M$. The I/O tensor representation captures the two most important factors that determine FFT's performance, i.e., data access patterns and computation load. As an example, the Cooley-Tukey FFT decomposition can be precisely denoted as an extended I/O tensor representation $u = \{d(L, M, M, I, O), t_M^L d(M, 1, 1, O, O)\}$. Here u is an I/O tensor that represents a multidimensional FFT.

III. Our Hybrid GPU/CPU FFT Library

A. Hybrid 2D FFT Framework

Our heterogeneous 2D FFT framework solves FFT problems that are larger than GPU memory. We denote this kind of problems as out-of-card FFTs. Suppose the problem size is $N = Y \times X$, where Y is the number of rows and X is number of columns. Generally 2D FFT involves two rounds of computation, i.e. Y dimensional 1D FFTs along X dimension and then X dimensional 1D FFTs along Y dimension. A 2D FFT for an 2D input $f(y,x)$ of size N is defined in equation (1),

$$\begin{aligned} out(k_y, k_x) &= \sum_{x=0}^{X-1} W_X^{xk_x} \sum_{y=0}^{Y-1} W_Y^{yk_y} f(y,x) \\ &= \sum_{x=0}^{X-1} W_X^{xk_x} \sum_{y=0}^{Y-1} W_Y^{yk_y} f(y, (x_{gpu}, x_{cpu})) \end{aligned} \quad (1)$$

where $x, k_x = 0, 1, ..., X_{gpu}, ..., X-1$; $y, k_y = 0, 1, ..., Y-1$; $x_{gpu} = 0, 1, ..., X_{gpu}-1$; $x_{cpu} = X_{gpu}, ..., X-1$; twiddle factor $W_c^{ab} = e^{-j2\pi ab/c}$.

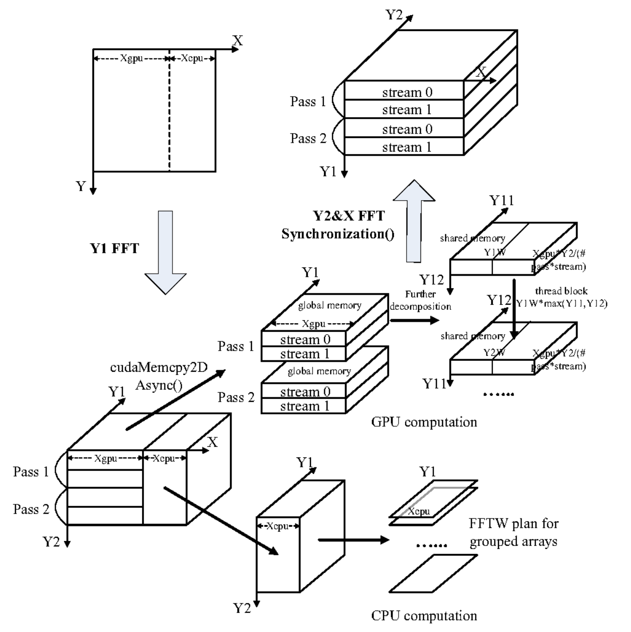

Fig. 1. Overview of hybrid large out-of-card 2D FFT framework.

1) Load Distribution: From equation (1), the work load of 2D FFT in the first round can be distributed into GPU and CPU, respectively. Since Y is computational dimension in the first round, the work ratio of GPU to CPU in round one is denoted as $R_X = \frac{Xgpu}{Xcpu}$ along X dimension, where $Xcpu = X - Xgpu$. The total 2D FFT can be represented as $u_{2d} = \{d(Y,X,X,I,O), d(X,1,1,O,O)\}$ in an extended I/O tensor format. Since performance hurdle will be triggered if the size of FFTs in each round are large, we apply Y dimensional Cooley-Tukey decomposition [15] to exploit more algorithmic parallelism. In our hybrid 2D FFT framework, the extended tensor representation of work distribution on GPU, i.e. u_{gpu}, and on CPU, i.e. u_{cpu}, are transformed as equation (2).

$$\begin{aligned} u_{gpu} =& \{d(Y_1, Y_2Xgpu, Xgpu, Igpu, Ogpu), Sync, \\ & t_{Y_2}^{Y_1} d(Y_2, Y_1X, Y_1X, O, O), d(X,1,1,O,O)\} \\ u_{cpu} =& \{d(Y_1, Y_2Xcpu, Xcpu, Icpu, Ocpu), Sync\} \end{aligned} \tag{2}$$

where $Y = Y_1 \times Y_2$, *Sync* denotes data transfer and synchronization between CPU and GPU within computation. As a result, three dimensional computations, i.e. Y_1, Y_2, X in order, need to be executed. Also note that a twiddle factor computation $t_{Y_2}^{Y_1}$ is introduced by the Cooley-Tukey decomposition between Y_1 and Y_2 step. Figure 1 shows the high-level working flow of our hybrid 2D FFT framework.

On GPU, a portion of Y_1 dimensional FFTs exceeding GPU global memory need to be computed. In this case, we divide the 2D FFT of GPU part into several passes such that the sub-problem of each pass can fit into GPU memory and be executed with the CPU portions concurrently. The number of passes equals to $\frac{\text{Xgpu*Y*\# of bytes per element}}{\text{GPU memory in bytes}}$. Each pass of GPU computation takes advantage of multiple streams to overlap computation and communication. The optimal number of streams derived from our empirical tuning is 8 on GTX480 and Tesla C2070/C2075.

2) GPU Computation Optimizations: On GPU, all the Y_1 dimensional 1D FFTs are calculated by codelets, i.e., highly-efficient straight-line code segments that solve small FFT problems. The codelet provides automatic matrix transposing within FFT substeps such that much transposition time can be saved. The concept of codelet was first introduced in FFTW, though its codelet generator only generates CPU code. We extend the FFTW codelet generator to generate GPU-based codelets. In addition, if size Y_1 is still large, we further decompose $Y_1 = Y_{11} \times Y_{12}$ sized 1D FFT into two dimensional FFTs with smaller sizes Y_{11} and Y_{12}, respectively. On GPU, device memory has much higher latency and lower bandwidth than on-chip memory. Therefore, shared memory is utilized for the decomposed FFTs to increase device memory bandwidth. For example, NVIDIA GTX480 GPU has 48KB shared memory which can store 6K complex single-precision data or 3K complex double-precision data in maximum. $Y_1W \times Y_{11} \times Y_{12}$ sized shared memory needs to be allocated, where Y_1W is chosen to 16 for half-warp of threads in GTX480 to enable coalesced access to device memory. The number of threads in each block, for both Y_{11} and Y_{12}-step sub-FFTs, is therefore $Y_1W \times max(Y_{11}, Y_{12})$ to realize maximum data parallelism on GPU. To calculate each Y_1-step 1D FFT, a size Y_{11} codelet is first executed to load data from global memory into shared memory for each block. Next, all threads in each block are synchronized to finish their work before data in shared memory is reused by the Y_{12}-step codelet and subsequently written back to global memory. Such shared memory technique effectively hides global memory latency and increases data reuse, both contributing to the GPU performance.

3) Asynchronous Strided Data Transfer: Another performance hurdle is strided memory transfers between CPU and GPU. Since we separate the load between CPU and GPU, the portion of input data required to be continuous in GPU memory is not contiguous in host memory. For each pass of our 2D FFT, if we use simple CUDA memory copy operations to transfer the total $\frac{Xgpu \times Y_1 \times Y_2}{\text{\# of passes} \times \text{\# of streams}}$ sized data into GPU, we need to utilize PCI bus $Y_1 \times \frac{Y_2}{\text{\# of passes} \times \text{\# of streams}}$ times because we transfer $Xgpu$ sized data each time. Clearly, the high PCI transfer overhead will kill all potential performance gain. CudaMemcpy2DAsync() can make only one call to transfer a strided 2D memory area of size $\frac{Xgpu \times Y_2}{\text{\# of passes} \times \text{\# of streams}}$ into GPU at a time. Therefore, the total number of PCI transfers is reduced to only Y_1. Moreover, such data transfer optimization supports for CUDA asynchronous concurrent execution. Different data transfers managed by different streams are executed concurrently and are overlapped with different streamed GPU kernel executions.

To best use cudaMemcpy2DAsync() in hybrid FFTs, when copying the GPU output back to CPU, it is used to copy multiple 2D strided arrays. Each streamed PCI transfer at this time could copy $\frac{Xgpu \times Y_2 \times Y_1}{\text{\# of passes} \times \text{\# of streams}}$ sized data. After the Y_1-step FFTs, all the streams on GPU are synchronized, and a subsequent barrier is set to synchronize GPU with CPU.

Our asynchronous strided transfer scheme achieves higher bandwidth than that of PCI transmission approach proposed in Gu's out-of-card FFT work [7] since we do not need to waste additional CPU resource to prepare buffer and therefore we get rid of overhead caused by buffering method in Gu's paper.

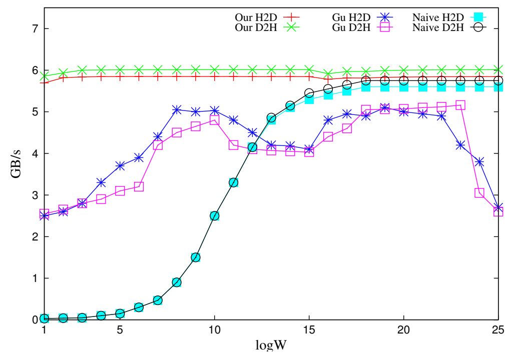

Fig. 2. PCIe bandwidth of data transfer schemes on GTX480.

To demonstrate the improvement of our PCI bandwidth, we used the same subarray test [7] as that in Gu's work, where there are C *regular subarrays* of length W each. Two regular subarrays are separated by a stride $X-W$ in a *large array* of size $C \times X$. Note that the large array is contiguous in system memory but regular subarrays are not contiguous. Figure 2 shows the improvement of our PCI bandwidth over Gu's work on GTX480 GPU. Overall, our 2D hybrid implementation can achieve 6 GB/s PCI bandwidth on average comparing to only 4.2 GB/s of Gu's work and 3.4 GB/s of naive PCI transfer.

4) CPU Computation Optimizations: For Y_1 dimensional computation on CPU, Y_1 sized 1D FFTs are required to calculate for $Xcpu \times Y_2$ times. For each Y_1 dimensional 1D FFT, data accesses have a stride of $Xcpu \times Y_2$. In addition, each 1D FFT needs to do a strided transpose. Both strided memory accesses and strided transpose are very expensive on CPU. Instead, we group the transformation of multiple complex arrays into a concurrent group operation and allow it to operate on non-contiguous (strided) data. Therefore, we need no input or output transposition and save much execution time. We set the number of arrays—$Xcpu$—to be the maximum of what a FFTW group plan could execute at a time. For each grouped array, the plan computes size Y_1 1D FFT across a stride of $Y_2 \times X$ for input and X for output. We need to execute such kind of plan for totally Y_2 times.

5) Co-Optimization of 2D FFT for CPUs and GPU: The overall coordination between CPU and GPU works like following. The workload of CPU, conceptually a loop of size Y_2, is parallelized into 4 concurrent subsections using 4 threads. Each thread is responsible to execute a distinct subsection in which independent grouped FFT computations are carried out. Simultaneously, the workload of GPU, including data transfers and kernel executions, is parallelized with CPU. Afterwards, jobs on GPU driven by different streams are synchronized before the task synchronization completes between GPU and CPUs. There is no matrix transposition on either GPU or CPU since computations in either side is re-organized to naturally subsume the strided transposition.

The subsequent calculation of twiddle factor multiplication $t_{Y_2}^{Y_1}$ and Y_2 & X dimensional FFTs is left for GPU. For $Y_2 = Y_{21} \times Y_{22}$ dimensional FFTs, Cooley-Tukey decomposition is again applied since relative large size of Y_2 would hurt the performance of codelet based GPU computing. Similarly, shared memory is taken into account for reusing data between

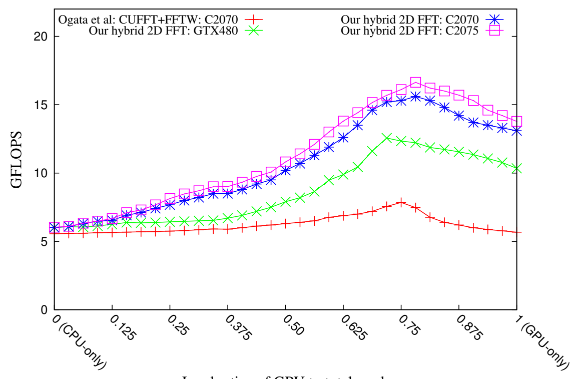

Fig. 3. Double-precision 2D hybrid FFT performance comparison.

the decomposed Y_{21}-step FFTs and the subsequent Y_{22}-step FFTs. For the last X-step, i.e., 1D contiguous FFT sub-problems, CUFFT library is used because it provides good performance for row-major contiguous 1D FFTs. Instead of using ordinary CUFFT plan, we make use of stream-enabled CUFFT plan so that all Y_2 and X dimensional computations plus both PCI transfers of Y_2's input and X's output become stream-based asynchronous executions.

6) Comparison to heterogeneous out-of-card CUFFT/FFTW implementation: To demonstrate the effectiveness of our approach on parallelizing FFT into heterogeneous processors, we compare our hybrid FFT library against a naive out-of-card hybrid 2D FFT implementation which simply uses CUFFT for GPU computation and FFTW for the CPU part. This heterogeneous distribution was first proposed in Ogata's [9] paper. Computation is firstly distributed along X dimension in the 1st-round of 2D FFT and along Y dimension for the 2nd-round. On GPU, sub-problems are divided into several passes to facilitate data transfer between GPU and CPU. Matrix transpose, CUFFT and data transfer are processed in asynchronous manner. On CPU, FFTW advanced interface is utilized to process strided data. The purpose of this comparison is to see how our optimization technique improves over a naive hybrid CUFFT/FFTW solution. In the experiment, we vary the CPU/GPU work ratio from 0% to 100% for the naive solution and show its double precision performance curve of size $2^{15} \times 2^{13}$ on C2070 in Figure 3. The peak performance of Ogata's naive hybrid FFT [9] gains only 7.7 GFLOPS which is far below that of our hybrid version. The main reason is the lacking of co-optimizations in the naive solution.

B. Hybrid 3D FFT Framework

General 3D FFT requires three rounds of computation. Each round computes 1D FFT along one dimension across the other two dimensions. Suppose the 3D input has sizes (Z,Y,X), the 3D FFT can be represented in tensor form as
$u_{3d} = \{d(Z,XY,XY,I,O), d(Y,X,X,O,O), d(X,1,1,O,O)\}$.

To describe how our hybrid 3D FFT works, we start with a simple hypothetical scenario where all the work is assigned to GPU, and then continue to reveal how computation is extracted from this GPU-only hypothetical case and is assigned to CPU. Suppose that $Z = Z_1 \times Z_2$ and $Y = Y_1 \times Y_2$, the u_{3d} can be

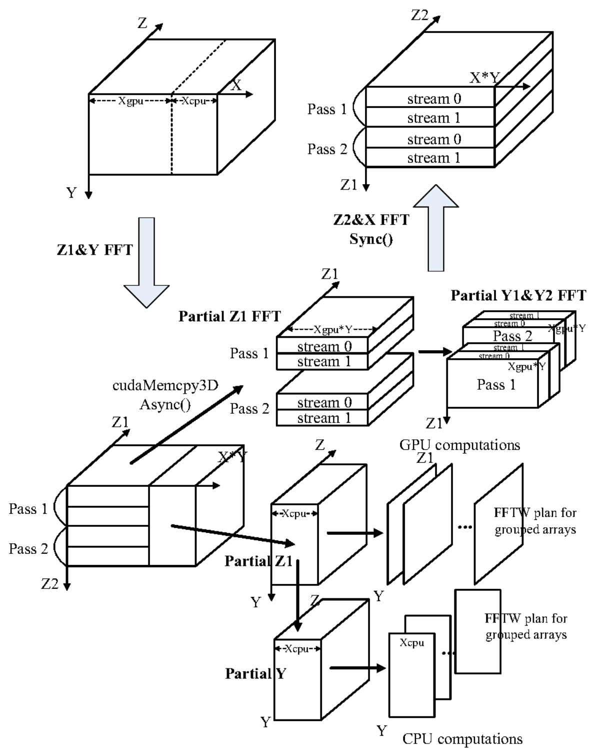

Fig. 4. Overview of hybrid large out-of-card 3D FFT framework.

transformed as formula (3), where $|_i$ and $|_o$ denotes respective input and output data transfers through PCI bus.

$$\{|_i, d(Z_1, XYZ_2, XY), |_o|_i, t_{Z_2}^{Z_1} d(Z_2, XYZ_1, XYZ_1), |_o|_i, \\ d(Y_1, Y_2X, X), |_o|_i, t_{Y_2}^{Y_1} d(Y_2, Y_1X, Y_1X), |_o|_i, d(X,1,1), |_o\} \tag{3}$$

The problem with this initial formula is that the workload in the formula cannot be well balanced between two computing devices. To save the number of PCI transfers and to balance computations between CPU and GPU, the computations sub-steps need to be reordered [7]. The reordered computations are summarized in equation (4).

$$\{|_i, d(Z_1, XYZ_2, XY), d(Y_1, Y_2X, X), t_{Y_2}^{Y_1} d(Y_2, Y_1X, \\ Y_1X), |_o|_i, t_{Z_2}^{Z_1} d(Z_2, XYZ_1, XYZ_1), d(X,1,1), |_o\} \tag{4}$$

1) Load Distribution: We have shown in hybrid 2D FFT framework that if we want to achieve actual high performance from heterogeneous implementation, we need to minimize the uses of PCI bus transfers. As Formula (4) indicates, in addition to the initial input transfer and final output transfer through PCI bus, at the very minimum, only two extra PCI transfers are called for, including copying output of Y_2 dimensional FFTs from GPU back to CPU, and copying the input of subsequent Z_2 dimensional FFTs from CPU into GPU. Therefore if we want to employ CPU for computing along with GPU, we need to arrange the data exchange between CPU and GPU to occur between Y_2 and Z_2 dimensional FFTs to reduce the total number of PCI bus transfers. Otherwise, more PCI transfers have to be invoked to merge the partial results of CPU and GPU between the two sub-steps FFTs. Hence, CPU is used to compute Z_1, Y_1 and Y_2 dimensional FFTs, and the subsequent calculation of Z_2 and X dimensional FFTs would be left for GPU. In summary, the heterogeneous 3D FFT tensor u_{gpu} on GPU and the tensor u_{cpu} on CPU are represented as formula (5), where *Sync* represents data transfer and synchronization between CPU and GPU within computation.

$$\begin{aligned} u_{gpu} &= \{|_i, d(Z_1, XgpuYZ_2, XgpuY), \\ & d(Y_1, Y_2Xgpu, Xgpu), t_{Y_2}^{Y_1} d(Y_2, Y_1Xgpu, Y_1Xgpu), \\ & Sync, t_{Z_2}^{Z_1} d(Z_2, XYZ_1, XYZ_1), d(X,1,1), |_o\} \\ u_{cpu} &= \\ & \{d(Z_1, XcpuYZ_2, XcpuY), d(Y, Xcpu, Xcpu), Sync\} \end{aligned} \tag{5}$$

For the first round computation composed of Z_1, Y_1 and Y_2 dimensional 1D FFTs, the work load is distributed to GPU and CPU along X dimension. Work ratio of GPU to CPU is $R_X = \frac{Xgpu}{Xcpu}$. The hybrid 3D FFT framework is illustrated in Figure 4.

2) Optimizations for GPU Computations and Data Transfers: Since the size of FFTs assigned to GPU is larger than size of device memory, total Z_1 dimensional FFTs on GPU are split into several passes. Number of passes equals to $\frac{\text{Xgpu*Y*Z*\# of bytes per element}}{\text{GPU memory in bytes}}$. $Z_1 = Z_{11} \times Z_{12}$ sized 1D FFTs are decomposed into two dimensional FFTs with size Z_{11} and Z_{12}. To achieve high bandwidth of global memory, shared memory is utilized with size $Z_1W \times Z_{11} \times Z_{12}$, where Z_1W enables coalesced access. The computation for decomposed FFTs within shared memory is the same as the Y_1 decomposed FFTs in our 2D hybrid FFT framework, and therefore is not further discussed here.

We use cudaMemcpy3DAsync() to transfer a strided 3D memory area of size $\frac{Xgpu \times Y \times Z_2}{\text{\# of passes} \times \text{\# of streams}}$ into GPU at a time. Total number of PCI transfer is Z_1. For copying output back from GPU to CPU, we still need the 3D strided memory copy. Each streamed PCI transfer at this time could copy $\frac{Xgpu \times Y \times Z_2 \times Z_1}{\text{\# of passes} \times \text{\# of streams}}$ sized data. All Z_1, Y_1 and Y_2 dimensional computations plus Z_1's input and Y_2's output transfers are asynchronous executions, and only required synchronization is set after Y_2-step. Similar as 2D transfer scheme, our 3D method stills outperforms Gu's out-of-card FFT work. PCI bandwidth for our 3D case on GTX480 can achieve 5.9 GB/s on average comparing to only 3.9 GB/s of Gu's work.

3) Co-Optimization for CPUs and GPU: Similar to 2D hybrid FFT, we execute the size Z_1 FFTs in groups on CPU. For each grouped array, FFTW plan computes size Z_1 1D FFTs across a stride of $X \times Y \times Z_2$ for input and $Xcpu * Y$ for output. The total number of executions of such plans is $Xcpu \times Z_2$. For the following Y_1 and Y_2 dimensional FFTs on CPU, we only need to calculate size Y FFTs instead. The number of grouped array is $Xcpu$. The plan computes size Y 1D FFTs across a stride of $Xcpu$ for input and X for output. Total number of such plans is Z. Moreover, all the grouped FFT tasks on CPU are distributed to 4 concurrent threads. There is only one invocation to cudaDeviceSynchronize() to synchronize all the streams on GPU. A subsequent barrier is set to synchronize the work of GPU with CPUs.

IV. LOAD BALANCING BETWEEN GPU AND CPU

The 2D and 3D hybrid FFT frameworks layout the basic schemes of workload distribution between CPU and GPU.

However, there are parameters whose values need to be tuned for the optimal load balancing for different CPU/GPU combinations. In this work, we combine both performance modeling and empirical searching to finish the last mile towards the optimal load balancing. The empirical tuning is done at build time.

Our approach is to split the total execution in either GPU or CPU into several primitive sub-steps, analyze the heterogeneous execution flow, and derive a performance model for each primitives. The model provides estimated execution time that is parameterized with the load ratio of GPU to total work. For each hardware configuration, we calibrate the models with two profiling runs, one on CPU and one on GPU, to determine the values of model parameters in different distribution ratios. Afterwards, using those parameters, we can automatically estimate, rather than really measuring, the total execution time of our implementation under varying ratios. We further use dynamic-programming to find the optimal implementation for different problems using the primitives as building blocks. The estimated performance might not be completely precise. Therefore, we only use it to provide a small region of potentially good choices. Within the region we exhaustively measure the performance and choose the best one. Overall, we avoid a walk-through of the vast space of all possible combinations of primitives.

A. Load Balancing of 2D FFT

Using the hybrid 2D FFT as an example, suppose that the total problem size is $Y_1 \times Y_2 \times X$. The load ratio of GPU to total work is defined as $R_g = \frac{Xgpu}{X}$ along X dimension, therefore the ratio of CPU to the total is $1 - R_g$. The execution time of the whole process can be modeled as 8 parameters, which are summarized in Table I. We used two runs, one on GPU and CPU each, to determine $T_{2d\text{H2D-gpu}}$, $T_{Y_1\text{kernel-gpu}}$, and $T_{2d\text{D2H-gpu}}$ as execution time of corresponding Table I's parameters in GPU-only case, and to determine $T_{Y_1\text{fftw-cpu}}$ as execution time of $T_{Y_1\text{fftw}}(1-R_g)$ in CPU-only case. Therefore, each parameter value in Table I can be modeled with different distribution ratios.

TABLE I. PARAMETERS FOR 2D FFT RUNNING TIME ESTIMATION

Parameters	Description
# passes	Total # of passes. Subproblem of each pass fits into GPU memory.
# streams	Total # of streams that support for asynchronous kernel executions and transfers.
# thds	# of threads of CPU.
$T_{2d\text{H2D}}(i,R_g)$	$= T_{2d\text{H2D-gpu}} \times R_g$. Time of copying a 2D strided array of size $\frac{R_g \times X \times Y_2}{\text{\# passes} \times \text{\# streams}}$ from host to device in stream i.
$T_{Y_1\text{kernel}}(i,R_g)$	$= T_{Y_1\text{kernel-gpu}} \times R_g$. Time of Y_1-step FFTs computation of concurrent kernel in stream i. Thread block size is $Y_1W \times max(Y_{11}, Y_{12})$, grid size is $\frac{R_g \times X \times Y_2}{\text{\# passes} \times \text{\# streams}}$.
$T_{2d\text{D2H}}(i,R_g)$	$= T_{2d\text{D2H-gpu}} \times R_g$. Time of copying a 2D strided array of size $\frac{R_g \times X \times Y}{\text{\# passes} \times \text{\# streams}}$ from device to host in stream i.
$T_{Y_1\text{fftw}}(1-R_g)$	$= T_{Y_1\text{fftw-cpu}} \times (1-R_g)$. Time of Y_1-step FFTs on advanced FFTW plan for grouped array of size $(1-R_g) \times X$ in CPU. Total number of plans is Y_2.
$T_{Y_2\&X}$	Time of subsequent calculation of Y_2 and X dimensional FFTs.

On GPU side, for hybrid Y_1 dimensional FFTs, the execution time is estimated as TG_{2D} shown in equation (6).

$$\begin{aligned} TG_{2D} = \quad & \#passes \times max\{[Y_1 \times T_{2d\text{H2D}}(0,R_g) + \\ & T_{Y_1\text{kernel}}(0,R_g) + T_{2d\text{D2H}}(0,R_g)]; [...]; \\ & [Y_1 \times T_{2d\text{H2D}}(\text{\# streams-1}, R_g) \\ & + T_{Y_1\text{kernel}}(\text{\# streams-1}, R_g) \\ & + T_{2d\text{D2H}}(\text{\# streams-1}, R_g)]; \quad \} \end{aligned} \tag{6}$$

On CPU side, for hybrid Y_1 dimensional FFTs, the execution time is estimated as $TC_{2D} = \frac{Y_2}{\#thds} \times T_{Y_1\text{fftw}}(1-R_g)$.

Since synchronization is set after Y_1-step FFT on both GPU and CPU side to guarantee the correctness of results, the execution time of hybrid Y_1 dimensional FFT can be modeled as the maximum of the GPU time and CPU time, i.e., $T_{Y_1} = max\{TG_{2D}, TC_{2D}\}$. And the total time estimation will be consequently calculated as $T_{\text{total}} = max\{TG_{2D}, TC_{2D}\} + T_{Y_2\&X}$. Afterwards, empirical searching is employed to find the parameter values that can make TG_{2D} equal to TC_{2D}, as well as the sub-steps along other dimensions, which indicates the optimal load balancing.

B. Load Balancing of 3D FFT

The load balancing in the hybrid 3D FFT framework is similar to that of the 2D cases. Suppose that the total problem size is $Z_1 \times Z_2 \times Y_1 \times Y_2 \times X$. The load ratio of GPU to total work is denoted as $R_g = \frac{Xgpu}{X}$ along X dimension and ratio of CPU to total problem is $1 - R_g$. Performance parameters for the sub-steps in 3D hybrid FFT are summarized in Table II. Two profiling runs still help determine $T_{3d\text{H2D-gpu}}$, $T_{Z_1\text{kernel-gpu}}$, $T_{Y_1\text{kernel-gpu}}$, $T_{Y_2\text{kernel-gpu}}$, $T_{3d\text{D2H-gpu}}$, and $T_{Z_1\text{fftw-cpu}}$, $T_{Y\text{fftw-cpu}}$ as execution time in respective GPU-only and CPU-only case for the parameters in Table II.

TABLE II. PARAMETERS FOR 3D FFT RUNNING TIME ESTIMATION

Parameters	Description
$T_{3d\text{H2D}}(i,R_g)$	$= T_{3d\text{H2D-gpu}} \times R_g$. Time of copying a 3D strided array of size $\frac{R_g \times Y \times Z_2}{\text{\# passes} \times \text{\# streams}}$ from host to device in stream i.
$T_{Z_1\text{kernel}}(i,R_g)$	$= T_{Z_1\text{kernel-gpu}} \times R_g$. Time of Z_1-step FFTs computation of concurrent kernel in stream i. Thread block size is $Z_1W \times max(Z_{11}, Z_{12})$, grid size is $\frac{R_g \times Y \times Z_2}{\text{\# passes} \times \text{\# streams}}$.
$T_{Y_1\text{kernel}}(i,R_g)$	$= T_{Y_1\text{kernel-gpu}} \times R_g$. Time of Y_1-step FFTs computation of concurrent kernel in stream i. Thread block size is Y_1W, grid size is $\frac{R_g \times Y_2}{Y_1W} \times \frac{Z}{\text{\# passes} \times \text{\# streams}}$
$T_{Y_2\text{kernel}}(i,R_g)$	$= T_{Y_2\text{kernel-gpu}} \times R_g$. Time of Y_2-step FFTs computation of concurrent kernel in stream i. Thread block size is Y_2W, grid size is $\frac{R_g \times Y_1}{Y_2W} \times \frac{Z}{\text{\# passes} \times \text{\# streams}}$,
$T_{3d\text{D2H}}(i,R_g)$	$= T_{3d\text{D2H-gpu}} \times R_g$. Time of copying a 3D contiguous array of size $\frac{R_g \times Y \times Z_1 \times Z_2}{\text{\# passes} \times \text{\# streams}}$ from device to host in stream i.
$T_{Z_1\text{fftw}}$	$= T_{Z_1\text{fftw-cpu}} \times (1-R_g)$. Time of Z_1-step FFTs on advanced FFTW plan for grouped array of size Y in CPU. Total # of plans is $(1-R_g) \times X \times Z_2$.
$T_{Y\text{fftw}}(1-R_g)$	$= T_{Y\text{fftw-cpu}} \times (1-R_g)$. Time of Y-step FFTs on advanced FFTW plan for grouped array of size $(1-R_g) \times X$ in CPU.Total # of plans is Z.
$T_{Z_2\&X}$	Time of subsequent calculation of Z_2 and X dimensional FFTs.

TABLE III. CONFIGURATIONS OF GPU, CPU, FFTW AND MKL

GPU	Global Memory	NVCC
GeForce GTX480	1.5GB	3.2
Tesla C2070	6GB	3.2
Tesla C2075	6GB	3.2
CPU	**Frequency, # of Cores**	**FFTW & MKL**
Intel i7 920	2.66GHz, 4 cores	3.3.3 & 10.3

On GPU side, for hybrid $Z_1\&Y$ dimensional FFTs, the execution time is estimated as TG_{3D} shown in equation (7).

$$\begin{aligned} TG_{3D} = \; & \#passes \times max\{[Z_1 \times T_{3d\mathrm{H2D}}(0,R_g)+ \\ & T_{Z_1\mathrm{kernel}}(0,R_g) + T_{Y_1\mathrm{kernel}}(0,R_g)+ \\ & T_{Y_2\mathrm{kernel}}(0,R_g) + T_{3d\mathrm{D2H}}(0,R_g)]; \quad [......]; \\ & [Z_1 \times T_{3d\mathrm{H2D}}(\text{\# streams-1}, R_g) \\ & + T_{Z_1\mathrm{kernel}}(\text{\# streams-1}, R_g) \\ & + T_{Y_1\mathrm{kernel}}(\text{\# streams-1}, R_g) \\ & + T_{Y_2\mathrm{kernel}}(\text{\# streams-1}, R_g) \\ & + T_{3d\mathrm{D2H}}(\text{\# streams-1}, R_g)];\} \end{aligned} \tag{7}$$

On CPU side, for hybrid $Z_1\&Y$ dimensional FFTs, the execution time is estimated as TC_{3D} represented in equation (8).

$$\begin{aligned} TC_{3D} = \; & \frac{(1-R_g) \times X \times Z_2}{\#thds} \times T_{Z_1\mathrm{fftw}} \\ & + \frac{Z}{\#thds} \times T_{Y\mathrm{fftw}}(1-R_g) \end{aligned} \tag{8}$$

Similarly, since synchronization is set after $Z_1\&Y$-step FFT on both GPU and CPU side, the execution time of the hybrid $Z_1\&Y$ dimensional FFT can be modeled as the maximum of the GPU time and CPU time, i.e., $T_{Z_1\&Y} = max\{TG_{3D}, TC_{3D}\}$. The total time estimation is calculated as $T_{\mathrm{total}} = max\{TG_{3D}, TC_{3D}\} + T_{Z_2\&X}$. Empirical searching techniques similar to 2D cases are used to balance the substeps, as well as those along other dimensions.

V. PERFORMANCE EVALUATION

In this section, we evaluate the hybrid 2D and 3D FFT implementation on three heterogeneous computer configurations. A single model of CPU, Intel i7 920, is coupled with three different NVIDIA GPUs, i.e. GeForce GTX480, Tesla C2070 and Tesla C2075 in the three experiments. Configurations of the FFT libraries, the GPUs and the CPU with total 24GB host memory are summarized in Table III, where NVCC is the compiler driver for NVIDIA CUDA GPUs.

As mentioned in section III.A.3), the performance reported here includes both computational time and data transferring time between host and device. Our library in both single- and double-precisions are compared to FFTW and Intel MKL, two of the best performing FFT implementations on CPU. Moreover, our hybrid FFT library is compared with Gu's out-of-card FFT work [7], a highly efficient GPU-based FFT library and the only one that we know can handle the problems sizes larger than GPU memory. The whole design of this performance evaluation is to let us see how much performance improvement can be achieved by using both CPU and GPU in computation, against the best-performing GPU-only or CPU-only FFT implementations. Please note that we can't compare our library with pure CUFFT implementation because it requires problem size to be smaller than GPU memory, and therefore won't accept problem sizes used in this evaluation. In FFTW, Streaming Single Instruction Multiple Data Extensions (SSE) on Intel CPU is enabled for better performance. Also FFTW results are got with the 'MEASURE' flag, the second most extensive performance tuning mode. The 'EXHAUSTIVE' flag in FFTW, which represents the most extensive searching and tuning, is not used because the problem sizes in this evaluation are so large that FFTW can't finish its search under the 'EXHAUSTIVE' mode. For example, we tried running FFTW in 'EXHAUSTIVE' mode for a 2^{28} FFT problem, but found FFTW couldn't finish the search in 3 days. In addition, Intel MKL automatically enables SSE at run time. Both FFTW and MKL are chosen to run with four threads. Even though the i7 CPU supports 8 hyperthreads, the 8-thread FFTW and MKL didn't show performance advantage over, actually in some cases were slower than, the 4-thread versions.

All FFT problem sizes are larger than GPU memory. For double-precision implementation on GTX480, we choose the test cases from 32M points (i.e. 2^{25}) to 256M points (i.e. 2^{28}). 32M-point FFT is twice the maximal problem size that GTX480 memory can accommodate and 256M-point FFT is the maximum problem size that can fit into host memory. For single precision tests on GTX480, the sizes are from 64M points (i.e. 2^{26}) to 512M points (i.e. 2^{29}). Similarly, for Tesla C2070/C2075, test cases are from 256M points (i.e. 2^{28}) to 512M points (i.e. 2^{29}) for double precision implementation and from 512M points (i.e. 2^{29}) to 1024M points (i.e. 2^{30}) for single precision test. The performance of a D dimensional complex FFT is evaluated in GFLOPS [1] defined as $GFlops = \frac{5M\sum_{d=1}^{D} log_2 N_d}{t} \times 10^{-09}$ where the total problem size is $M = N_1 \cdot N_2 \cdot ... \cdot N_D$ and t is execution time in seconds.

A. Performance Tuning

For both 2D and 3D FFTs, our performance modeling and empirical searching find the optimal ratio and best performance for different input sizes. Figure 5 demonstrates the effect and accuracy of load distribution ratio tuning on overall FFT performance. The figure shows the actual and modeled double-precision 2D FFT performance on three different GPUs under different load ratios with problem size $2^{15} \times 2^{13}$. The x-axis is the distribution ratio from 0% to 100%. In particular, 0% represents running our hybrid FFT library only on CPU and 100% represents running only on GPU. The two extreme cases will help demonstrating the intrinsic overhead incurred by splitting computation/communication into two devices. Table IV shows the values of model parameters from the profiling runs of GPU-only and CPU-only case described in section IV.A and IV.B, where # passes, # streams and # thds are 2, 8, 4, respectively. The modeled optimal and average performance is 99%, 96%, 95%, and 98%, 93%, 91% of the actual measured on GTX480, Tesla C2070 and C2075, respectively. Our performance modeling, even without empirical tuning, is accurate.

Figure 6 shows the actual and modeled double-precision 3D FFT performance with size $2^{10} \times 2^9 \times 2^9$. The modeled optimal and average performance is again very accurate, and is 99%, 95%, 93%, and 98%, 94%, 91% of the actual values.

In addition, the final performance of our library is also compared against the cases that run our library only on GPU or

TABLE IV. MODEL PARAMETERS FOR 2D AND 3D FFTS ON C2070

Parameter	Value	Parameter	Value	Parameter	Value
$T_{2d\mathrm{H2D\text{-}gpu}}$	0.003	$T_{Y_1\,\mathrm{kernel\text{-}gpu}}$	0.041	$T_{2d\mathrm{D2H\text{-}gpu}}$	0.042
$T_{Y_1\,\mathrm{fftw\text{-}cpu}}$	0.195	$T_{Y_2\&X}$	1.137		
$T_{3d\mathrm{H2D\text{-}gpu}}$	0.024	$T_{Z_1\,\mathrm{kernel\text{-}gpu}}$	0.014	$T_{Y_1\,\mathrm{kernel\text{-}gpu}}$	0.028
$T_{Y_2\,\mathrm{kernel\text{-}gpu}}$	0.1	$T_{3d\mathrm{D2H\text{-}gpu}}$	0.02	$T_{Z_1\,\mathrm{fftw\text{-}cpu}}$	0.0008
$T_{Y\,\mathrm{fftw\text{-}cpu}}$	0.01	$T_{Z_2\&X}$	1.221		

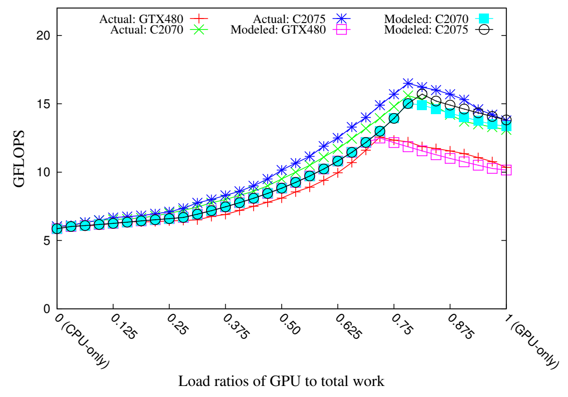

Fig. 5. Double-precision hybrid 2D FFT performance tuning.

only on CPU. As shown in Figure 5 for 2D FFT, on GTX480, Tesla C2070 and C2075, the optimal ratio of GPU to total work is 71.9%, 78.2% and 78.2%. The library's best performance is 21.4%, 19.1% and 20.7% faster than GPU-only performance, and is 1.09×, 1.59× and 1.76× faster than CPU-only cases. Moreover, for 3D FFT in Figure 6, the optimal ratio of GPU to the total is 75.0%, 78.2% and 78.2%. The load-balanced library performs 25.6%, 22.8% and 23.1% faster than GPU-only performance, and is 1.25×, 1.51× and 1.62× faster than CPU-only case.

Not shown in this figure, but the single-precision performance tuning has similar curve as that of the double-precision case. Also the optimal ratio of GPU to CPU in single-precision version is larger than that of double precision since GPU has relatively higher performance on single precision operations than CPU.

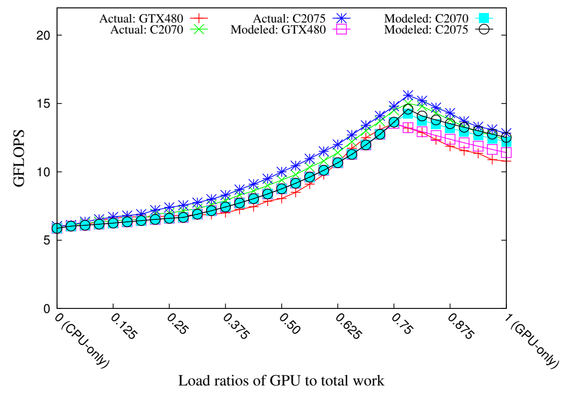

Fig. 6. Double-precision hybrid 3D FFT performance tuning.

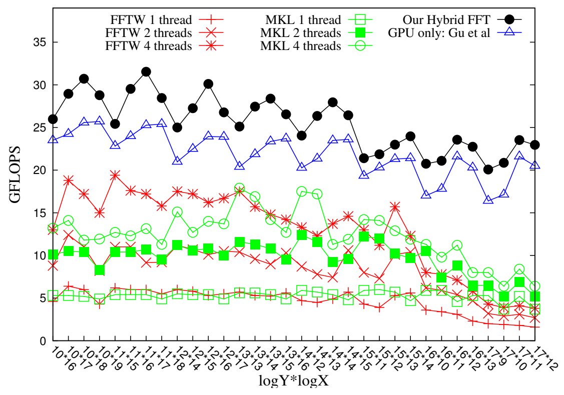

Fig. 7. Single-precision 2D FFT of size from 2^{26} to 2^{29} on GTX480.

B. Evaluation for 2D Hybrid FFT

We evaluate various 2D FFT problems on the three heterogeneous configurations. In all the figures, the test points are indexed in an increasing order of Y in the problem sizes. Figure 7 shows our single-precision 2D FFT performance on Geforce GTX480 with problem sizes from 2^{26} to 2^{29}. On average, our single-precision 2D hybrid FFT on GTX480 achieves 25.5 GFLOPS. Our optimally-distributed performance is 16% faster than Gu's pure GPU version, and is also 95% faster than the 4-thread FFTW and 1.06× faster than the 4-thread MKL. In particular, even if we run our hybrid FFT only on GPU, it is still faster than Gu's work, a high-performance GPU-based FFT implementation, mainly attributing to the asynchronous transfer schemes in our hybrid algorithm.

Furthermore, we also test 2D hybrid FFT performance in double-precision as shown in Figure 8. Our double-precision 2D hybrid FFT on GTX480 achieves 13.1 GFLOPS. Moreover, our optimal performance is 20% faster than Gu's pure GPU implementation, and is 98% faster than the 4-thread FFTW and 1.04× faster than the 4-thread MKL.

Additionally, Figure 9 and Figure 10 show our large 2D FFT results on the Tesla C2070/C2075 with even larger problem sizes in single and double precision. On average, our single-precision 2D hybrid FFT achieves 37.2 GFLOPS on Tesla C2075 and 33.7 GFLOPS on Tesla C2070, which represent speedups of 26% and 24% over Gu's pure GPU implementation, 2.23× and 1.93× over the 4-thread FFTW, and 2.41× and 2.09× over the 4-thread MKL, respectively.

For double precision, the performance is 19.1 GFLOPS and 17.8 GFLOPS on Tesla C2075 and C2070, which represent 29% and 28% speedups over Gu's pure GPU implementation, 2.08× and 1.87× speedups over the 4-thread FFTW and 2.24× and 2.02× speedups over the 4-thread MKL.

Particularly notable is that as Y increases, the performance of both FFTW and MKL decreases rapidly because the data locality loses rapidly along the Y dimensional computation when Y increases. On the contrary, our hybrid FFT demonstrates a much more stable performance.

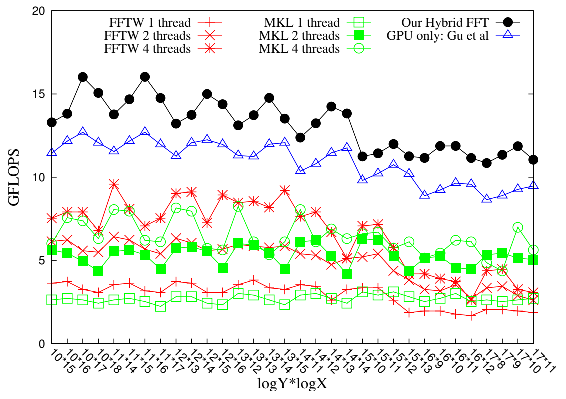

Fig. 8. Double-precision 2D FFT of size from 2^{25} to 2^{28} on GTX480.

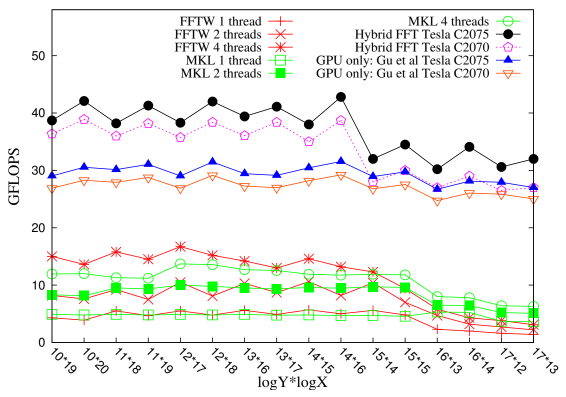

Fig. 9. Single-precision 2D FFT of size from 2^{29} to 2^{30} on Tesla.

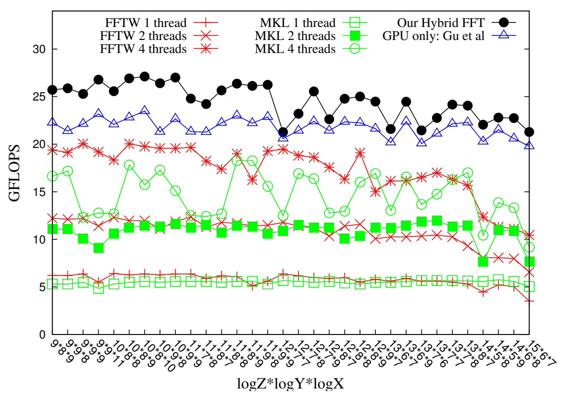

Fig. 11. Single-precision 3D FFT of size from 2^{26} to 2^{29} on GTX480.

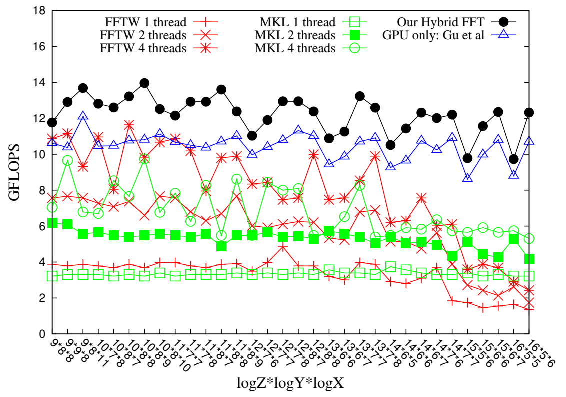

Fig. 12. Double-precision 3D FFT of size from 2^{25} to 2^{28} on GTX480.

C. Evaluation for 3D Hybrid FFT

Figure 11, 13 and Figure 12, 14 show the performance of our single- and double-precision 3D hybrid FFT on GTX480 and Tesla C2075/C2070. On average our library achieves 18.4 GFLOPS on GTX480, 23.2 GFLOPS on C2075 and 21.5 GFLOPS on C2070.

On average, our hybrid 3D FFT library is 19.5% faster than Gu's GPU only FFT implementation, 74.2% faster than the 4-thread FFTW and 1.09× faster than MKL. Similar to their 2D performance, FFTW's and MKL's 3D performance decrease quickly as Z increases due to the loss of data locality though MKL generally performs better than FFTW for large Zs. Our hybrid library generally maintains its good performance for the same large Z cases.

D. Accuracy of Our Hybrid FFT

The correctness of our hybrid FFT library is verified against FFTW and MKL. All three libraries are tested with

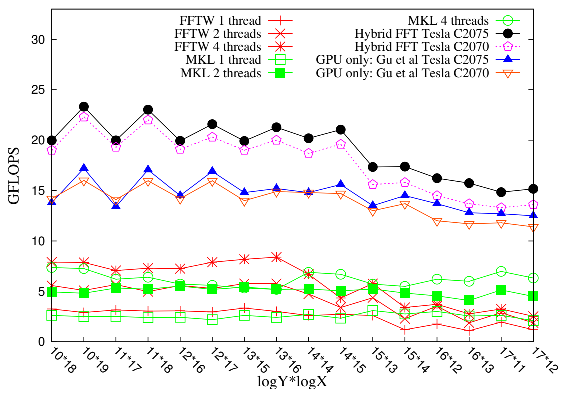

Fig. 10. Double-precision 2D FFT of size from 2^{28} to 2^{29} on Tesla.

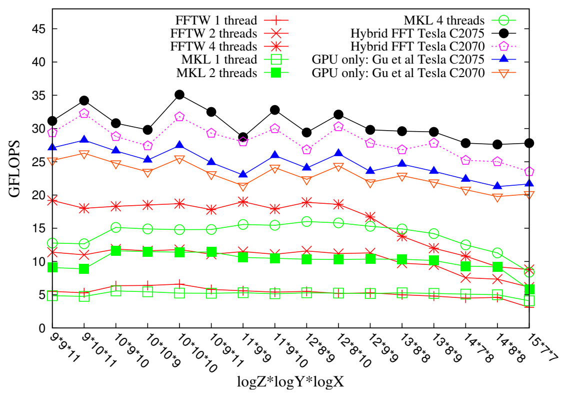

Fig. 13. Single-precision 3D FFT of size from 2^{29} to 2^{30} on Tesla.

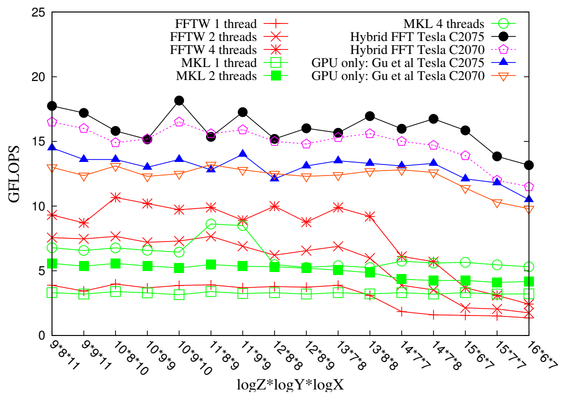

Fig. 14. Double-precision 3D FFT of size from 2^{28} to 2^{29} on Tesla.

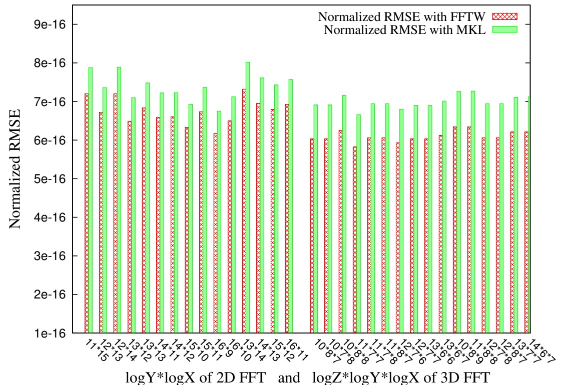

Fig. 16. Double-precision accuracy of 2D and 3D FFT.

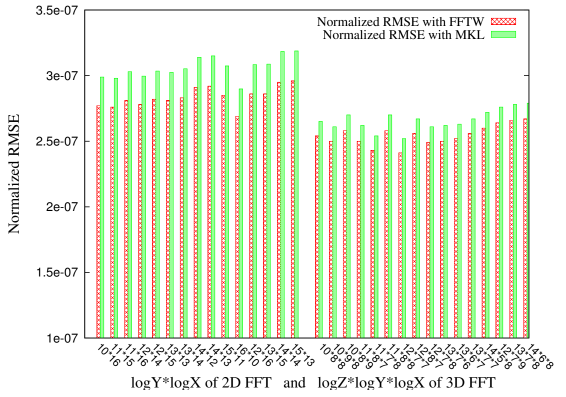

Fig. 15. Single-precision accuracy of 2D and 3D FFT.

the same input data randomly chosen from -0.5 to 0.5 and the difference in output is quantified as normalized RMSE over the whole data set. The normalized RMSE evaluates the relative degree of deviations and is a wildly used metric for numeric accuracy. The normalized RMSE is defined as $\sqrt{\frac{\sum_{i=0}^{N-1}[(X_i-R_i)^2+(Y_i-S_i)^2]}{2N}} / \sqrt{\frac{\sum_{i=0}^{N-1}(R_i^2+S_i^2)}{2N}}$.

The normalized RMSEs of single- and double-precision for both 2D and 3D FFTs are shown in Figure 15 and Figure 16. As we can see the normalized RMSE is extremely small and is in the range from 2.41×10^{-07} to 3.18×10^{-07} for single precision and 5.82×10^{-16} to 8.02×10^{-16} for double precision. In other words, our hybrid FFT library produces the same accurate results as FFTW and MKL.

VI. Conclusion

In this paper, we proposed a hybrid FFT library that concurrently uses both CPU and GPU to compute large FFT problems. The library has four key components: a decomposition paradigm that mixes two FFT algorithms to extract different types of computation and communication patterns for the two different processor types; an optimizer that exploits substantial parallelism for both GPU and CPUs; a load balancer that assigns workloads to both GPU and CPU, and determines the optimal load balancing by effective performance modeling; and a heuristic that empirically tunes the library to best tradeoff among communication, computation and their overlapping. Overall, our hybrid library outperforms several latest and widely used large-scale FFT implementations.

References

[1] L. Gu, X. Li, and J. Siegel, "An empirically tuned 2d and 3d fft library on cuda gpu," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. New York, NY, USA: ACM, 2010, pp. 305–314.

[2] (2012) NVIDIA CUFFT Library. [Online]. Available: http://developer.nvidia.com

[3] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka, "Bandwidth intensive 3-D FFT kernel for GPUs using CUDA," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, pp. 1–11.

[4] A. Nukada and S. Matsuoka, "Auto-tuning 3-D FFT library for CUDA GPUs," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, pp. 1–10.

[5] N. K. Govindaraju and S. e. Larsen, "A memory model for scientific algorithms on graphics processors," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2006, p. 89.

[6] N. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, "High performance discrete Fourier transforms on graphics processors," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008, pp. 1–12.

[7] L. Gu, J. Siegel, and X. Li, "Using gpus to compute large out-of-card ffts," in *Proceedings of the international conference on Supercomputing*, ser. ICS '11. New York, NY, USA: ACM, 2011, pp. 255–264.

[8] Y. Chen and X. e. Cui, "Large-scale FFT on GPU clusters," in *Proceedings of the 24th ACM International Conference on Supercomputing*. ACM, 2010, pp. 315–324.

[9] Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka, "An efficient, model-based cpu-gpu heterogeneous fft library," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, april 2008, pp. 1 –10.

[10] M. Frigo and S. Johnson, "The Fastest Fourier Transform in the West," 1997.

[11] M. Frigo, "A fast Fourier transform compiler," *ACM SIGPLAN Notices*, vol. 34, no. 5, pp. 169–180, 1999.

[12] M. Frigo and S. G. Johnson, "The design and implementation of fftw3," *Proceeding of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[13] (2012) Intel Math Kernel Library. [Online]. Available: http://software.intel.com/en-us/articles/intel-mkl-103-release-notes

[14] P. Duhamel and M. Vetterli, "Fast fourier transforms: a tutorial review and a state of the art," *Signal Process.*, vol. 19, no. 4, pp. 259–299, Apr. 1990.

[15] J. Cooley and J. Tukey, "An algorithm for the machine computation of complex Fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.