Towards Memory-Load Balanced Fast Fourier Transformations in Fine-grain Execution Models

Chen Chen, Yao Wu, Stéphane Zuckerman, and Guang R. Gao
Electrical and Computer Engineering Department
University of Delaware
Newark, DE 19716, USA
Email: {chenchen,yaowu,szuckerm,ggao}@capsl.udel.edu

***Abstract*—The codelet model is a fine-grain dataflow-inspired program execution model that balances the parallelism and overhead of the runtime system. It plays an important role in terms of performance, scalability, and energy efficiency in exascale studies such as the DARPA UHPC project and the DOE X-Stack project.**

As an important application, the Fast Fourier Transform (FFT) has been deeply studied in fine-grain models, including the codelet model. However, the existing work focuses on how fine-grain models achieve more balanced workload comparing to traditional coarse-grain models. In this paper, we make an important observation that the flexibility of execution order of tasks in fine-grain models improves utilization of memory bandwidth as well. We use the codelet model and the FFT application as a case study to show that a proper execution order of tasks (or codelets) can significantly reduce memory contention and thus improve performance. We propose an algorithm that provides a heuristic guidance of the execution order of the codelets to reduce memory contention.

We implemented our algorithm on the IBM Cyclops-64 architecture. Experimental results show that our algorithm improves up to 46% performance compared to a state-of-the-art coarse-grain implementation of the FFT application on Cyclops-64.

Keywords*—*FFT; fine grain; execution model; memory bandwidth.

I. Introduction

Chip multiprocessors (CMPs) have rekindled a strong interest for efficient parallelization of well-known algorithms. With CMP systems, the infamous "memory wall" [33], [41], [48] has been on the mind of high-performance programmers more than ever. An impressive literature has emerged since 2005. Papers describe how to better characterize shared-memory multiprocessing on symmetric multiprocessor (SMP) and/or CMP systems [2], [29], [31]; how to parallelize specifically for one or more CMP sharing the same memory (SMP/CMP), [9], [38]; etc. Moreover, the current architectural features (out-of-core execution, hardware prefetchers, branch predictors, etc.) contrast a lot with current so-called many-core processors which feature much simpler core architectures.

Indeed, numerous so-called general purpose "many-core chips" have been announced or are already available on the market [1], [6], [14], [17], [37], [40], [43]. All these architectures, while different on many levels, also feature one important characteristic: the amount of (local) memory per core is drastically lower than on current mainstream CMP systems.

The conventional wisdom for parallelizing algorithms for shared-memory SMP/CMP systems is to use coarse-grain synchronization in the form of barriers. This works reasonably well on a system with a low core count, but becomes cumbersome as the number of cores on a chip increases [24], [36]. This is in part due to the competition for shared resources that is becoming fiercer between cores as their number increases. Hence many-core chips require a different approach to high-performance computing, as not only is memory scarce on a per-core basis, but it also means that the conventional coarse-grain approach to synchronizing tasks is showing its limits.

A very important piece of software in signal processing is the Fast Fourier Transform (FFT). Many very clever algorithms and implementations have been proposed over the years (see Section VI). However, most of these parallelization schemes still rely on coarse-grain synchronization between threads. There are a few FFT studies based on fine-grain execution models. Those works focus on how fine-grain models achieve more balanced workload comparing to the coarse-grain approaches.

We intend to demonstrate that, besides achieving more balanced workload, fine-grain execution models also improve the utilization of memory bandwidth comparing to the coarse-grain execution models. A fine-grain execution model allows more freedom to reorder the execution of tasks. Since each task may have different workload on the memory bandwidth, the freedom of reordering enables the system to better balance the memory bandwidth usage required by the running application. To address our point, we use FFT as a case study on the IBM Cyclops-64 (C64) many-core architecture to show the advantage of the fine-grain execution model. This paper makes three main contributions:

1) The design of a fine-grain FFT algorithm with a heuristic guidance of the execution order of the codelets to reduce memory contention for general purpose many-core architectures. It is implemented on the IBM Cyclops-64 many-core architecture.
2) Using the codelet execution model [49] as a basis, we performed extensive experimentations to study the behavior of FFT depending on the granularity of synchronization. Three versions are compared: coarse-grain (using barriers), fine-grain (using point-to-point synchronization), and guided fine-grain (fine-

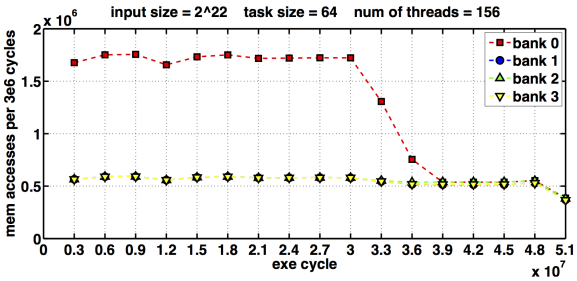

Fig. 1: Access rates of the 4 off-chip memory banks in the coarse-grain FFT algorithm. *Bank 0* is accessed three times more than the other banks, causing contention.

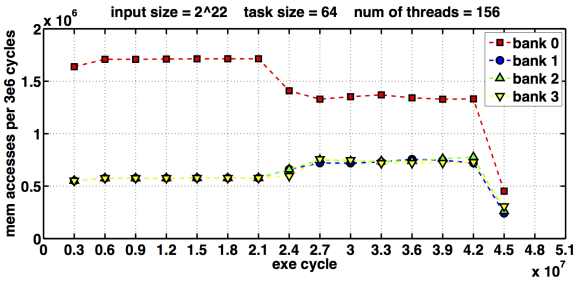

Fig. 2: Access rates of the 4 off-chip memory banks in the fine-grain FFT algorithm. Compared to Fig. 1, traffic is much more balanced across the memory banks as the computation reaches the second half of the program execution. This leads to lower contention.

grain with the heuristic guidance). We show that our algorithm improves up to 46% performance comparing to a state-of-the-art implementation on the same system.

3) We compare our fine-grain approach to an alternative solution that reduces memory contention via randomizing memory addresses. We show that the fine-grain approach outperforms the address randomization approach when the input data size is large enough. Moreover, the performance gap will enlarge along with the increment of the input data size.

This paper proposes the following structure: Section II gives a motivating example which illustrates the reasons behind the proposed research; Section III provides the necessary background to understand the proposed FFT study; Section IV describes the followed methodology; Section V describes the obtained experimental results; Section VI examines the related work for FFT parallelization; finally, Section VII gives our conclusions for this study.

II. Motivating Example

The coarse-grain computation of FFT on C64 is a multi-stage algorithm. In each stage, the cores execute many parallel tasks to apply butterfly computations on the intermediate data. At the end of each stage, all the cores wait for a barrier. Each task loads some data and *twiddle factors*[1], computes, and stores computed data in place. This algorithm has shown good on-chip performance in Chen *et al.*'s work [11].

However, this technique does not work well in the case that the data and twiddle factors are stored in the off-chip memory. We found that the problem is caused by the unbalanced memory accesses to the off-chip memory banks.

The interconnection network on a C64 chip has four ports connecting four off-chip memory banks, respectively. In the coarse-grain FFT algorithm, we found that only the memory accesses in the last few stages are equally distributed to the four memory banks. The C64 hardware interleaves data across the four memory banks in a round-robin fashion, switching banks every 64 bytes (or 4 double precision complex elements). In early stages of the FFT computation, the twiddle factor array is accessed with a stride of a multiple of 4 elements as well. This in turn creates contention on bank 0 for those early stages as the same bank holds the required elements. Fig. 1 shows access rates (number of memory accesses per 3×10^6 cycles) of the 4 memory banks. We can see that *bank 0* has much higher access rate than the other three memory banks in the first 2/3 of the execution time. In the last 1/3 of the execution time, the access rates of the four memory banks are more balanced because the stride of the accessed addresses in the last few stages is less than 64.

In fine-grain execution models, an interesting finding is that a task in a later stage may be executed prior to a task in an early stage. Therefore, it is possible to reorder the execution of the tasks to get more balanced accesses to the memory banks. For example, the tasks in early stages have heavy memory contention on *bank 0*. If one moves some of these tasks to the end of the computation, the access rate may be reduced on *bank 0*. On the other hand, *bank 1,2* and *3* have light access rates in early stages. So we may execute some tasks of last few stages earlier since they have more accesses on *bank 1,2* and *3*. Fig. 2 shows the access rates of the off-chip memory banks in our designed fine-grain FFT algorithm. We can see that the access rate of *bank 0* is decreasing and those of the other 3 banks are increasing starting around the middle of the program execution.

An alternative solution to balance the memory access workload is to randomize the memory addresses of the elements in the twiddle factor array. However, the software randomizing apporach may introduce extra overhead. On the other hand, the hardware randomizing approach such as the memory address hashing feature on Cray XMT [10] may reduce the generality of the architecure because it is hard to exploit locality for regular applications.

III. Background

This section introduces the Cyclops-64 architecture, which is used as the testbed in our study, the existing study of the FFT algorithm on the architecture, and a fine-grain dataflow-inspired execution model (called the codelet model) as the basis of our fine-grain algorithm design.

A. The Cyclops-64 Architecture

Fig. 3 shows a block-diagram of a C64 node. Each node is a 160-core processor, clocked at 500 MHz. Each pair of cores

[1]In the FFT algorithm, twiddle factors are pre-computed constant trigonometric coefficients that are multiplied by the data in the course of the algorithm.

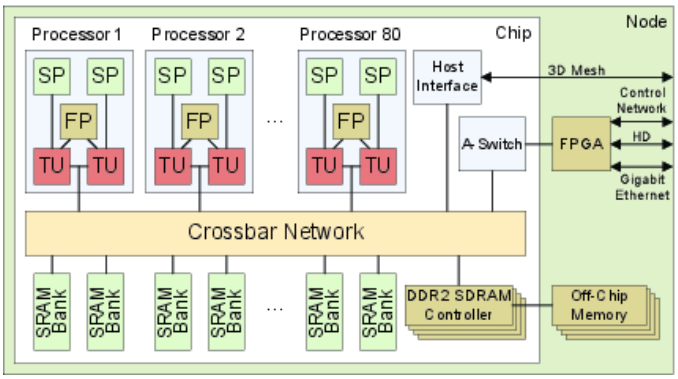

Fig. 3: The Cyclops-64 node block-diagram. Note that although scratchpads (SP) are logically separate from SRAM, in practice they are physically part of the same 30 kB bank.

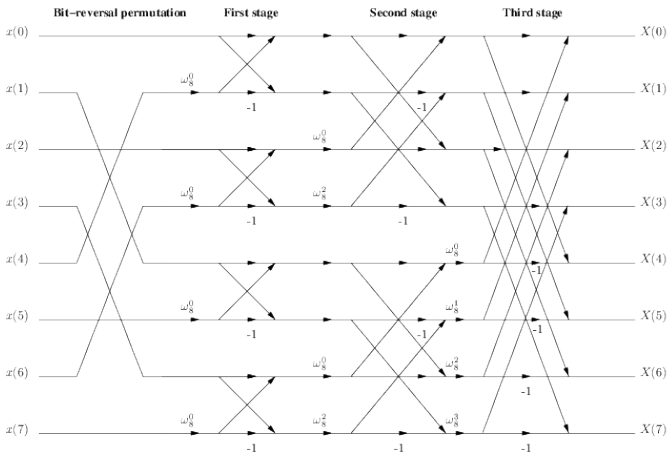

Fig. 4: An 8-point butterfly, as described by Chen *et al.* [11].

(called thread units, or TUs) share a floating-point unit, able to issue one fused multiply-add instruction (FMA) per cycle. Hence a single C64 chip yields a theoretical 80 GFLOPS peak performance. Each TU features a very simple in-order RISC architecture, and a register file composed of 64 64-bit registers. One TU normally runs one thread since it does not support context switching. A C64 chip features about 5 MB of on-chip memory, divided in 160 memory banks of 30 kB each. By default, these banks are equally divided into SRAM (accessible to all TUs) and scratchpad (local to a single TU)[2]. All TUs access the on-chip memory through a 96-port crossbar switch. The bandwidth to access SRAM (which totals around 2.5 MB) is 320 GB/s (640 GB/s in the case of scratchpad memory access). Each C64 board is equipped with 1 GB off-chip DRAM memory. Off-chip accesses to DRAM are significantly slower (16 GB/s). This is due to the fact that off-chip memory is only accessible through four banks: if one is not careful how DRAM is accessed, distributing data evenly on all four banks, then an imbalance can occur, saturating some ports and not some other. Load imbalance on DRAM ports for C64 is the reason behind the present work: balancing requests to access DRAM probably means reducing the biggest bottleneck during an application's life on C64.

B. Implementation of FFT on Cyclops-64

Chen *et al.* [11] have demonstrated how to implement 1D and 2D FFT on C64. They started from the classical Cooley-Tukey algorithm in an iterative fashion. They ensured that all threads were computing with the correct values by using the hardware barrier provided by C64 at various stages of the computation. Using only SRAM and the register file of each C64's thread unit, they then proceeded to expand the initial 2-point butterfly into an 8-point butterfly, as shown in Fig. 4. In the figure, the first step is called "bit reversal permutation" which changes the permutation of the initial input data in order to guarantee correct permutation of the final output data. This step is applied once and only once in the whole FFT computation. After that, the computation is partitioned into many small parallel tasks (called "work units" in [11]). Each work unit takes 8 data points as input, applies the 3-stage butterfly computation as shown in Fig. 4, and then store the result into memory (SRAM). The 8-point FFT algorithm brought several benefits which in turn improved performance:

1) The ratio between the amount of floating-point operations and data movement increased, thus keeping TUs busy doing useful work for a longer time.
2) By fetching 8 points of data instead of 2, there are fewer stages required to perform the FFT, thus requiring to use fewer barriers for synchronization.
3) Further observations of the behavior of FFT on C64 showed that for several stages of the computation, the same *twiddle factors* were used by all butterflies, doubling the number of different twiddle factors for each new computation stage. They exploited this fact to show how to save registers and use a 16-point butterfly as a single work unit.

Chen *et al.* assumed that all the data are stored in the on-chip memory. With this assumption, they figured out that an 8-point butterfly is the best work unit size because a larger size would require more registers than C64 provided. However, a large FFT problem may need to store data in the off-chip memory due to the limited on-chip memory space. In such a circumstance, we found that 64-point FFT performs better than the 8-point FFT because it reduces the total amount of off-chip memory accesses. The details are explained in Section IV .

C. The Codelet Model

The threading model followed by this study is inspired by the codelet program execution model (codelet PXM) [49]. While it does not exactly stick to the base model, the unit of computation (the codelet) is the same.

1) The Codelet Abstract Machine: Codelets are based on an abstract machine (AMM), which exposes a hierarchical, heterogenous topology. As Fig. 5 shows, the codelet AMM is made out of compute nodes linked by some interconnect. Each node is made out of at least one many-core chip. Each chip is composed of clusters of cores. Each cluster is composed of a few computation units (CUs), which are dedicated to the execution of codelets (and which can hold at least one

[2]In practice, the amount of shared SRAM vs scratchpad memory can be configured at boot-time.

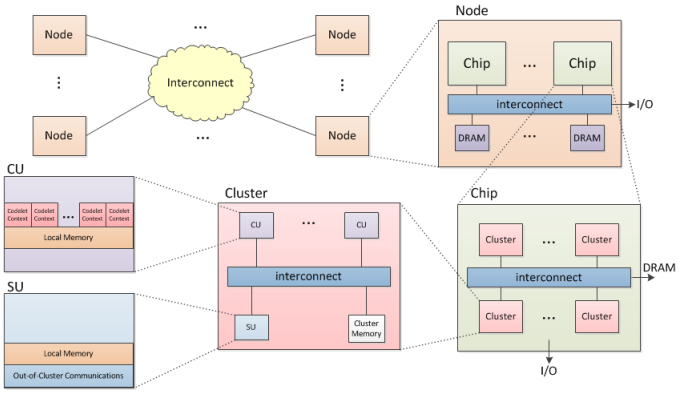

Fig. 5: The codelet abstract machine model. Computation units execute one of the ready codelets held in one of their codelet contexts. Synchronization units handle codelet scheduling at the cluster level. There is a memory module at each level of the hierarchy, shared by all components belonging to the same level.

Algorithm 1: The pseudo code of the coarse-grain 64-point FFT algorithm

input : Array D with input data
Array W with pre-computed twiddle factors
output: Array D with final results

PSEUDO CODE:

```
Bit_reversal(D) in parallel;
N←D.length;
last_stage ←⌈log2N/6⌉ − 1;
for stage = 0 to last_stage do
    if stage ≠ last_stage then
        for t_id = 0 to N/64 - 1 in parallel do
            FFT_64p_kernel(D,W,stage,t_id);
    else
        for t_id = 0 to N/64 - 1 in parallel do
            FFT_last_stage_kernel(D,W,stage,t_id);
    barrier;
```

codelet context, but can potentially hold more), and at least one synchronization unit (SU) which schedules codelets, handles interrupts, as well as off-cluster requests. At each level of the hierarchy, there is some memory available–e.g. each CU has some local memory available, each cluster has some "cluster memory" available to all CUs/SUs belonging to it, each chip has some memory accessible by all clusters, etc.

2) Definition and Operational Semantics of Codelets: The codelet PXM is a hybrid Von Neumann/dataflow based model [15]. A codelet is a sequence of non-preemptive machine instructions: they run until completion. It is associated with a synchronization slot which is updated to reflect the availability of data and/or resources required by a given codelet to start executing.

The operational semantics of codelets (also called *firing rules*) mainly follow dataflow semantics: they are *enabled* when all the data they require to run have been produced. However, codelets go one step further and also use resource availability as tokens: a codelet is *ready* when both its data and resource tokens are available to it. Then and only then can a codelet *fire* (i.e. start executing).

3) Codelet Graphs: Codelets are grouped into graphs, very much like dataflow graphs: the codelet graphs (CDGs). Codelet graphs are akin to data flow schemas [15]: if a codelet graph is well-behaved (i.e. there is no "structural" deadlock such as forming a cycle between regular codelets), then the computation will be determinate, i.e. for a given set of inputs fed to the CDG, the resulting outputs will always be the same; however the order in which the different intermediate steps are executed to reach the result may vary from execution to execution, depending on events occurring at run-time.

While in general CDGs are not necessarily completely statically defined, a property of the codelet model is to produce CDGs which feature at least *some* codelets and their dependencies statically. Other subgraphs can be spawned dynamically by active codelets if necessary. Our fine-grain FFT algorithms implicitly construct the CDG at the beginning of the execution once the input data size is known.

IV. Methodology

This section presents the methodology and various associated techniques which are the basis of our research. Section IV-A present the 64-point butterfly used to compute FFTs, using coarse-grain, fine-grain, and guided fine-grain methodologies; Section IV-B explains how we managed to ensure the randomization of off-chip memory addresses for balanced accesses to the off-chip memory banks. Each section describes pros and cons for these techniques.

A. Computing 64-Point FFTs

1) Coarse-Grain Algorithm: The 64-point FFT coarse-grain algorithm is a simple extension of Chen *et al.*'s 8-point FFT algorithm that is introduced in Section III-B.

The pseudo code of the 64-point FFT algorithm is shown in Alg. 1. At the beginning, the parallel bit-reversal process rearranges the position of the data to guarantee the correctness of the final result as done in the other Cooley-Tukey based FFT algorithms such as [11]. After that, the computation is partitioned into $\lceil log_2 N/6 \rceil$ stages, where N is the length of the input data array[3]. In each stage, there are $N/64$ tasks. Each task is a 64-point FFT kernel that loads 64 data points and 63 twiddle factors, applies butterfly computation on 6 levels, and stores 64 computed data in place. At the end of each stage, all the threads need to wait for a barrier. Tasks in the last stage may apply less than 6 levels of butterfly computation because $log_2 N$ may not be a multiple of 6. In such a case, tasks in the last stage only applies $log_2 N \bmod 6$ levels of butterfly computation.

Each task loads 64 data in the following way: Suppose that the task is the ith one in stage j. The thread will load $data_0$,...,$data_{63}$ from the data array D where

$$data_k = D[64^{j+1} \times \lfloor i/64^j \rfloor + i \bmod 64^j + k \times 64^j]$$

[3]Without loss of generality, we assume that N is a power of two because an input size can always be adjusted to a power of two by appending enough amount of zero data.

Algorithm 2: The pseudo code of the fine-grain 64-point FFT algorithm

input : Array D with input data
Array W with pre-computed twiddle factors
output: Array D with final results
Data: Q is a codelet pool that stores all the ready codelets
cnt is a 2-D array that counts the satisfied dependency of each codelet

PSEUDO CODE:

```
Bit_reversal(D) in parallel;
N←D.length;
last_stage ←⌈log2N/6⌉ − 1;
for t_id = 0 to N/64 - 1 do Q ← Q ∪ {(0, t_id)};
for each element e of cnt do e←0;
while Q ≠ ∅ in parallel do
    (stage, t_id)←Q.pop();
    if stage ≠ last_stage then
        FFT_64p_kernel(D,W,stage,t_id);
        for child=0 to 63 do
            child_id=Get_child_id(t_id,child);
            cnt[stage + 1, child_id] + +;
            if cnt[stage+1,child_id] == 64 then
                Q←Q ∪ {(stage + 1, child_id)};
    else
        FFT_last_stage_kernel(D,W,stage,t_id);
```

Moreover, the task also loads twiddle factors for each level of the FFT computation. At level l, the mth butterfly computation needs the twiddle factor

$$\omega_{lm} = W[m \bmod 2^l \times 2^{log_2 N - l - 1}]$$

2) Fine-Grain Algorithm: In the coarse-grain FFT algorithm, each task only needs 64 data and 63 pre-computed twiddle factors as input. The 64 inputs are provided by the 64 parent tasks in the prior stage. This implies that the barriers in the coarse-grain FFT algorithm may be removed. For example, a task in *stage 1* can start as long as its 64 parent tasks in *stage 0* have finished. It does not care about the completion of other tasks in *stage 0*. Based on this observation, we propose the fine-grain FFT algorithm as shown in Alg. 2. In the algorithm, the two *for* loops preceding the *while* loop take insignificant execution time. So we execute them sequentially.

We use the codelet model to represent the fine-grain FFT algorithm. As expressed in the model, each task in the coarse-grain FFT algorithm is a codelet in the fine-grain FFT algorithm. Each codelet is assigned a counter to count the number of dependencies that have been satisfied. There is a concurrent codelet pool to store all the codelets that have satisfied all the dependencies. Initially, the codelets in *stage 0* are all in the pool because their input data are ready. During the execution, once a thread completes a codelet, it will increase the dependency counters of all the children of the codelet. The child codelet that reaches 64 on its counter becomes ready and will be put into the codelet pool. The thread will then take the next codelet from the codelet pool. The algorithm terminates when the codelet pool becomes empty and all the threads complete their work. In practice, we found that this termination condition guarantees balanced workload as long as the total number of codelets are much larger than the total number of threads.

Let the parent codelet be the ith codelet in stage j, and its kth child be the lth codelet in stage $j+1$, then

$$l = \lfloor \frac{i}{64^{j+1}} \rfloor \times 64^{j+1} + i \bmod 64^{j+1} \bmod 64^j + k \times 64^j$$

In practice, we observe that every 64 children codelets share the same 64 parent codelets. That is, if codelets $A0,\ldots,A63$ are the 64 parent codelets of codelet $B0$, then there will be another 63 codelets $B1,\ldots,B63$ whose parents are also $A0,\ldots,A63$. For example, the 80th codelet in stage 3 is the 0th child of its 64 parent codelets in stage 2 if we apply $j=2$, $k=0$ and the following i to the above formula. The t_id (or i in the above formula) of its 64 parents are $80 + 4096 \times m$ where $m = 0, 1, \ldots, 63$. Using the above formula again, we can verify that the 4176th codelet in stage 3 is the next child of the same 64 parent codelets by applying $j=2$, $k=1$, and $l=4176$. Therefore, every 64 codelets may share a counter. In our implementation, the sharing greatly reduces the overhead of updating and checking the counters, as well as the storage requirement.

In the fine-grain FFT algorithm, codelet scheduling doesn't have to follow stage order as long as data dependencies are respected. Since the codelets in early stages have heavy memory contention on *bank 0*, executing them later reduces the bandwidth requirement on *bank 0* and improves the overall memory throughput.

3) Guided Fine-Grain Algorithm: Following the codelet execution model, the fine-grain FFT algorithm is determinate (see Section III-C3). However, the execution order of the codelets may be various in different runs. Both the initial arrangement of the codelets in the codelet pool and the execution speed of the codelets may affect the execution order. A good execution order may achieve more balanced memory accesses to the off-chip memory banks than a bad one. So an interesting question is how to guarantee a good execution order of the codelets. In this section, we discuss a guided approach to achieve the goal.

A good execution order implies more balanced access rates across the DRAM ports. Since only the codelets in the last few stages (especially the last stage) have a balanced workload to the off-chip memory banks, a good execution order should execute those codelets as early as possible. However, those late-stage codelets cannot be executed too early because they depend on the completion of many parent and ancestor codelets. On the other hand, the codelets in early stages have little chance to be executed very late. For example, a codelet c in *stage 0* is an ancestor of all the codelets in the last stage. Therefore, c has no chance to be executed last since many other codelets directly or indirectly depend on its completion.

Based on the above observation, we propose a guided fine-grain approach. We first partition the stages into two parts: We choose an integer i. Stages 0 to i are called *early stages* and the rest are called *late stages*. Then we apply two steps of the fine-grain FFT algorithm. The first step completes the codelets in the *early stages*. Then all the threads wait for a barrier

Algorithm 3: The pseudo code of the guided fine-grain 64-Point FFT algorithm

input : Array D with input data
Array W with pre-computed twiddle factors
output: Array D with final results
Data: Q is a concurrent LIFO codelet pool that stores all the ready codelets
cnt is a 2-D array that counts the satisfied dependency of each codelet

PSEUDO CODE:

```
Bit_reversal(D) in parallel;
last_stage ←⌈log2N/6⌉ − 1;
last_early_stage←last_stage − 2;
for t_id = 0 to N/64 - 1 do Q ← Q ∪ {(0, t_id)};
for each element e of cnt do e←0;
while Q ≠ ∅ in parallel do
    (stage, t_id)←Q.pop();
    FFT_64p_kernel(D,W,stage,t_id);
    if stage ≠ last_early_stage then
        for child=0 to 63 do
            child_id=Get_child_id(t_id,child);
            cnt[stage + 1, child_id] + +;
            if cnt[stage+1,child_id] == 64 then
                Q←Q ∪ {(stage + 1, child_id)};
barrier;
for every 64 codelets t_id0,...t_id63 of (last_stage - 1)
that have the same child codelets do
    Q ← Q ∪ {(last_stage −
    1, t_id0), ..., (last_stage − 1, t_id63};
for each element e of cnt do e←0;
while Q ≠ ∅ in parallel do
    (stage, t_id)←Q.pop();
    if stage ≠ last_stage then
        FFT_64p_kernel(D,W,stage,t_id);
        for child=0 to 63 do
            child_id=Get_child_id(t_id,child);
            cnt[stage + 1, child_id] + +;
            if cnt[stage+1,child_id] == 64 then
                Q←Q ∪ {(stage + 1, child_id)};
    else
        FFT_last_stage_kernel(D,W,stage,t_id);
```

to ensure the completion of all those codelets. In the second step, we use a last-in-first-out (LIFO) codelet pool to store the codelets of stage $i+1$ with a properly designed order that helps the codelets in the last stage to satisfy their dependencies as soon as possible. In such a way, the codelets in the last stage have higher chance to be executed earlier. It improves the memory access balance on the four off-chip memory banks. Alg. 3 shows our guided approach with the selection of the last two stages as the *late stage*. We execute the short *for* loops sequentially as in Alg. 2 because they take insignificant execution time.

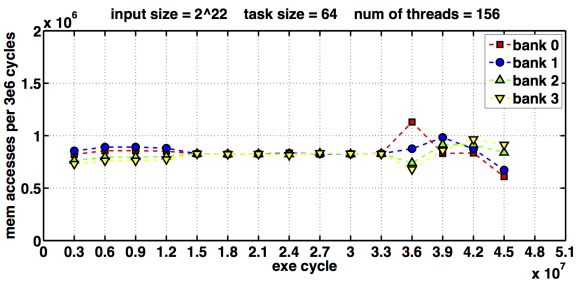

Fig. 6: Access rates of the 4 off-chip memory banks in the fine-grain FFT algorithm with randomized twiddle factor addresses. Using the hash function, all banks are accessed in a uniform manner.

B. Using a Hash Function to Randomize Bank Accesses in DRAM

An alternative technique to reduce the memory contention is to randomize the memory addresses of the elements in the twiddle factor array W. The randomization can be achieved by a perfect hash function

$$f : X \to X$$

where $X = \{0, 1, ..., M-1\}$ and M is the total number of elements in W. Now the ith element of W will be stored in $W[f(i)]$. In such a way, the addresses of all the elements in W are randomized. The accesses to them have the balanced workload on the four off-chip memory banks.

In practice, a perfect hash function is too expensive to implement. Instead, we use the bit reversal function BR to replace f. Let $i = (b_0 b_1 ... b_k)_2$, then BR is defined as follows:

$$BR(i) = (b_k ... b_1 b_0)_2$$

We choose bit reversal as the hash function because it is supported by hardware instructions on C64.

Fig. 6 shows the access rates of the 4 off-chip memory banks after the randomization of the twiddle factor addresses with the bit reversal function. We can see that the memory accesses on the four memory banks are balanced. However, it does not mean that the address randomization approach always achieves better performance due to the overhead of the hash function. The detailed experimental results will be explained in Section V .

V. Experimental Results

This section reports our experimental results, observations, and analyses on various FFT algorithms that are introduced in Section IV .

A. Task Size and Theoretical Peak Performance

In this section, we calculate the theoretical peak performance of the FFT application on the C64 chip. We assume that the input data size is large. So both the data array and twiddle factor array are located in the off-chip memory. The task (or codelet) size affects the theoretical peak performance because larger size has less amount of off-chip memory accesses. We

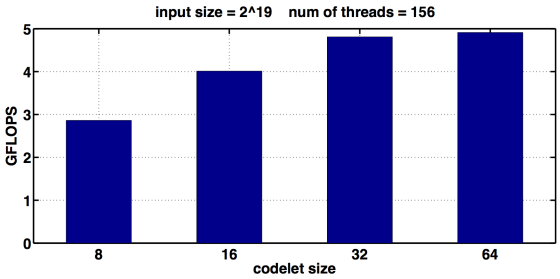

Fig. 7: The best performance of the fine-grain FFT algorithm for 156 threads units running in parallel and a global input data set of 2^{19}. 64-point FFT codelets perform best. The X axis shows the number of data points given as input to each codelet. The Y axis features the resulting performance (in GFLOPS).

choose 64 as the task (or codelet) size because a size over 64 may need too much on-chip space and exceed the scratchpad limit.

Fig. 7 shows the best performance of the fine-grain FFT algorithm under various codelet sizes. As we expected, 64-point FFT outperforms the algorithms with smaller codelet sizes.

The theoretical peak performance can be calculated as follows:

$$\begin{aligned} peak &= \frac{\#\ of\ floating\ point\ operations}{theoretical\ exectime} \\ &= \frac{5 \times N \times log_2 N}{exectime\ per\ task \times \#\ of\ tasks} \end{aligned} \quad (1)$$

$$\#\ of\ tasks = \frac{N}{64} \times \lceil \frac{log_2 N}{log_2 64} \rceil \quad (2)$$

$$exectime\ per\ task = \frac{(64+64+63) \times 16 Bytes}{DRAM\ bandwidth} \quad (3)$$

where N is the total number of data elements. To simplify the computation, we remove the ceiling function in (2). The removal does not reduce the theoretical peak performance because it will decrease the denominator in (1). Equation (3) is calculated as follows: Each task needs to load 64 elements from the data array, load 63 elements from the twiddle factor array, and store 64 elements to the data array. Each element takes 16 bytes because it is a double-precision complex number. When we assume that the off-chip memory is fully busy, we get the best execution time of a task as shown in (3). The DRAM bandwidth on C64 is 16 GB/sec as shown in [23], [24]. So we get the following theoretical peak performance from (1), (2), and (3).

$$\begin{aligned} peak &= \frac{5 \times N \times log_2 N \times 64 \times 6 \times 16G}{N \times log_2 N \times (64+64+63) \times 16} \\ &= 10\, GFLOPS \end{aligned} \quad (4)$$

As shown in (4), the theoretical peak performance of the FFT algorithm on C64 is 10 GFLOPS when the data array and twiddle factor array are located in the off-chip memory.

TABLE I: Description of the various methods used to perform FFT on C64. `fine best` and `fine worst` are results reported for the `fine` algorithm. Other results are named after the algorithm described in the right hand side column.

Name	Description
`coarse`	Coarse-grain synchronization (see Section IV-A1)
`coarse hash`	Coarse-grain synchronization with hashed twiddle factor array (see Section IV-B)
`fine worst`	Worst execution time for fine-grain synchronization (see Section IV-A2)
`fine best`	Best execution time for fine-grain synchronization (see Section IV-A2)
`fine hash`	Fine-grain synchronization with hashed twiddle factor array (see Section IV-B)
`fine guided`	Guided fine-grain synchronization (see Section IV-A3)

B. Experimental Setup

We implement FFT on the FAST simulator [11] which is a functionally-accurate simulator. It models the memory hierarchy of the C64 architecture, including the latencies and bandwidth of each memory segment. The input data are double-precision complex numbers and put into off-chip DRAM. The twiddle factors are pre-computed and stored in DRAM as well. We choose 64 as task size and vary input size from 2^{15} to 2^{22} using 156 threads. Besides, $20, 40, \ldots, 140, 156$ threads are used to run 2^{19} as the input size. We use 156 of the 160 threads because the remaining 4 thread units are reserved for the OS kernel.

In the experiments, we tested 5 versions of the FFT algorithms, and we report their results using 6 types of results. They are described in Table I: `coarse`, `coarse hash`, `fine` (divided between `fine worst` and `fine best`), `fine hash` and `fine guided`. In the fine-grain algorithm, we found that the initial order of the ready codelets in the concurrent pool may affect the performance a lot. So we show both the worst case and the best case of the fine-grain algorithm in Fig. 8 and 9 as `fine worst` and `fine best`, respectively.

C. Major Observations

From our experimental results, we made the following major observations:

1) The performance of `fine best`, `fine hash`, and `fine guided` are close and outperforms `coarse`, `coarse hash` and `fine worst` which also perform close;
2) `fine best` performs the best and `coarse hash` performs the worst;
3) When the input data size is small, `fine hash` outperforms `fine guided`. However, when input data size is large, `fine guided` outperforms `fine hash`.

The detailed results and analyses are explained in the following sections.

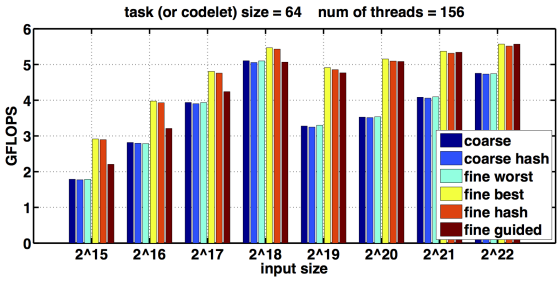

Fig. 8: Performance of 5 versions of FFT algorithms on C64. The X axis shows the various input sizes fed to FFT. The Y axis features the resulting performance (in GFLOPS). Higher is better.

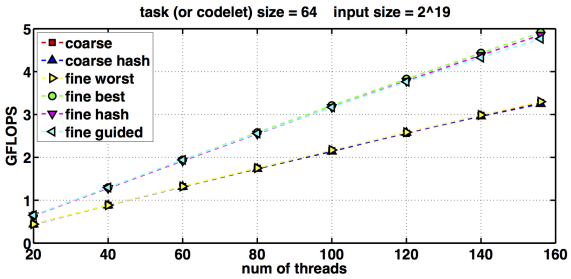

Fig. 9: Performance of 5 versions of FFT algorithms on C64 for an input size of 2^{19} data elements and 64-point butterfly codelets. The X axis represents the number of thread units used in the computation. The Y axis features the resulting performance (in GFLOPS). Higher is better.

D. Performance of the Various FFT Algorithms

Fig. 8 shows the performance of the 5 versions when the input data size varies from 2^{15} to 2^{22}. From Fig. 8, we observed that:

1) `fine guided` always performs between `fine best` and `fine worst` and close to `fine best`. This is because `fine guided` takes advantage of a proper codelet execution order that improves the memory access balance.
2) `fine hash` performs better than `fine guided` when the data input size is small. For example, when the data input size is 2^{18}, `fine hash` is 7% faster than `fine guided`. However, when the data size increases the `fine guided` becomes faster (e.g., 1% faster for 2^{22} input size) than `fine hash`. This is because the overhead of the bit reversal function increases on larger input sizes due to the work of handling more bits for each element. So our conjecture is that the performance gap between the `fine guided` and the `fine hash` will increase as the input data set gets higher. However, we are unable to test larger input sizes due to the time it takes to run our program on the simulator.
3) `coarse hash` always performs the worst, because of the combined overheads of the hash function and of the barrier which lead to unnecessary stalls in the codelet execution.

E. Scalability and Speedup

Fig. 9 shows the performance of the 5 versions of FFT algorithms on various number of working thread units. We tested from 20 to 156 thread units on the data input size 2^{19}. We do not test with fewer than 20 threads units or with larger input sizes due to the limitations of real-life execution time when using the simulator. From the figure we can see that:

- The `fine hash` and `fine guided` scale better than the other algorithms. They reach near linear speedup in our tests. `fine hash` and `fine guided` perform almost the same, with less than 1% difference. This is because both of the algorithms have more balanced workloads on the off-chip memory banks than the others. In fact, `fine hash` has an almost perfect balanced workload. However, it does not outperform `fine guided` due to the overhead of the hash function.
- `coarse` and `coarse hash` perform worst. For example, `fine guided` is about 46% faster than `coarse` whatever number of thread units we use. The reason are analyzed in Section IV-A1 and Section IV-B. `coarse` suffers from memory contention on *bank 0* in the early stages of the computation. `coarse hash`, however, suffers from the overhead of the hash function computation. Moreover, both algorithms suffer from the overhead of the barriers.
- `fine` exhibits unstable performance. When we exchange the initial order of the codelets, the performance fluctuates a lot. We found that `fine best` reaches more or less the same performance as `fine hash` and `fine guided`. Moreover, `fine worst` has more or less the same performance as `coarse` and `coarse hash`. This is because the different initial order affects the workload balance of the off-chip memory banks.

VI. Related Work

Efficiently computing FFT has been studied extensively in the past, both for sequential and parallel computations. One of the most famous works on this topic is FFTW [20]–[22]. It features a planner to decompose a (possibly multi-dimensional) FFT computation in a cache-oblivious (and architecture independent) way.

FFTW decomposes a given FFT computation into fragments called *codelets*. An FFTW codelet is some generated straight-line code which performs a specific fragment of an overall FFT computation. Hence, in FFTW linguo, codelets can perform real-inputs-only or complex-input-only FFTs, etc. However it is important to note that while the name is identical, the specification of what is a codelet differs from ours. Codelets, in the context of FFTW are units of work to perform FFT computations. As explained in section III-C, the codelets of the codelet execution model are dataflow-inspired: they obey specific rules to start executing, such as readiness of data, and explicitly define data

dependencies between themselves.

Other work also includes autotuning libraries such as UHFFT [3], [34] which reuses some techniques demonstrated in FFTW, and attempt to provide the best plan scheduling for multi-core systems. Of note is also PFFT [39] which uses FFTW as a core library, but enables the computation to run on large-scale systems.

Choi *et al.* demonstrate how a careful FFT algorithm, coupled with a specific multi-core architecture (using simple SIMD processing element arrays) can lead to better efficiency both for energy and performance [12]. Takahashi describes how to combine SIMD instructions with blocking on general purpose multi-core processors for 2D FFT by alternating transpose phases with computation phases [44]. Spiral is a digital signal processing program generator [19]. This framework performs code generation based on a domain-specific language for DSPs, and performs the appropriate C code generation according to the properties of the transforms described in the DSL: loop interchange, unrolling, etc. Spiral was extended to deal with FFT on multi-core systems and provide better work load-balancing and reduce false-sharing.

Additional research on FFT was applied to network-on-chips (NoC). Bahn compares the reference parallel FFT algorithm to two new algorithms applied to NoCs organized as a mesh-of-processing-elements [5]. Their study documents very well the requirements for work load-balancing on massively parallel chips. Mattson ports several compute and memory intensive kernels on Intel's experimental Terascale chip–among them, FFT [32]. However, due to architectural constraints (limited branching and indexing), the Cooley-Tukey algorithm was not chosen for their implementation. They instead chose the Pease FFT algorithm, which features fixed memory access patterns and lends itself well to systolic arrays–and therefore to mesh-like configurations.

As general purpose GPU (GPGPU) computing started to rise a few years ago, extensive research also looked into applying FFT to GPUs. Moreland showed early results of FFT computations on GPGPUs [35] (before the advent of Cuda and OpenCL). Because GPUs were still very constrained at the time, the study was done on single-precision values. Moreland uses classical techniques such as preliminary bit-reversal (also described by [11] and shown as the first step of a butterfly computation in Fig. 4) to reduce memory copies among the butterfly computations, frequency compression, etc. More recently, Volkov has shown how modern GPUs can be exploited to run FFT computations [47]. Volkov identifies a limitation of the G80 architecture: the bandwidth of non unit-stride memory accesses is significantly lower than than unit-stride ones. They evoke the possibility to perform additional reshufflings to circumvent this problem but did not implement it for their research (they instead laid the memory for FFT in a specific, predetermined order). Following their work on UHFFT, Franchetti *et al.* provided an overview of performing FFT computation on various multi and many-core architectures, including GPUs, FPGAs, and generic purpose multi-core systems [18]. Among other things, they use a different machine model for the GPUs/FPGAs than generic SMP/CMP systems. Lloyd and Govindaraju applied the radix-2 Stockham algorithm (which avoids the bit reversal preliminary stage of FFT) on GPUs [26], [30]. Garland *et al.* describe their implementation of various computation-intensive kernels, including FFT, on Cuda [25]. Dotsenko *et al.* propose an auto-tuning library on GPUs for generic FFT computations [16].

In a way closer to the C64 is the Cell Broad Engine hybrid multi-core microprocessor. Several efforts were produced to port FFT on Cell [4], [7], [13], [28]. Bader *et al.* implemented FFT on IBM Cell BE by applying barriers at every stage [4]. The Cell/B.E. architecture has many common features with C64–among them, the use of a scratchpad for SPEs. However, some fundamental differences (such as the availability of a DMA engine for the Cell/B.E.) make the methodology to perform computations on those two architectures very different.

FFT computations were also performed on FPGA systems [8], [27]. The implementations have notable differences, as the size of words is not necessarily comparable to the work presented in this paper (e.g. 16-bit words for Kamalizad's work [27]).

Whether they target small-scale multi-core systems (FFTW, UHFFT, FFTC) or large-scale ones (PFFT, Bahn's work for instance, or work on GPUs), the authors of these works rely on statically scheduling the units of work on a given machine, while trying to reduce the synchronization overhead. However, they still rely on coarse-grain synchronization (barriers) to go from one stage of the FFT to the other.

By contrast, the work described in this paper achieves high performance thanks to the dataflow-inspired semantics provided by the codelet model w.r.t. data availability, as well as a careful expression of data dependencies between 64-point butterfly computations. This in turn means using fine-grain synchronization instead of barriers on top of the dynamic scheduling scheme used at run-time.

The work that is most related to ours was performed by Saybasili [42] and Thulasiraman [46]. Saybasili *et al.* show how to efficiently port the radix-2 Cooley-Tukey algorithm on the heavily threaded XMT processor. The XMT proposes a hardware randomized address space, which is similar to our (software) hash-based implementation to access twiddle factors, but is applied to the whole system. The XMT's memory subsystem allows for a balanced workload, and thus a high throughput. Their algorithm is a fine-grain approach in the sense that it has 2-level parallelism. That is, the algorithm applies parallel 1-D FFT computation on every dimension of the 2-D data. However, their parallel 1-D FFT computation still needs barriers at the end of every stage. Our algorithm is more fine-grain because barriers between stages are eliminated by dataflow-driven synchronization.

It is worth noting that, at the time it was written, the XMT was implemented on an FPGA system, and was unable to perform floating-point operations, forcing the authors to use fixed-point arithmetic, which skews the comparison a bit, as

extra computations are required in the case of the XMT. Also, the implementation only considers on-chip performance.

Thulasiraman *et al.* compare fine-grain methodologies to express FFT algorithms using the EARTH model [45], the ancestor of the codelet model [49]. The major difference of the two approaches: Receive-Initiated and Sender-Initiated in [45] is the direction to establish dependency. However, both algorithms can only propagate one level at a time which is the same as our algorithm when task size is two. Due to the multi-level propagation in our algorithm, it saves remote accesses between two adjacent levels. Our own work not only guarantees good workload-balancing, but also efficient memory access balancing.

Finally, Long Chen's work on 8-point FFT butterflies has already been discussed in Section III-B. The main differences are that we extended Chen's work to 64-point butterflies using C64's scratchpad memory, and that our work reports performance for off-chip memory accesses. Finally, our approach uses fine-grain synchronization.

VII. Conclusions

In this paper, we use FFT as a case study to show the advantage of fine-grain execution models. We found that these models (such as the codelet model) allow for more freedom to reorder the execution of tasks than coarse-grain execution models. Since each task may have a different workload on the memory bandwidth, the freedom of reordering enables the system to better balance the memory bandwidth usage required by the running application. We designed and implemented a fine-grain FFT algorithm with a heuristic guidance of the execution order of the codelets to achieve good utilization of the memory bandwidth on the IBM Cyclops-64 many-core architecture. We show that our algorithm improves up to 46% performance comparing to a state-of-the-art implementation on the same system.

Acknowledgments

This research was made possible by the generous support of the NSF through grants CCF-0833122, CCF-0925863, CCF-0937907, CNS-0720531, and OCI-0904534. This research was also based upon work supported by the Department of Energy (National Nuclear Security Administration) under the Award Number DE-SC0008717. Moreover, this work was partly supported by European FP7 project TERAFLUX, id. 249013.

References

[1] A. Agarwal. The tile processor: A 64-core multicore for embedded processing. In *Proceedings of HPEC Workshop*, 2007.

[2] S. Alarm, R. Barrett, J. Kuehn, P. Roth, and J. Vetter. Characterization of Scientific Workloads on Systems with Multi-Core Processors. In *Workload Characterization, 2006 IEEE International Symposium on*, pages 225 –236, oct. 2006.

[3] A. Ali, L. Johnsson, and J. Subhlok. Scheduling FFT computation on SMP and multicore systems. In *Proceedings of the 21st annual international conference on Supercomputing*, ICS '07, pages 293–301, New York, NY, USA, 2007. ACM.

[4] D. A. Bader and V. Agarwal. FFTC: Fastest Fourier Transform for the IBM Cell Broadband Engine. In S. Aluru, M. Parashar, R. Badrinath, and V. Prasanna, editors, *High Performance Computing HiPC 2007*, volume 4873 of *Lecture Notes in Computer Science*, pages 172–184. Springer Berlin Heidelberg, 2007.

[5] J. H. Bahn, J. Yang, and N. Bagherzadeh. Parallel FFT Algorithms on Network-on-Chips. In *Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on*, pages 1087 –1093, april 2008.

[6] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 - Processor: A 64-Core SoC with Mesh Interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88 –598, feb. 2008.

[7] P. Bientinesi, N. P. Pitsianis, and X. Sun. Parallel 2d ffts on the cell broadband engine. *Submitted to International Journal of High Performance Computing Applications*, 2007.

[8] M. Butts. Synchronization through Communication in a Massively Parallel Processor Array. *Micro, IEEE*, 27(5):32 –40, sept.-oct. 2007.

[9] L. Chai, Q. Gao, and D. Panda. Understanding the Impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System. In *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, pages 471 –478, may 2007.

[10] D. Chavarria-Miranda, A. Marquez, J. Nieplocha, K. Maschhoff, and C. Scherrer. Early experience with out-of-core applications on the cray xmt. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 –8, april 2008.

[11] L. Chen, Z. Hu, J. Lin, and G. R. Gao. Optimizing the Fast Fourier Transform on a Multi-core Architecture. In *IPDPS*, pages 1–8, 2007.

[12] J. Choi, J. Kim, and C.-H. Kim. Parallel implementation of the FFT algorithm using a multi-core processor. In *Strategic Technology (IFOST), 2010 International Forum on*, pages 19 –22, oct. 2010.

[13] A. C. Chow, G. C. Fossum, and D. A. Brokenshire. A programming example: Large FFT on the Cell Broadband Engine. *Global Signal Processing Expo (GSPx)*, 2005.

[14] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl. Evaluation and improvements of programming models for the Intel SCC many-core processor. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 525 –532, july 2011.

[15] J. B. Dennis, J. B. Fosseen, and J. P. Linderman. Data flow schemas. In *International Sympoisum on Theoretical Programming*, pages 187–216, 1972.

[16] Y. Dotsenko, S. Baghsorkhi, B. Lloyd, and N. Govindaraju. Auto-tuning of fast fourier transform on graphics processors. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 257–266. ACM, 2011.

[17] D. Foley, P. Bansal, D. Cherepacha, R. Wasmuth, A. Gunasekar, S. Gutta, and A. Naini. A Low-Power Integrated x86–64 and Graphics Processor for Mobile Computing Devices. *Solid-State Circuits, IEEE Journal of*, 47(1):220 –231, jan. 2012.

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

[19] F. Franchetti, Y. Voronenko, and M. Püschel. FFT program generation for shared memory: SMP and multicore. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[20] M. Frigo. A fast Fourier transform compiler. In *Proc. 1999 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, volume 34, pages 169–180. ACM, May 1999.

[21] M. Frigo and S. Johnson. FFTW: an adaptive software architecture for the FFT. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 3, pages 1381 –1384 vol.3, may 1998.

[22] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[23] E. Garcia, D. Orozco, R. Khan, I. Venetis, K. Livingston, and G. R. Gao. Dynamic Percolation: A case of study on the shortcomings of traditional optimization in Many-core Architectures. In *Proceedings of 2012 ACM International Conference on Computer Frontiers (CF 2012)*, Cagliari, Italy, May 2012. ACM.

[24] E. Garcia, I. E. Venetis, R. Khan, and G. Gao. Optimized Dense Matrix Multiplication on a Many-Core Architecture. In *Proceedings of the 16th International European Conference on Parallel Computing (Euro-Par 2010), Part II*, volume 6272 of *Lecture Notes in Computer Science*, pages 316–327, Ischia, Italy, August 2010. Springer-Verlag.

[25] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel computing experiences with CUDA. *Micro, IEEE*, 28(4):13–27, 2008.

[26] 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*, page 2. IEEE Press, 2008.

[27] A. H. Kamalizad, C. Pan, and N. Bagherzadeh. Fast parallel FFT on a reconfigurable computation platform. In *Computer Architecture and High Performance Computing, 2003. Proceedings. 15th Symposium on*, pages 254 – 259, nov. 2003.

[28] Y. Li, J. R. Diamond, X. Wang, H. Lin, Y. Yang, and Z. Han. Large-scale fast Fourier transform on a heterogeneous multi-core system. *International Journal of High Performance Computing Applications*, 26(2):148–158, 2012.

[29] L. Liu, Z. Li, and A. H. Sameh. Analyzing memory access intensity in parallel programs on multicore. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 359–367, New York, NY, USA, 2008. ACM.

[30] D. Lloyd, C. Boyd, and N. Govindaraju. Fast computation of general Fourier Transforms on GPUS. In *Multimedia and Expo, 2008 IEEE International Conference on*, pages 5 –8, 23 2008-april 26 2008.

[31] A. Mandal, R. Fowler, and A. Porterfield. Modeling memory concurrency for multi-socket multi-core systems. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 66 –75, march 2010.

[32] T. G. Mattson, R. Van der Wijngaart, and M. Frumkin. Programming the Intel 80-core network-on-a-chip terascale processor. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 38:1–38:11, Piscataway, NJ, USA, 2008. IEEE Press.

[33] S. A. McKee. Reflections on the memory wall. In *Proceedings of the 1st conference on Computing frontiers*, CF '04, pages 162–, New York, NY, USA, 2004. ACM.

[34] D. Mirković, R. Mahasoom, and L. Johnsson. An adaptive software library for fast Fourier transforms. In *Proceedings of the 14th international conference on Supercomputing*, ICS '00, pages 215–224, New York, NY, USA, 2000. ACM.

[35] K. Moreland and E. Angel. The FFT on a GPU. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '03, pages 112–119, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[36] D. Orozco, E. Garcia, R. Khan, K. Livingston, and G. Gao. Toward high-throughput algorithms on many-core architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):49:1–21, January 2012.

[37] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879 –899, may 2008.

[38] F. Petrini, G. Fossum, J. Fernandez, A. Varbanescu, N. Kistler, and M. Perrone. Multicore Surprises: Lessons Learned from Optimizing Sweep3D on the Cell Broadband Engine. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1 –10, march 2007.

[39] M. Pippig. An Efficient and Flexible Parallel FFT Implementation Based on FFTW. In C. Bischof, H.-G. Hegering, W. E. Nagel, and G. Wittum, editors, *Competence in High Performance Computing 2010*, pages 125–134. Springer Berlin Heidelberg, 2012.

[40] P. Salihundam, S. Jain, T. Jacob, S. Kumar, V. Erraguntla, Y. Hoskote, S. Vangal, G. Ruhl, and N. Borkar. A 2 Tb/s 6×4 Mesh Network for a Single-Chip Cloud Computer With DVFS in 45 nm CMOS. *Solid-State Circuits, IEEE Journal of*, 46(4):757 –766, april 2011.

[41] A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the Memory Wall: The Case for Processor/Memory Integration. In *Computer Architecture, 1996 23rd Annual International Symposium on*, page 90, may 1996.

[42] A. Saybasili, A. Tzannes, B. Brooks, and U. Vishkin. Highly Parallel Multi-Dimensional Fast Fourier Transform on Fine-and Coarse-Grained Many-Core Approaches. In *Proceedings of the 21st IASTED International Conference*, volume 668, page 107, 2009.

[43] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):18:1–18:15, Aug. 2008.

[44] D. Takahashi. Implementation and Evaluation of Parallel FFT Using SIMD Instructions on Multi-core Processors. In *Innovative architecture for future generation high-performance processors and systems, 2007. iwia 2007. international workshop on*, pages 53 –59, jan. 2007.

[45] K. B. Theobald. *EARTH: an efficient architecture for running threads*. PhD thesis, McGill University, Montreal, Que., Canada, Canada, May 1999. AAINQ50269.

[46] P. Thulasiraman, K. B. Theobald, A. A. Khokhar, and G. R. Gao. Multithreaded algorithms for the fast Fourier transform. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, SPAA '00, pages 176–185, New York, NY, USA, 2000. ACM.

[47] V. Volkov and B. Kazian. Fitting FFT onto the G80 architecture. *University of California, Berkeley*, 40, 2008.

[48] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, Mar. 1995.

[49] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao. Using a "Codelet" Program Execution Model for Exascale Machines: Position Paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 64–69, New York, NY, USA, 2011. ACM.