# Performance Analysis of Cooley-Tukey FFT Algorithms for a Many-core Architecture

**Long Chen and Guang R. Gao**
**Department of Electrical & Computer Engineering**
**University of Delaware**
{**lochen, ggao**}**@capsl.udel.edu**

**Keywords:** Fast Fourier Transform, Performance Analysis, Many-core

## Abstract

Given that many-core architectures are becoming the mainstream framework for high performance computing, it is important to develop a performance model for many-core architectures to assist parallel algorithms design and applications performance tuning. In this paper, we propose a performance modeling technique for parallel Cooley-Tukey FFT algorithms, for an abstract many-core architecture that captures generic features and parameters of a class of real many-core architectures.

We derive the performance model by determining the cost functions for computation, memory access and synchronization in a parallel FFT algorithm. We have verified our performance model on the IBM Cyclops-64 (C64) many-core architecture, using both the simulator and a preliminary version of its chip. The experimental results demonstrate that our model can predict the performance trend accurately, with an average relative error of 16%, when running on up to 16 cores. The average relative error rate gradually increases to 29%, when running on up to 64 cores. The experimental results also reveal that key to performance for this class of many-core architectures is using the local memory and higher radix algorithms to reduce the memory traffic requirements.

## 1 Introduction

The fast Fourier transform (FFT) is of great use across a large number of fields, including spectral analysis, data compression, partial differential equations, polynomial multiplication, and multiplication of large integers [8, 19]. Various parallel FFT algorithms have been designed for numerous parallel computer systems [14, 18, 22, 1, 23, 4, 20, 25].

Many-core architectures are state-of-the-art parallel systems that offer massive thread level parallelism, massive on-chip memory bandwidth and other novel features. Examples of such architectures include Intel Terascale [17], Nvidia Tesla [21], and IBM Cyclops-64 (C64) [12]. While many-core architectures are becoming increasingly attractive platforms for high performance computing, it is difficult for programmers to fully explore their computing capabilities, partly due to a lack of performance modeling that can assist the design of parallel algorithms and direct applications performance tuning. For instance, when designing and tuning FFT algorithms, programmers may ask the following questions:

- What is the expected performance of an FFT implementation programmed in a high-level language for a many-core architecture?
- How does the performance of a parallel FFT algorithm change with the problem size?
- How scalable is an FFT algorithm, given a problem size?

A performance model that can answer these questions provides valuable assistance for designing FFT algorithms on many-core systems, and tuning them to achieve the maximum performance.

In this paper, we propose a performance model that estimates the performance of parallel FFT algorithms for a many-core architecture. This performance model is targeting for an abstract many-core architecture that captures generic features and parameters of several real many-core architectures. It is therefore applicable for any architecture with similar features. The parallel FFT algorithms we studied in our analysis are based on the one-dimensional decimation-in-time(DIT) Cooley-Tukey algorithm [7]. Our model can be easily extended to multi-dimensional FFT, which is calculated as a set of one-dimensional FFTs, performed along one dimension at a time. Note that the performance model presented in this paper is based on the nature of FFT: memory access pattern and computation pattern are fixed in advance. It is therefore applicable for other algorithms with the same property, but limited to general numerical algorithms.

We evaluate our model on the C64 architecture. Both simulations and the executions on the real hardware have verified the effectiveness of our performance model. The experimental results also reveal that the memory access delay has a crucial impact on performance of a parallel FFT algorithm for many-core architectures. Therefore programmers are suggested to optimize use of local memory and higher radix algorithms to reduce memory traffic requirements.

The rest of this paper is organized as follows. Section 2 presents the abstract architecture model. Section 3 presents two parallel FFT algorithms. The performance model is discussed in Section 4. Section 5 reports experimental results on the C64 architecture. Related work is summarized in Section 6, and Section 7 concludes with future directions.

## 2 Abstract Architecture Model

In this paper, we restrict our analysis to a class of many-core architectures. We abstract their main architectural fea-

tures into a generic form as illustrated in Figure 1. This abstract architecture model consists of a large number of identical cores/processors, each of whom has one or more processing elements (PEs), and a three-layer memory hierarchy, i.e., the local memory (LM), the on-chip global memory (on-chip GM), and the off-chip global memory (off-chip GM). The on-chip GMs and off-chip GMs are interleaved to achieve higher memory bandwidth. The memory hierarchy is explicitly software addressable to all cores. An on-chip interconnection network connects cores/processors to global memories. All PEs can access on-chip GMs and off-chip GMs via the network. An LM may or may not be globally accessed by all PEs, however, its associated core/PE can access it through some "backdoor" with very low latency. We simplify the interconnection network by assuming that the unloaded latency [10] of global memory accesses, either on on-chip GMs or on off-chip GMs, is equal, regardless of the origin or the destination of the access. Instances of such architecture include IBM C64, ClearSpeed CSX700 [6], etc.

To better understand the performance issues, we characterize this abstract model with a set of major architectural parameters, which are summarized in Table 1. These parameters and their denotations will be used in our following discussion. While there exist more general parallel machine models in the literature with fewer parameters, like LogP [9] and BSP [26], our abstract model (and the corresponding parameters) is developed for a specific class of state-of-the-art many-core architectures. Therefore it involves some low-level details, and we do not claim that this model/analysis can be immediately applied to a large diversity of parallel architectures.

**Table 1.** Architectural parameters

| | |
|---|---|
| $C$ | number of cores in one chip |
| $P$ | number of PEs in one core |
| $M$ | number of on-chip memory modules |
| $O$ | number of off-chip memory modules |
| $B_{in}$ | bandwidth of a inbound link between a core and the network, measured in bytes per cycle. |
| $B_{out}$ | bandwidth of a outbound link between a core and the network, measured in bytes per cycle. |
| $B_{net}$ | network bandwidth of the on-chip Interconnection network, measured in bytes per cycle. |
| $B$ | bandwidth of a single memory module, measured in bytes per cycle. |
| $W$ | granularity of the interleaved memory system, measured in bytes |

# 3 FFT Algorithms

In this section, we present two parallel FFT algorithms. The performance model presented in Section 4 is derived for the parallel FFT algorithms presented in this section.

## 3.1 Sequential FFT Algorithm

Figure 2 outlines Algorithm SEQ-R2-FFT, a sequential radix-2 DIT Cooley-Tukey algorithm. The input data and the pre-computed *long weight vector* [3], which is a stacking of twiddle factors used from the first stage to the last stage, are stored in array $x$ and array $\omega$, respectively. Note that a *bit-reversal permutation* needs to be performed on input before the butterfly computation stages. In this paper, we assume that $x$ has already been reordered, and therefore such permutation is not explicitly addressed in our algorithms.

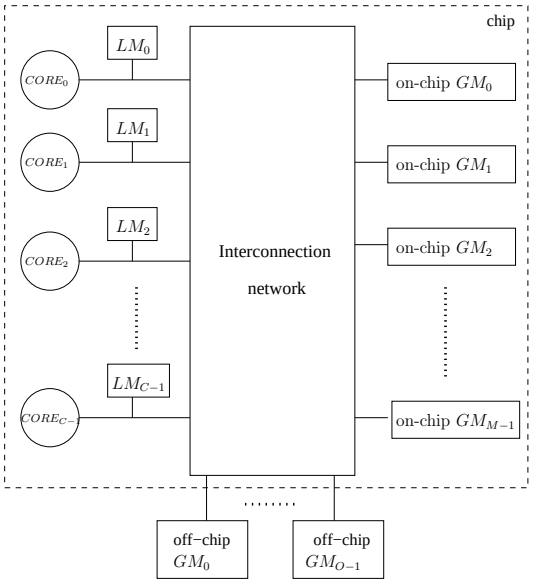


**Figure 1.** An abstract many-core architecture model

In $N$-point SEQ-R2-FFT, $N = 2^t$, we denote the data points accessed in the $i$-th butterfly, $1 \le i \le N/2$, in the $p$-th stage, $1 \le p \le t$, as $M(p,i)$. An interesting memory access pattern of SEQ-R2-FFT is captured in the following observation:

**Observation 3.1** *In the p-th stage of SEQ-R2-FFT, data points accessed from the* $a \cdot (2^c)$*-th* [1] *iteration to the* $((a+1)\cdot 2^c - 1)$*-th iteration, i.e.,* $\bigcup_{k=a\cdot 2^c}^{(a+1)\cdot 2^c-1} M(p,i)$*, constitute either one continuous data region, or two separate continuous data regions with equal lengths, where c is an integer between* 0 *and* $\log_2 N/2$*, and a is another integer between* 0 *and* $(N/2c) - 1$.

Due to the space limitation we omit the proof of this observation in this paper. Detailed discussion of this observation and its application to our performance modeling can be found in Section 4.3.

## 3.2 Parallel FFT Algorithms

Figure 3 presents a straightforward parallel version of Algorithm SEQ-R2-FFT, PAR-R2-FFT, which is executed by PE_e, $0 \le e < P \cdot C$. Barriers are used to ensure the read-after-write data dependence between stages. Similarly, a parallel radix-4 Cooley-Tukey algorithm, PAR-R4-FFT, is presented in Figure 4. In this algorithm, $\omega$ is a $\frac{N-1}{3}$ by 3 array, which is a stacking of twiddle factors and their squares and cubes used from the first stage to the last stage [19].

[1] We regard $2^0 = 0$ here.

```
Algorithm SEQ-R2-FFT
Input: a N-point data x, N = 2^t
Output: x overwritten with its DFT

1.    n ← 2^(t−1)
2.    for p ← 1 to t do
3.        l ← 2^p
4.        s ← 2^(p−1)
5.        for i ← 0 to n − 1 do
6.            k ← i/s
7.            j ← i mod s
8.            τ ← ω[s − 1 + j] ∗ x[kl + j + s]
9.            x[kl + j + s] ← x[kl + j] − τ
10.           x[kl + j] ← x[kl + j] + τ
11.       endfor
12.   endfor
```

**Figure 2.** Sequential radix-2 DIT Cooley-Tukey FFT algorithm

```
Algorithm PAR-R2-FFT
Input: a N-point data x, N = 2^t
Output: x overwritten with its DFT

1.    n ← 2^(t−1)
2.    for p ← 1 to t do
3.        l ← 2^p
4.        s ← 2^(p−1)
5.        for i ← e to n − 1 step P · C do
6.            k ← i/s
7.            j ← i (mod s)
8.            τ ← ω[s − 1 + j] · x[kl + j + s]
9.            x[kl + j + s] ← x[kl + j] − τ
10.           x[kl + j] ← x[kl + j] + τ
11.       endfor
12.       barrier
13.   endfor
```

**Figure 3.** Parallel radix-2 DIT Cooley-Tukey algorithm

Note that both Algorithm PAR-R2-FFT and Algorithm PAR-R4-FFT are straightforward parallel version of their sequential counterparts, and are not optimized for any specific architecture. While highly optimized algorithms could achieve much better performance [5], simple algorithms are beneficial to the illustration of our performance model.

# 4 Performance Estimation Strategy

In this section, we first introduce assumptions that are used throughout the paper. We then present the strategy to estimate the performance of parallel algorithms proposed in Section 3.

## 4.1 Assumptions

In order to simplify the modeling, we take the following assumptions.

- We assume that each core and memory bank has an infinite incoming buffer and an infinite outgoing buffer out of the network interface. Therefore, no request/response packet will be dropped.
- We assume that the problem size $N$ is much larger than the total number of PEs participating in the computation, i.e., $N \gg P \cdot C$, which is often true for most scientific applications.
- We do not consider the cost associated with the bit-reversal permutation, because it is not directly related to the cost of butterfly operations. Our model can be easily extended to incorporate this cost, though.

```
Algorithm PAR-R4-FFT
Input: a N-point data x, N = 2^t
Output: x overwritten with its DFT

1.    n ← 4^(t−1)
2.    for p ← 1 to t do
3.        l ← 4^p
4.        s ← 4^(p−1)
5.        ν ← (s − 1)/3
6.        for i ← e to n − 1 step P · C do
7.            k ← i/s
8.            j ← i (mod s)
9.            α ← x[kl + j]
10.           β ← ω[ν + j, 0] · x[kl + s + j]
11.           γ ← ω[ν + j, 1] · x[kl + 2s + j]
12.           δ ← ω[ν + j, 2] · x[kl + 3s + j]
13.           τ0 ← α + γ
14.           τ1 ← α − γ
15.           τ2 ← β + δ
16.           τ3 ← β − δ
17.           x[kl + j] ← τ0 + τ2
18.           x[kl + s + j] ← τ1 − iτ3
19.           x[kl + 2s + j] ← τ0 − τ2
20.           x[kl + 3s + j] ← τ1 + iτ3
21.       endfor
22.       barrier
23.   endfor
```

**Figure 4.** Parallel radix-4 DIT Cooley-Tukey algorithm

- If not explicitly stated otherwise, we assume that thread private data resides in local memories, and shared data, e.g, $x$, and $\omega$ reside in global memories.
- To further simplify the analysis, we assume that all architectural parameters are even numbers. In particular, $N$, $P$, $C$ and $W$ are powers of two.

Due to the above assumptions, we do not claim that our performance model can predict the accurate execution time of an application; rather, we attempt to use this model to quantitatively evaluate the performance impact (trend) of algorithms and architectural features on many-core systems.

## 4.2 Basic Strategy

Both Algorithm PAR-R2-FFT and Algorithm PAR-R4-FFT described in Section 3 have an iterative structure. More specifically, some synchronization-free computation pattern is repeated in every stage, and a global barrier is enforced after each stage to guarantee that all operations in that stage have completed. For example, an $N$-point PAR-R2-FFT proceeds in $\log_2 N$ stages, each of which composed of a set of independent butterfly operations evenly distributed among PEs. Each butterfly operation starts from loading two input data points and a pre-calculated twiddle factor from the global memory, followed by a computation kernel, and finally ends with storing two output points back to memory. The execution time of such $N$-point PAR-R2-FFT can therefore be calculated by

$$T_{FFT} = \sum_{r=1}^{log_2 N} max(T_C(r,p) + T_M(r,p) + T_B(r,p)) \qquad (1)$$
$$0 \le p < P \cdot C$$

where $T_C(r,p)$, $T_M(r,p)$ and $T_B(r,p)$ denote the computa-

tion time, memory access time and synchronization time, respectively, of PE $p$ in stage $r$. The execution time of a $N$-point PAR-R4-FFT can be obtained in a very similar way, except that the algorithm proceeds in $\log_4 N$ stages, and each butterfly operation works on a 4-point input dataset. For the sake of brevity, we focus our analyses on Algorithm PAR-R2-FFT, and only show the difference when necessary.

In our abstract architecture, all PEs are identical. Since butterfly operations in one stage are evenly distributed among PEs, and every butterfly takes the same amount of computation time, we can regard of $T_C(r,p)$ being same for all $r$ and $p$. We let $T_C$ denote the total computation time, i.e., the computation time for $N/2$ butterfly operations, in each stage.

Similarly, $T_B(r,p)$ can be regarded as same for all $r$ and $p$, since the semantics of a global barrier requires that all PEs wait at the barrier before any of them is allowed to proceed. We let $T_B$ denote the synchronization time immediately after each stage. Further, using $T_M(r)$ as the short for $max(T_M(r,p))$, we can rewrite Equation (1) as

$$T_{FFT} = \sum_{r=1}^{\log_2 N} T_M(r) + \log_2 N \cdot T_C + \log_2 N \cdot T_B \qquad (2)$$
$$0 \le p < P \cdot C$$

By deriving cost functions for $T_M(r)$, $T_C$ and $T_B$, we can quantitatively estimate the performance of Algorithm PAR-R2-FFT and PAR-R4-FFT on the abstract architecture model.

### 4.3 Estimated Memory Latency

We now derive the cost function for memory access delay for Algorithm PAR-R2-FFT.

**Estimated Local Memory Latency**

As described in Section 2, each PE can access its associated LM through some exclusive "back-door", without going through the network. Hence, we can simply treat the latency of accessing a PE's associated LM as a constant.

**Estimated Global Memory Latency**

In our abstract architecture model, the memory access delay of load/store operations issued to GMs is determined by the unloaded latency [10] and contention delays. The unloaded latency is the transmission time under ideal conditions. It is determined by the system design and is fixed for a given architecture. The contention delay occurs when multiple requests compete for some hardware resource. There are four types of contention delays in our abstract architecture model: (1) the *outbound* delay, when multiple PEs from the same core compete for a shared channel to inject memory access requests to the network, (2) the *network* delay, when multiple memory accesses compete for the network transmission, (3) the *memory contention* delay, when memory access requests are waiting to be handled by a memory bank, and (4) the *inbound* delay, when multiple data elements are loaded to the same core (for memory loads only). Each type of contention delays can be roughly calculated by dividing the size of the request (in bytes) by the average service rate (in bytes/cycle).

In Algorithm PAR-R2-FFT, the longest memory access delay occurs when multiple memory loads/stores are issued to GMs in a burst. Let $T_{ld}$ and $T_{st}$ denote the time (in cycles) to complete a burst of $P \cdot C$ load and store requests, one from each PE, respectively. $T_{ld}$ and $T_{st}$ can be represented as

$$\begin{aligned} T_{ld} &= \underbrace{\frac{P \cdot S_r}{B_{out}} - 1}_{outbound} + \underbrace{\frac{P \cdot C \cdot S_r}{B_{net}} - 1}_{network} + \underbrace{\frac{P \cdot C \cdot S_d}{B_m} - 1}_{memory} \\ &+ \underbrace{\frac{P \cdot C \cdot S_d}{B_{net}} - 1}_{network} + \underbrace{\frac{P \cdot S_d}{B_{in}} - 1}_{inbound} \end{aligned} \qquad (3)$$

$$T_{st} = \underbrace{\frac{P \cdot S_d}{B_{out}} - 1}_{outbound} + \underbrace{\frac{P \cdot C \cdot S_d}{B_{net}} - 1}_{network} + \underbrace{\frac{P \cdot C \cdot S_d}{B_m} - 1}_{memory} \qquad (4)$$

In the above equations, $S_r$ and $S_d$ are the size (in bytes) of a single request/response, with or without containing the data of $x$ or $\omega$, respectively[2]. Then $P \cdot S_r$ is the total size of load requests issued by a single core, and $P \cdot S_d$ is the total size of the store requests issued by a single core, or the total size of the response delivered back to a single core. Similarly, $P \cdot C \cdot S_d$ is the total size of data to be served for a memory access burst. $B_m$, the *aggregate effective memory bandwidth*, denotes the real achievable memory bandwidth (in bytes/cycle) when a burst of memory accesses are handled. Note that a memory load travels the network twice for sending the request and receiving the response, while a memory store travels the network only once.

Due to varieties of interconnection networks in topology, routing algorithms, switching strategy, and flow control mechanism, it is hard to induce a general equation for network delay, hence we focus on a type of interconnection network - crossbar switch - in this paper. The methodology presented here can be easily extended to other types of networks.

A crossbar switch is one form of the multistage networks that allows any input port to communicate with any output port in one pass through the network [16]. One important property of the crossbar switch is its *non-blocking connectivity* within the switch, which allows concurrent connections between multiple input-output pairs with a constant transmission time per packet, provided that inputs/outputs are always available during the connections [15]. For a many-core architecture employing a crossbar switch as the interconnection network, the components annotated with "network" in Equation (3) and (4) are constants. Furthermore, under our assumption that every core and memory bank has infinite incoming and outgoing buffers out of the network interfaces, $B_{in}$ and $B_{out}$ are considered as constants too.

[2] To simplify the expression, we assume that a load response and a store request are of an equal size. In the actual hardware, they may have different sizes. This fact does not affect our method presented here.

In this paper we focus on the cases where $x$ and $\omega$ are residing in the on-chip GM. Off-chip memory accesses usually involve more complicated hardware behaviors through the datapath, and the corresponding analysis will be a natural extension of the method presented in this paper.

For a many-core architecture employing a crossbar switch as the interconnection network, $B_m$ is an accumulated bandwidth of accessed memory banks. It is worth to note that the value of $B_m$ may be different for memory operations performed on $x$ and $\omega$, since, in a given stage, different PEs always access distinct data elements in $x$, while they probably attempt to load the same twiddle factor from $\omega$, especially in the first several butterfly stages. When multiple PEs access the same data, they introduce more contention in memory banks. This implies that different contention delay may occur when accessing $x$ and $\omega$ through the execution. To clarify this point, we denote $B_{m\_x}(r)$ as the aggregate effective bandwidth for loading/storing $x$ during stage $r$, and denote $B_{m\_\omega}(r)$ as the aggregate effective bandwidth for loading $\omega$ during stage $r$. To determine the exact value of $B_{m\_x}(r)$ and $B_{m\_\omega}(r)$, we assume, without any loss of generality, that $x$ and $\omega$ are aligned to a memory bank boundary.

In our analysis, we consider that a PE can issue load/store requests in a pipelined way, i.e., one request per machine cycle, which is true for modern architectures. A single radix-2 butterfly operation contains 3 load requests (2 for $x$ and 1 for $\omega$), and 2 store requests (for $x$). The completion time of the pipelined requests is determined by when the last request is finished, i.e., the longest delay. For a burst of radix-2 butterfly operations, we have the following equations to compute the latency $T_{ld\_p}$ and $T_{st\_p}$, for a pipelined load and store, respectively,

$$\begin{aligned} T_{ld\_p}(B_{m\_x}, B_{m\_\omega}) &= \underbrace{\frac{3P \cdot S_r}{B_{out}} - 1}_{outbound} + 2D + \underbrace{\frac{3P \cdot S_d}{B_{in}} - 1}_{inbound} \\ &+ \underbrace{max(\frac{2P \cdot C \cdot S_d}{B_{m\_x}} - 1, \frac{P \cdot C \cdot S_d}{B_{m\_\omega}} - 1)}_{memory} \end{aligned} \quad (5)$$

$$T_{st\_p}(B_{m\_x}) = \underbrace{\frac{2P \cdot S_d}{B_{out}} - 1}_{outbound} + D + \underbrace{\frac{2P \cdot C \cdot S_d}{B_{m\_x}} - 1}_{memory} \quad (6)$$

where the constant $D$ is the one way transmission latency of the crossbar switch.

**Determining $B_{m\_x}(r)$**

Since $B_{m\_x}(r)$ is an accumulated bandwidth of all accessed memory banks, the key issue is to determine the number of memory banks accessed in a burst of butterfly operations. Recall that Observation 3.1 in Section 3 states that data points accessed from iteration $a \cdot 2^c$ to iteration $(a+1)2^c - 1$ in one stage constitute either one continuous region or two separate continuous regions on $x$. This reminds us to transform the problem of calculating $B_{m\_x}(r)$ into determining the number of memory banks that are "covered" by those continuous region(s).

We first discuss the case where all accessed elements in $x$ during a burst constitute one continuous region, which happens during the first $\log_2(2P \cdot C)$ stages. Denote the length of such continuous region as $L_1$, where $L_1 = 2P \cdot C \cdot S_d$, and such region spans $\lceil \frac{L_1}{W} \rceil$ memory banks. We then have

$$B_{m\_x}(r) = min(M, \lceil \frac{L_1}{W} \rceil) \cdot B \quad (7)$$

Next we discuss the case where all accessed elements in $x$ during a burst constitute two separate continuous data regions with equal lengths, which happens in stage $r$, where $\log_2(2P \cdot C) < r \leq \log_2 N$. Denote the length of such regions as $L_2$, and denote the distance between two regions (i.e., the distance from the start of the first region to the start of the second region) as $Z$, where $L_2 = P \cdot C \cdot S_d$, and $Z = 2^{r-1} \cdot S_d$. Since both regions are aligned to the memory bank boundary, the number of memory banks on which each region spans is $\lceil \frac{L_2}{W} \rceil$, and the number of memory banks "covered" by the distance $Z$ is $\lceil \frac{Z}{W} \rceil$. Note that if $Z$ is long enough, then the second region might "fall off" the end of the last memory bank and "wrap around" to the start of the first memory bank, as illustrated in Figure 5(c).

If $\lceil \frac{L_2}{W} \rceil \geq M$, $B_{m\_x}(r)$ is simply $M \cdot B$, since all memory banks will be simultaneously active in serving memory access requests in a burst. Otherwise, we have to investigate the relative positioning of those two regions. Figure 5 lists all of three possible scenarios. In the figure, the shaded boxes represent memory regions, and the dotted arrow lines show the wrap-around.

**Scenario 1.** As shown in Figure 5(a), two regions are not overlapping on memory banks. This occurs when $(M \cdot W - 2L_2) \geq (Z \ mod \ (M \cdot W) - L_2) \geq 0$, that is, $(M \cdot W - L_2) \geq Z \ mod \ (M \cdot W) \geq L_2$. In this case, the number of covered memory banks is $\lceil \frac{2L_2}{W} \rceil$, and we have

$$B_{m\_x}(r) = \lceil \frac{2L_2}{W} \rceil \cdot B \quad (8)$$

**Scenario 2.** Figure 5(b) shows one kind of overlapping of two regions. When this scenario happens, it satisfies the condition $0 \leq Z \ mod \ (M \cdot W) < L_2$. The number of covered memory banks $Y = \lceil \frac{Z \ mod \ (M \cdot W) + L_2}{W} \rceil$, and we have

$$B_{m\_x}(r) = min(M, Y) \cdot B \quad (9)$$

**Scenario 3.** Another kind of overlapping is shown in Figure 5(c). The second region falls off the end of the last memory bank and wraps around to the first memory bank. In this scenario $Z$ satisfies the condition $Z \ mod \ (M \cdot W) \geq M \cdot W - L_2$. The number of covered memory banks is $\lceil \frac{L_2}{W} \rceil + \lceil \frac{M \cdot W - (Z \ mod \ (M \cdot W))}{W} \rceil$, and we have

$$B_{m\_x} = (\lceil \frac{L_2}{W} \rceil + \lceil \frac{M \cdot W - (Z \ mod \ (M \cdot W)}{W} \rceil) \cdot B \qquad (10)$$

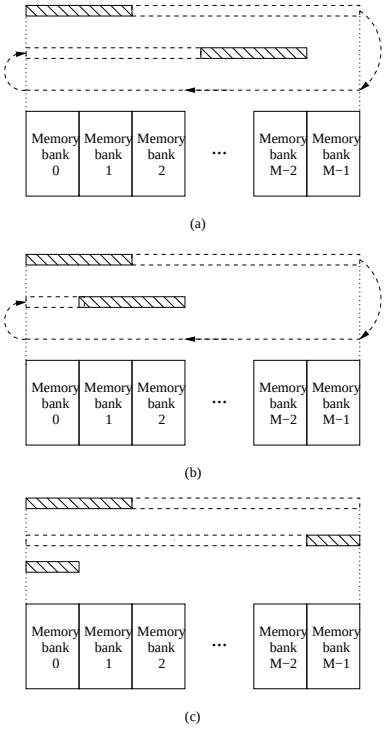


**Figure 5.** Relative positioning of two memory regions

**Determining $B_{m\_\omega}(r)$**

Determining $B_{m\_\omega}(r)$ is different from what we have done with $B_{m\_x}(r)$, since the number of distinct twiddle factors accessed in each stage varies through the execution. Here we list two possible scenarios.

**Scenario 1.** In the case of $P \cdot C \cdot S_d \leq W$, $B_{m\_\omega}(r)$ is always equal to $B$, because $P \cdot C$ requests of ω always fit into one memory bank[3].

**Scenario 2.** When $P \cdot C \cdot S_d > W$, $B_{m\_\omega}(r)$ can be easily determined as $B$ for the first $\log_2 \frac{2W}{S_d}$ stages, because the number of distinct twiddle factors used in each stage does not exceed $\frac{W}{S_d}$. For the rest stages, $B_{m\_\omega}(r)$ is mutually decided by the number of memory banks holding the twiddle factors used in stage $r$, and the number of requested (distinct) twiddle factors in a burst. This can be generalized as

$$B_{m\_\omega}(r) = min(2^{r - log_2(\frac{2W}{S_d})}, \frac{P \cdot C \cdot S_d}{W}, M)B \qquad (11)$$

Given Equations (5) to (11), $T_{m\_b}(r)$, the memory latency for a burst of radix-2 butterfly operations, one for each PE, in stage $r$, can be estimated by

$$T_{m\_b}(r) = T_{ld\_p}(B_{m\_x}(r), B_{m\_\omega}(r)) + T_{st\_p}(B_{m\_x}(r)) \qquad (12)$$

Since the workload is evenly distributed to all PEs, and every PE performs $\frac{N}{2P \cdot C}$ identical butterfly operations during each stage, the overall memory latency for Algorithm PAR-R2-FFT can be approximated as

$$\sum_{r=1}^{\log_2 N} T_M(r) = \frac{N}{2P \cdot C} \sum_{r=1}^{\log_2 N} T_{m\_b}(r) \qquad (13)$$

With slight modifications to Equations (5) to (13), one can obtain the overall memory latency for Algorithm PAR-R4-FFT. For example, to estimate the latency for a pipelined load for a radix-4 butterfly, instead of having 3 loads for a radix-2 butterfly shown in Equation (5), we simply substitute with 7 loads i.e., 4 for $x$ and 3 for ω.

### 4.4 Estimated Computation Time

To estimate the computation time, we examine the generated instruction sequence of the computation kernel, and use a simplified PE model to approximate the execution time, under ideal conditions: no interference from other PEs, no instruction fetch delays, and perfect branch prediction.

Since memory latency has already been taken care of in Section 4.3, all memory instructions are removed from the instruction sequence. The PE model executes the remaining instructions in a pipelined way, i.e., one instruction per cycle. Instructions are executed in-order, such that if one instruction is stalled due to data dependence, no later instruction can be issued. Special care needs to be taken when any shared hardware resource in a core is competed by PEs. Our PE model simply "perfect shuffles" $P$ sets of such instructions into a new sequence, in which all original data dependence relation is preserved, and executes this interleaved sequence. The estimated execution time of this interleaved instruction sequence is used as the execution time of a single set on this PE model. Given this model and the architecture specification, we can express $T_C$ as a function of $N$, $P$, and $C$.

### 4.5 Estimated Barrier Overhead

Given the complexity and variety of barrier implementations, it is difficult to estimate $T_B$ without knowing the details of the real architecture/software. We thus propose an experiment-based approach in our modeling. This approach makes every PE call the barrier function many times, and reports the average elapsed time per call. In this way, we can obtain the cost function of the barrier waiting time as a function of $P$ and $C$.

## 5 Case Study: IBM Cyclops-64

In this section we evaluate our performance model in the context of the IBM Cyclops-64 (C64) chip architecture.

### 5.1 C64 Chip Architecture

The C64 architecture is an instance of the abstract architecture model proposed in Section 2. A C64 chip contains 80

[3] $B_{m\_\omega}(r)$ could be a little bit larger than $B$, since the requested ω in a burst may reside in two consecutive memory banks; the first bank holds only one twiddle factor, and the next one holds at least $(\frac{W}{S_d} - 1)$ twiddle factors. However, the occurrences are few along the computation, and we approximate it as the bandwidth of a single bank.

cores. Each core has two single-issue, in-order PEs operating at a moderate clock race (500MHz), and a floating-point unit (FPU) shared by both PEs. The Instruction Set Architecture (ISA) of C64 supports *Floating Multiply-Add* instructions, which can be issued at every cycle. Therefore, the theoretical peak performance of a C64 chip is 80Gflops.

C64 features an explicitly addressable three-level memory hierarchy, including 160 local memories (LMs), one for each PE, 160 on-chip global memories (GMs), and 4 off-chip GMs. Both on-chip GMs and off-chip GMs are interleaved by a 64-byte boundary, and are accessible to all PEs on a chip. An LM is also accessible to all PEs, however, its associated PE can access it with a very low and fixed latency. There is no data cache in the C64 chip architecture.

All cores and memory banks are connected to an on-chip pipelined crossbar switch with $96 \times 96$ ports. In particularly, 80 ports are shared by 160 on-chip GM units, and 4 ports connected to the off-chip GM controllers. Each port can consume one request packet and send up to 8-byte data to the network/memory in one cycle, while all the other packets waiting in an associated FIFO queue. An important property of the crossbar switch is that memory access instructions issued by one PE to any on-chip GM (or off-chip GM) bank experience the same latency in the crossbar. This equal-latency property makes the on-chip memory model as sequential consistency [30], which implies that no "fence" instruction is required to enforce ordering relation between memory accesses. C64 provides no hardware support for context switch, and uses a non-preemptive thread execution model.

As a summary, Table 2 lists major architectural parameters of C64. $W_{GM}$ is the granulate of interleaved on-chip GMs. Since two on-chip GMs share one crossbar switch port, it can be approximated that there are 80 on-chip GM banks that are interleaved by a 128-byte boundary.

**Table 2.** Summary of C64 architectural parameters

| | |
|---|---|
| $C$ | up to 64 |
| $P$ | 2 |
| $M$ | 80 |
| $O$ | 4 |
| $B_{in}$ | 8 bytes/cycle |
| $B_{out}$ | 8 bytes/cycle |
| $B_{net}$ | up to 1140 bytes/cycle |
| $B$ | 8 bytes/cycle |
| $W_{GM}$ | 128 bytes |

### 5.2 Evaluations and Discussions

In this section, we present a set of extensive evaluations of the proposed performance model. We compare our estimations with experimental results obtained from a C64 simulator [11]. The experimental results show an average relative error of 16%, when running on up to 16 cores. This average relative error increases as more cores are used, and it reaches 29% at 64 cores. It is worth to note that similar results were obtained on a preliminary version of the real C64 chip.

**Estimated execution time on each stage.** We first want to compare the predicted execution time of each individual computation stage (plus the waiting time of the following barrier) with the experimental result, since its accuracy is the fundamental requirement for our subsequent analysis. Figure 6 and Figure 7 show such comparison when computing a $2^{10}$-point FFT with Algorithm PAR-R2-FFT and Algorithm PAR-R4-FFT, respectively. It can be observed that our performance

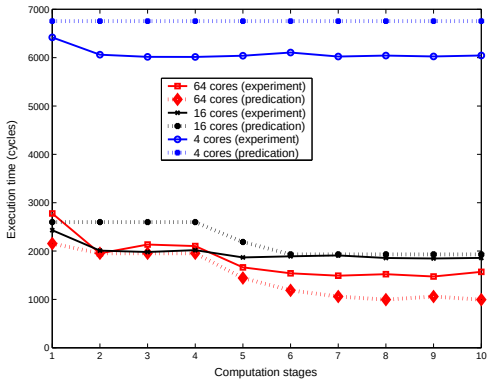


**Figure 6.** Execution time of stages, PAR-R2-FFT

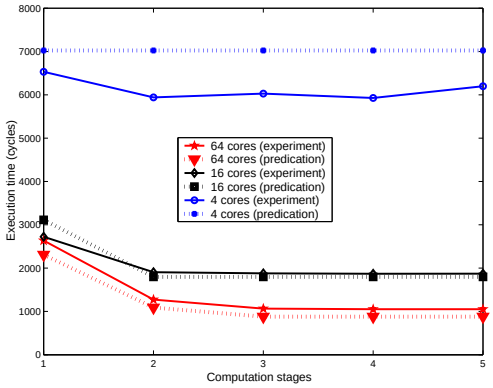


**Figure 7.** Execution time of stages, PAR-R4-FFT

model can predict the time spent on each stage with relative accuracy. Both figures show that the predicted time is $1\% - 29\%$ higher than the experimental execution time when running on up to 16 cores (part of the data are not shown in the figure). This difference is probably caused by our assumption that all instructions in a butterfly calculation must be stalled until all input data points, and the twiddle factors are loaded. In the actual system, however, one instruction can be executed as soon as all its operands are available and all its dependence relation is resolved, hence a long stall expected in our model can be avoided.

It can be also observed that when more cores are used (e.g., up to 64 cores), the predicted time is $5\% - 31\%$ lower than the experimental execution time (part of the data are not shown in the figure). One possible reason for this difference is that the

behavior of the crossbar network cannot be accurately captured by the current method under heavy traffic. We expect that this issue could be alleviated by incorporating a more accurate network model into our performance model.

**Performance impact as the problem size varies.** We investigate how the predicted execution time and performance change as a function of the problem size, when running on varied number of cores. The results of Algorithm PAR-R2-FFT are summarized in Figure 8 and Figure 9. Both figures demonstrate that our performance model correctly predicts the performance trend as the problem size increases, when compared with the experimental execution time and performance. Figure 9 shows that, when running on a large number of cores (e.g., 64 cores), the performance increases as the increase of the problem size, while it keeps flat when running on a small number of cores.

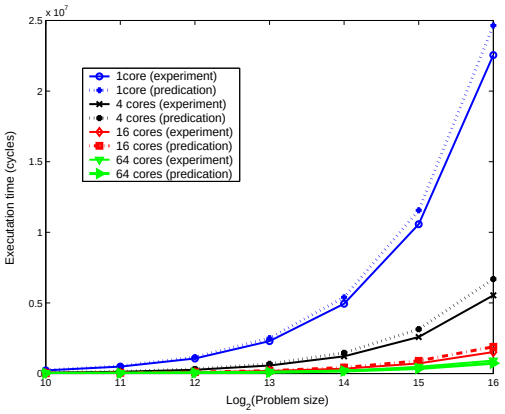


**Figure 8.** Total execution time versus the problem size, PAR-R2-FFT

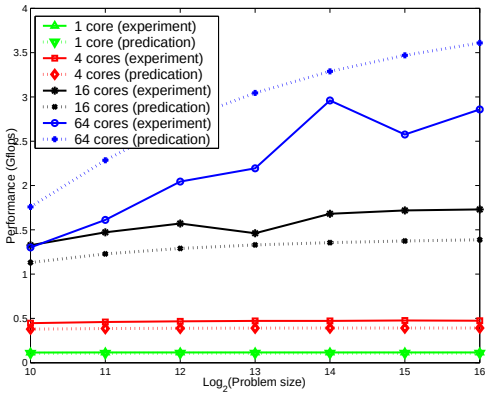


**Figure 9.** Performance versus the problem size, PAR-R2-FFT

**Performance impact as the number of cores varies.** We now show how the performance for a fixed input size changes with the number of cores. As shown in Figure 10, the estimations closely match the experimental results for all three problem sizes, when running on up to 32 cores. The difference between predicted and simulated performance is becoming rather noticeable, when running on a large number of cores, i.e., up to 29% difference when running on 64 cores. One possible reason is the inaccurate modeling under heavy traffic. Figure 11 shows the corresponding speedup curves.

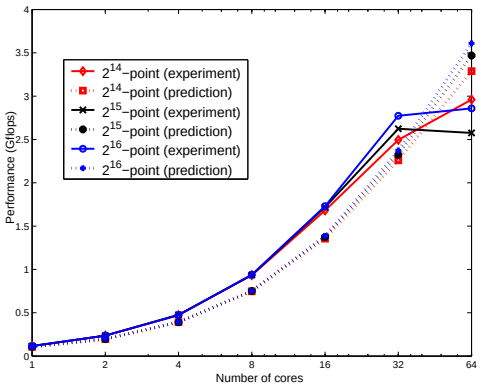


**Figure 10.** Performance versus the number of cores, PAR-R2-FFT

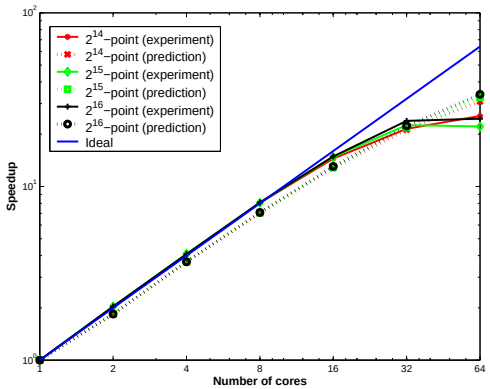


**Figure 11.** Speedup versus the number of cores, PAR-R2-FFT

**Performance impact as the algorithm changes.** From Figure 6 we can observe that when running on a large number of cores, the first several stages take a much longer time than the rest of stages. A careful investigation into both algorithms indicates that it is probably caused by the contention delay on loading the shared twiddle factors. For example, recall that $2^{i-1}$ $(1 \leq i \leq log_2 N)$ distinct twiddle factors are used in the $i$-th stage of Algorithm PAR-R2-FFT. In the first several stages a large number of PEs compete for loading a small number of twiddle factors, resulting in intensive contentions. Based on our performance model, both the accessing latency and the contention in the first stages could be greatly reduced, if each

PE keeps a local copy of twiddle factors in its associated LM. We then revised Algorithm PAR-R2-FFT according to this idea. We call this revised algorithm PAR-R2LM-FFT. Due to the limited size of the LM on the C64, in the real implementation, only twiddle factors used in stage 1 to 6 are stored in each PE's associated LM. In the rest of the stages, PEs still have to load the twiddle factors from GMs. The predicted execution time and the experimental execution time of Algorithm PAR-R2LM-FFT for a $2^{10}$-point FFT are shown in Figure 12. Compared with Figure 6, this new algorithm shows significant performance improvement in the first 6 stages. However,

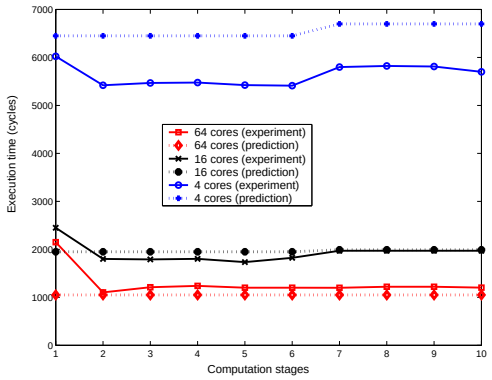


**Figure 12.** Execution time of stages, PAR-R2LM-FFT

even in the improved algorithm, memory access operations still cost about 300% – 500% more time than floating-point operations in a butterfly. This also explains why the achieved performance is far below the theoretic peak performance. One way to improve the performance is to use algorithms concerning data reuse, like higher radix algorithms , which can reduce memory traffic significantly. As shown in Figure 13, PAR-R4-FFT doubles the performance for various problem size - system configuration combinations, compared with PAR-R2-FFT. Our performance model shows that up to 140% performance gain could be achieved if a radix-8 FFT algorithm is used, compared with PAR-R2-FFT.

**Performance impact as the architectural parameters change.** Programmers and architects often want to know the performance impact of architectural changes to the existing algorithms. To this end, we consider a hypothetical many-core machine, C64+, which has the exact same configuration as C64, except that each core now has 4 PEs, instead of 2 in the original C64 design. We then apply our performance model with architectural parameters of this C64+ for Algorithm PAR-R2-FFT.

Figure 14 shows the predicated performance data for a $2^{10}$-point FFT and a $2^{16}$-point FFT. For the purpose of comparison we also include the experimental performance data obtained on C64 for these two problem sizes. From the figure we can observe that adding more PEs to a core does not yield a

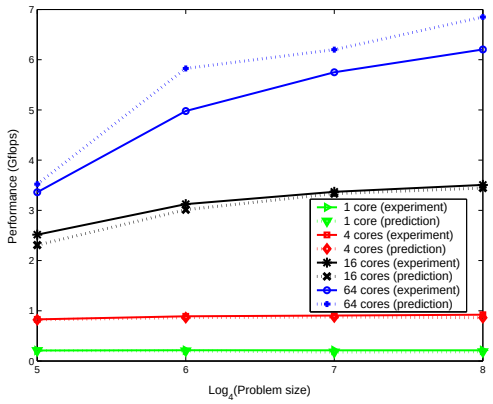


**Figure 13.** Performance versus the problem size, PAR-R4-FFT

significant gain of performance for our FFT algorithm. In particular, for the problem size of $2^{10}$-point, using more than 16 cores even has a negative performance impact. This is probably due to the increased memory contention delay and the longer barrier waiting time.

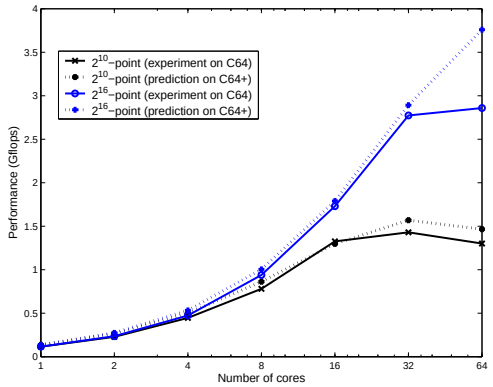


**Figure 14.** Performance predication for C64+, PAR-R2-FFT

## 6 Related Work

The most relevant previous work on performance modeling of FFT is the work by Cvetanović [27] on an abstract shared memory architecture. The work investigates the impact of the data layout on the memory access latency. Closed-form performance expressions are derived for the best-case and worst-case data layout. This work also approximates that memory operations regarding the input samples are issued by all processors in a burst. Our work differs from this work in several ways. First, while no specific algorithm is studied in [27], we present detailed analyses of two parallel FFT algorithms, together with experimental results on the real system. Secondly, the former study does not consider the memory traffic generated for loading the twiddle factors, and it assumes that the same network contention is produced during each stage, which may not be realistic for all FFT problems. Our work

investigates both issues, and take into account their effects upon the execution behaviors.

The technique of using instruction count to estimate the FFT performance is also used in [20], where several FFT algorithms are analyzed for IBM RP3 system. However, the work treats memory and synchronization delays as constants. Since the memory latency may vary due to the different memory access patterns through the execution, this assumption affects the accuracy of the results. Such issue has been explicitly taken into account into our analysis.

Due to the increasing complexity of modern architectures, empirical search has been introduced to find the optimal optimizations for several domain-specific problems, such as FFTW [13], ATLAS [2], and SPIRAL [24]. However, it is not clear how this method can be extended to many-core architectures. Moreover, as reported in [28, 29], carefully built model-driven optimization procedures show comparative or even better performance than the empirical search.

## 7 Conclusion and Future Work

The work presented in this paper is an attempt to quantitatively analyze the interaction between existing algorithms and the emerging many-core architectures. The model can be further improved in several dimension as discussed below. As we mentioned in Section 4, the analysis of off-chip GM accesses is a natural extension to the work presented in this paper. This is particularly important for the study of explicit data movement between levels of the memory hierarchy, which is used in many high performance FFT algorithms. It will be interesting to include analyses of such data movement, and thus verify the effectiveness of the existing FFT algorithms for many-core architectures. This performance model can be incorporated into an FFT computational framework, as a search engine to find suitable algorithms and optimal parameters for a given FFT problem. For example, as shown in Section 5, the performance model could identify the optimal number of PEs to be used for a given problem. Unlike an empirical search approach, by examining the properties of the algorithms and the architecture parameters, this performance model can potentially provide faster and more accurate solutions. Last, although our analysis presented in this paper is focused on the FFT algorithms, it will be interesting to investigate how the general methodology can be applied to other problems of statically defined communication and computation patterns, like matrix operations.

## References


[1] *Computational Arrays for the Discrete Fourier Transform*, 1981.

[2] C. W. Antoine, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the atlas project. *Parallel Computing*, 27:2001, 2000.

[3] D. H. Bailey. A high-performance fast Fourier transform algorithm for the Cray2. *Journal of Supercomputing*, 1:43–60, 1987.

[4] W. Briggs, L. Hart, R. Sweet, and A. O'Gallagher. Multiprocessor FFT methods. *SIAM J. Sci. Stat. Comput*, 8:27–42, January 1987.

[5] 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.

[6] ClearSpeed. ClearSpeed CSX700.

[7] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Math. Comput.*, 19:297–301, 1965.

[8] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[9] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1–12, 1993.

[10] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, inc., 1999.

[11] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. FAST: A functionally accurate simulation toolset for the Cyclops64 cellular architecture. In *MoBS'05*.

[12] M. Denneau and H. S. Warren, Jr. 64-bit Cyclops principles of operation part I. Technical report, IBM Watson Research Center, 2007.

[13] 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".

[14] A. G. and P. I. Parallel implementation of 2-d FFT algorithms on a hypercube. In *Proc. Parallel Computing Action, Workshop ISPRA*, 1990.

[15] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach, 4th edition*. Morgan Kauffman, 2007.

[16] J. L. Hennessy and D. A. Patterson. *Computer organization and design (3rd ed.): the hardware/software interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[17] Intel. Intel develops tera-scale research chips. http://www.intel.com, Sept. 2006.

[18] S. L. Johnsson and R. L. Krawitz. Cooley-tukey FFT on the connection machine. *Parallel Computing*, 18(11):1201–1221, 1992.

[19] C. V. Loan. *Computational framework for the fast Fourier transform*. SIAM, Philadelphia, 1992.

[20] A. Norton and A. J. Silberger. Parallelization and performance analysis of the Cooley-Tukey FFT algorithm for shared-memory architectures. *IEEE Transactions on Computers*, 36(5):581–591, 1987.

[21] Nvidia. NVIDIA Tesla many core parallel supercomputing.

[22] D. M. S. L. Johnsson, R.L. Krawitz and R. Frye. A radix 2 FFT on the connection machine. In *Proceedings of Supercomputing 89*, pages 809–819, 1989.

[23] V. Singh, V. Kumar, G. Agha, and C. Tomlinson. Scalability of parallel sorting on mesh multicomputers. In *IPPS'91*, pages 92–101, 1991.

[24] SPIRAL. SPIRAL website. http://www.spiral.net.

[25] P. N. Swarztrauber. Multiprocessor FFTs. *Parallel Computing*, 5(1-2):197–210, 1987.

[26] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[27] Žark Cvetanović. Performance analysis of the FFT algorithm on a shared-memory parallel architecture. *IBM J. Res. Dev.*, 31(4):435–451, 1987.

[28] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance blas?, 2005.

[29] K. Yotov, T. Roeder, K. Pingali, J. Gunnels, and F. Gustavson. An experimental comparison of cache-oblivious and cache-conscious programs. In *SPAA'07*, pages 93–104, New York, NY, USA, 2007. ACM.

[30] Y. Zhang, W. Zhu, F. Chen, Z. Hu, and G. R. Gao. Sequential consistency revisited: The sufficient conditions and method to reason consistency model of a multiprocessor-on-a chip architecture. In *PDCN2005*, page 12, Innsbruck, Austria, 2005.