# MEMORY OPTIMIZATION IN CODELET EXECUTION MODEL ON MANY-CORE ARCHITECTURES

by

Yao Wu

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering

Spring 2014




# MEMORY OPTIMIZATION IN CODELET EXECUTION MODEL ON MANY-CORE ARCHITECTURES

by

Yao Wu

Approved: _____

Guang R. Gao, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _____

Kenneth E. Barner, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: _____

Babatunde A. Ogunnaike, Ph.D.
Dean of the College of Engineering

Approved: _____

James G. Richards, Ph.D.
Vice Provost for Graduate and Professional Education

## ACKNOWLEDGMENTS

First of all, I want to express my deep gratitude to my advisor Prof. Guang R. Gao who guided and support in my research. His research attitude and enthusiasm have influenced me greatly. Under his guidance, I gained a lot of knowledge and skills. The experiences in CAPSL will benefit me for my future life.

I would like to acknowledge Dr. Chen Chen and Dr. Long Zheng who are my mentors and gave great help on my research. Dr. Chen Chen introduced OpenMP, SWARM, Cyclops-64 and parallel coding skills to me. I learned Hadoop and Java coding from Dr. Long Zheng. He provided feedback to revise my thesis. I can not accomplish my thesis without their help.

I also want to thank all CAPSL members who taught and helped me during my study.

Finally, I would like to give thanks to my parents, wife, and son. They always supported my work, understood me and gave me great patience.

This work is supported by the Department of Energy [Office of Science] under Award Number DE-SC0008717.

This work is partially supported by European FP7 project TERAFLUX, id. 249013.

## TABLE OF CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

## LIST OF ALGORITHMS

## ABSTRACT

The upcoming exa-scale era requires a parallel program execution model capable of achieving scalability, productivity, energy efficiency, and resiliency. The codelet model is a fine-grained dataflow-inspired execution model which is the focus of several tera-scale and exa-scale studies such as DARPA's UHPC, DOE's X-Stack, and the European TERAFLUX projects.

Current codelet implementations aim to making fully use of computation resources by balancing their workload in the multi-core and many-core systems. The performance is improved by this method. However, by making use of the features of the codelet model the memory optimization can be also implemented to improve the performance as well as energy efficiency. In this thesis, we focus on the memory optimization on memory workload balance and locality exploitation in the codelet model. As a case study, various versions of FFT algorithms are implemented on IBM Cyclops-64 – a many-core system to demonstrate that the fine-grain codelet execution model is able to execute the codelets that involve different workload on the memory bandwidth in an appropriate order to reduce memory contention and thus improve performance. The experiment result shows that our fine-grain guided algorithm achieves up to 46% performance improvement comparing to a coarse-grain implementation on Cyclops-64. To automatically exploit locality in codelet execution, we provide three optimal or nearly optimal scheduling algorithms based on static information of codelet graph and locality. They have different trade-offs in algorithmic complexity, locality exploitation, program execution time, and energy efficiency. We test and analyze the three algorithms on various applications on an emulation platform of Cyclops-64. The experiment result shows that our algorithms reduce up to 59.7% of global memory access by using local memory to buffer intermediate data between two adjacent codelets

on the same core and thus improve up to 68.1% performance improvement and 40.7% energy saving comparing to the dynamic codelet scheduling approach.

# Chapter 1

# INTRODUCITON

To achieve better performance, more and more cores are integrated into systems. Multi-core and many-core systems are becoming popular and many have been available on market [2, 6, 67, 20, 27, 65, 60]. Therefore, the approaches to effectively utilizing these systems draw considerable attention. Conventionally, parallel execution in multi-core and many-core systems is based on coarse-grain synchronization by using barriers *e.g.* MPI and OpenMP that are the most prevailing parallel programming models today. These coarse-grain models perform well on a system where the number of cores is small, but tend to degrade as the core count increases [38, 58]. The reason is that the contention for shared resources *e.g.* shared memory and floating-point units becomes severer among cores as the number of cores increases.

For this reason, different execution models are required to achieve high-performance computing for multi-core and many-core systems. The codelet model is designed to realize fine-grain parallel execution for extreme-scale machines. This model is originated from dataflow and also takes advantage of the Von Neumann model. In the codelet model, the programs can be expressed by codelets and the dependencies among them. Runtimes based on codelet execution model are able to schedule ready codelets to run on available resources. Comparing to the coarse-grain model, the asynchronously event-driven execution in the codelet model can achieve more balanced workload on computation.

Moreover, by making use of the features of the codelet model the memory optimization can be implemented to improve the performance as well as energy saving in

the mutli-core and many-core systems. In this thesis, we focus on the memory optimization on memory workload balance and locality exploitation in the codelet execution model.

In this thesis, we show that the fine-grain codelet execution model can achieve more balanced workload on not only computation but also memory bandwidth than the coarse-grain execution model can. The codelet execution model provide an opportunity to control the order of the codelet execution. Because each codelet maybe has different memory access patterns, the execution order can be controlled to provide a evenly distributed access pattern among memory banks such that the system achieves better workload balance on the memory bandwidth usage. We use FFT algorithm on the IBM Cyclops-64 many-core architecture to demonstrate the advantage of the fine-grain execution model by using codelet model.

Although the codelet model is able to exploit parallelism effectively to improve workload balance, data locality is not well considered when scheduling codelet. In some multi-core and many-core systems, system memory is organized hierarchically by globally shared memory among cores and local memory for each core *e.g.* IBM Cyclops-64 [21], IBM CELL Broadband Engine [1], and Intel UHPC straw-man [48]. Globally shared memory *e.g.* interleaved SRAM and DRAM are used to share data and communicate/synchronize among cores. Local memory *e.g.* scratchpad SRAM is used to store the data for further reuse. The access latency and energy consumption of local memory is much lower than that of shared global memory. For such systems, locality exploitation is very important, because it achieves better performance and more efficient energy consumption. The conventional method to exploit data locality highly relies on programmers to manually generate the scheduling plan. This process is extremely time-consuming. In this thesis, our effort is to introduce a automatic mechanism to guide codelet runtimes in exploiting locality by using the codelet graph and locality information. This automatic approach can exploit best data locality as well as keep the highest parallelism in programs by using static scheduling in codelet runtimes.

The major contributions of this thesis are as follows:

- A fine-grain FFT algorithm based on codelet model is designed and implemented on the IBM Cyclops-64 many-core architecture. With a heuristic order guiding the execution of the codelets, the memory contention is reduced.
- The behavior of three versions of FFT on different granularity of synchronization are compared: coarse-grain (using barriers), fine-grain (using point-to-point synchronization), and guided fine-grain (fine-grain with the heuristic guidance). The experiment result shows that our algorithm achieves up to 46% performance improvement comparing to the coarse implementation on Cyclops-64.
- The fine-grain approach with an alternative solution that reduces memory contention by randomizing memory addresses is implemented and compared. The fine-grain approach outperforms the address randomization approach when the input data size is large enough. Moreover, the performance gap will enlarge as the input data size increases.
- A polynomial-time algorithm is proposed to statically scheduling codelets to achieve best data locality while keeping the highest parallelism when there are enough computation resources. The least computation resource requirement for running parallel programs is also guaranteed by the algorithm.
- Other two widely used algorithms are analyzed and implemented to make a comparison. These three algorithms have different trade-offs in algorithmic complexity, locality exploitation, program execution time, and energy efficiency. Theoretically and experimentally, our algorithm provides the best performance as well as the most efficient energy consumption.
- These three algorithms are tested and analyzed on various applications including matrix multiply, merge sort, and random generated codelet graphs with reasonable assumptions. The experimental results show that our algorithm can reduce up to 59.7% of global memory access by optimizing locality exploitation. Our algorithm also improves up to 68.1% performance improvement and 40.7% energy saving comparing to the dynamic codelet scheduling.

Our work described above has been published in the international conference and workshop. In particular, this thesis is also based on the published papers [14, 12, 13].

The rest of the thesis is organized as follows. Chapter 2 provides the background on the architecture, execution model, and FFT algorithm of our work in this thesis. Chapter 3 demonstrates the memory workload balance optimization in the codelet model by using FFT algorithm. Chapter 4 introduces the memory locality exploitation

optimization in the codelet model by comparing different scheduling algorithms on various applications. Chapter 5 gives the conclusion and future work of our work.

# Chapter 2

# BACKGROUND

In this chapter, we introduce the evolution of multi- and many-core systems in Section 2.1, the IBM Cyclops-64 many-core architecture that is used as the experimental platform in Section 2.2, the previous work on the FFT algorithm on Cyclops-64 in Section 2.3, and the codelet model by which we implement various fine-grain algorithms in Section 2.4.

## 2.1 Overview of Multi- and Many-core Systems

As the requirements for computation are drastically increasing, the manufacturers achieve the microprocessor performance mainly in two ways. One is based on the frequency boost of a microprocessor, so that more cycles are in one second *i.e.* more instructions can be executed in a second. On the other hand, as more transistor can be integrated into a microprocessor (since a transistor is getting smaller and smaller), Instruction Level Parallelism (ILP) is achieved to execute more instructions in a cycle. Therefore, both approaches improve the performance of microprocessors. Instruction Level Parallelism has been widely studied over many years, such as super scale, out-of-order execution, long pipeline, multi-issue, branch prediction, and speculation. However, high clock frequency brings in thermal problem and small transistor tends to reach its physical limit. Therefore, the two methods are not suitable for performance enhancement any more.

Recently, the microprocessor industry is tending toward multi-core and many-core. It is an easier and more efficient way to improve the performance of microprocessors by integrating more cores into a chip. Many multi-core and many-core architectures are designed to demonstrate the significant computation power. The CELL

Broadband Engine [1] processor is a heterogeneous multi-core microprocessor which provides powerful graphics processing performance. It consists of a Power Processor Element (PPE) which controls eight SIMD Synergistic Processor Element (SPE). IBM Cyclops-64 [21] (see Section 2.2) contains 80 processors each of which has two thread units. Intel UHPC straw-man [48] architecture is composed of three levels: chip, cluster, and block. A chip consists of a group of clusters. A cluster contains many blocks. In a block, there are $N$ execution engines (XE) and one control engine (CE). Network processors contain tens of thousands of cores to enhance and optimize packet processing in the networks [4]. GPU is widely used in graphics processing as well as supports the general-purpose computation [44]. Nowadays, it can contain up to thousands of cores. Different architectures involve different hardware features which can be taken advantage to optimize specific computations. After the microprocessors evolve into multi-core and many-core, not only Instruction Level Parallelism but also Thread Level Parallelism (TLP) are used to enhance the performance. The difference between multi-core and many-core is the number of cores integrated in a chip. In general, multi-core refers to the systems with eight or less cores, such as most of the Intel x86 CPUs with a small number of cores. Many-core refers to the systems with more than eight cores, such as Cyclops-64, Intel straw-man, and GPU. Nowadays, multi-core and many-core are becoming the mainstream in computer systems. However, it also brings in challenges to effectively make use of their computation power as well as improve energy efficiency.

In this thesis, we focus on both performance and energy optimization on multi-core and many-core architectures by using fine-grain codelet execution model (see Section 2.4) instead of the conventional coarse-grain approaches, such as MPI and OpenMP.

## 2.2 The Cyclops-64 Architecture

Figure 2.1 shows a block-diagram of a Cyclops-64 node. Each node runs at 500 MHz and contains 80 processors. Each processor contains 2 thread units (TU)

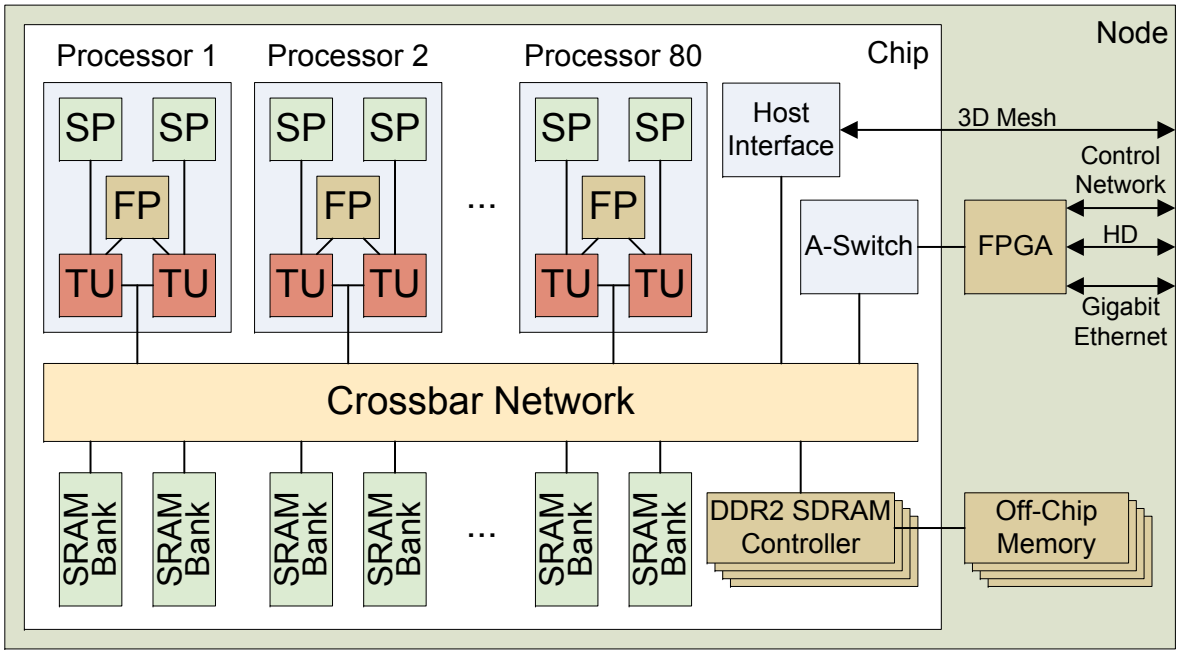


**Figure 2.1:** The Cyclops-64 node block-diagram. A node has 80 processors each of which contains 2 thread units (TU), a floating-point unit (FP), and local scratchpad SRAM (SP). All processors access global SRAM and off-chip DRAM through the crossbar network.

which share a floating-point unit (FP). A thread unit is an in-order 64-bit RISC core with a register file composed of 64 64-bit registers. It does not support for context switch, so each TU only runs one thread. Each FP is able to issue one fused multiply-add instruction (FMA) per cycle. Hence, the theoretical peak performance of a Cyclops-64 node is a 80 GFLOPS.

As shown in Figure 2.2, Cyclops-64 has a three layer memory hierarchy without data cache: scratchpad SRAM, interleaved SRAM, and DRAM which are accessed by all TUs through a 96-port crossbar switch. A Cyclops-64 node is equipped with about 5 MB on-chip memory SRAM, which is divided into 160 memory banks of 30 KB each. By default, these banks are equally split into interleaved (global) SRAM which can be accessed by all TUs and scratchpad (local) SRAM which is local to and can be accessed much faster by the corresponding TU). Note that the amount of global SRAM

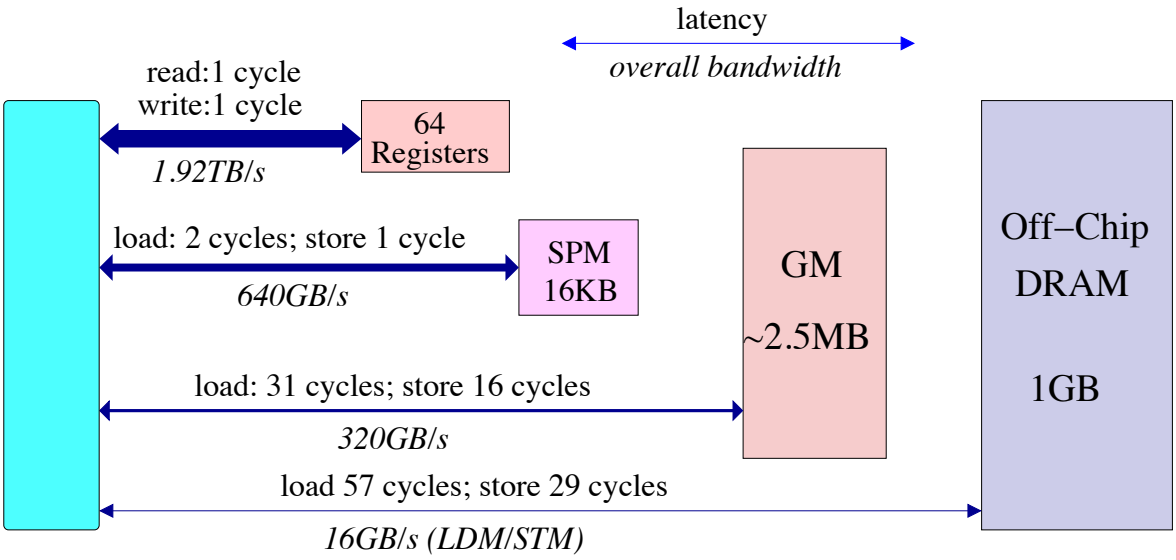


**Figure 2.2:** The Cyclops-64 memory hierarchy. A node has a three layer memory hierarchy without data cache: scratchpad SRAM, interleaved SRAM, and off-chip DRAM. Each layer has different memory bandwidth and access latency.

and scratchpad memory can be configured at boot time. The bandwidth to access interleaved and scratchpad SRAM are 320 GB/s and 640 GB/s respectively. There is 1 GB off-chip DRAM memory on a Cyclops-64 node. Off-chip DRAM accesses are significantly slower from 16 GB/s for multiple-load / multiple-store instructions down to 2 GB/s for sequences of single-load or single-store instructions. Off-chip memory is only accessed through 4 banks. Workload imbalance on DRAM ports on Cyclops-64 becomes serious if the data does not distribute evenly on each bank. In this case, some ports are so busy that causes contentions and access delay while others are free. Our work shows that the codelet model provides the possibility to balance DRAM access pattern to achieve better performance.

## 2.3 Implementation of FFT on Cyclops-64

Fast Fourier Transform (FFT) is a very useful algorithm in signal processing area. Many different implementations of FFT on different architectures have been

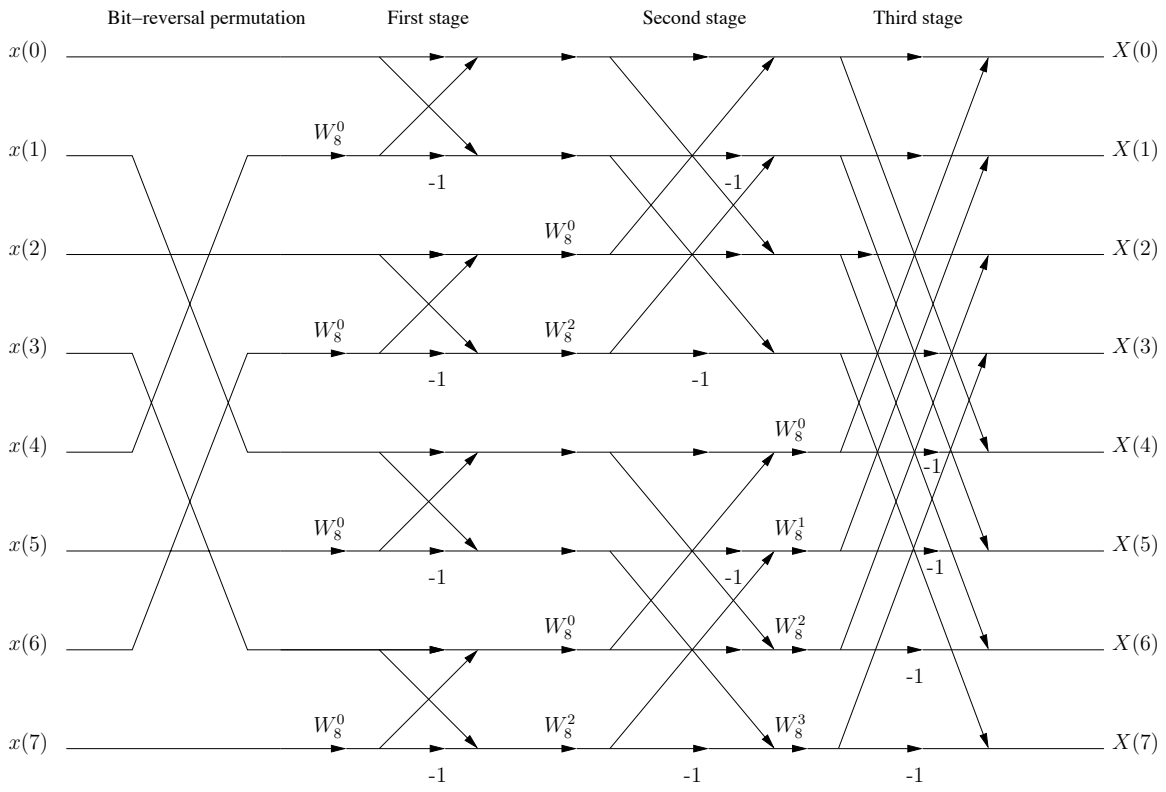


**Figure 2.3:** An 8-point FFT butterfly, as demonstrated by Chen *et al.* [15]. Such a task can be executed on a thread independently in 3 stages butterfly operations *i.e.* without communication or data exchange with other tasks.

widely studied over the years. One of which, FFTW, is still considered among the most efficient on multi-core and many-core systems. However, most of these parallel implementations still take advantage of coarse-grain synchronization between cores. There are only a few FFT studies driven by fine-grain execution models to achieve better balanced workload on computation.

Chen *et al.* [15] demonstrated the implementation of 1D and 2D FFT algorithm on Cyclops-64. Their method implemented the classical Cooley-Tukey algorithm in an iterative instead of recursive way. The cores are synchronized by using the hardware barrier provided by Cyclops-64 at the end of multiple stages which depends on the size of butterfly computation *i.e.* task size. They only used SRAM and the register files

in FFT computation. As shown in Figure 2.3, 2-point butterfly in original Cooley-Tukey algorithm was extended to 8-point butterfly. There is additional step *i.e.* bit reversal permutation before all butterfly operations. It is used to change the input data positions to guarantee the output data in correct positions. If we don't apply such a step before the butterfly computation, we can also involve it at end of the butterfly computation to achieve the same effect. Such a step must be executed once and only once in FFT implementation. They divided the whole computation into many small parallel tasks which load the data the pre-computed twiddle factors from SRAM to registers, apply butterfly computation, and store the intermediate or final results back to SRAM in place [15]. Figure 2.3 shows a task whose input is 8 data points. Accord to the feature of FFT algorithm, such a task can be executed on a thread independently in 3 stages butterfly operations *i.e.* without communication or data exchange with other tasks. We can see that as the task size increases the data exchange between SRAM and registers reduces. It keeps cores busy for computation for a longer time. Furthermore, it also makes the whole computation need less barriers. However, too large task size hurts the parallelism and can not be implemented in their work because the number of register on each core is limited on Cyclops-64. They stored all the data are stored in the on-chip SRAM memory and found that when the task size is set as an 8-point butterfly, the performance is the best.

However, because of the limited size, on-chip memory can not be fitted in for a large input data size. Hence, we need to extend on-chip to off-chip memory data storage and use scratchpad instead of registers for intermediate data. In this case, we found that 64-point FFT performs better than the 8-point FFT due to the reduction of off-chip memory accesses.

## 2.4 The Codelet Model

Our work is based on the codelet program execution model [75, 34]. The feature of the codelet model provides us a chance to improve the performance and reduce energy computation during parallel execution.

The codelet model takes both advantages of dataflow and Von Neumann models [25]. It is a hierarchical fine-grain multithreading model that are based on the concepts and semantics of codelets. According to the parallelism and data dependencies, the programs are divided and capsulated into codelets *i.e.* a collection of instructions. The codelet is the finest granularity of parallelism in the model. Inside a codelet, the instructions are executed sequentially. A codelet run asynchronously without blocking until it completes. Usually, a codelet runtime is involved to schedule each codelet to the computation unit by the scheduling rules. Since codelets have a larger granularity than instructions, it provides an opportunity to reduce runtime overhead by tuning the size of codelet. Codelet execution is event-driven which means only when the required data and resources are available, the codelets receive all synchronized signal and can start execution.

Each codelet can have one of the following 4 status:

- Dormant: Not yet receive all required data.
- Enabled: Receive all the data required to execute the codelet.
- Ready: Enabled and all resources needed to execute the codelet are ready.
- Firing: Execution of a ready codelet when it is scheduled on a processing unit.

A codelet changes among the 4 status according to availability of data and computation resources.

According to data dependencies, the codelets in a program are connected together into a graph called the codelet graph (CDG) which has its root in dataflow graph [25]. If a codelet graph is well-behaved *e.g.* no deadlock caused by a cycle in the codelet graph, the execution will be determinate *i.e.* the outputs always are the same for a given input to the codelet graph. However, the execution order of codelets in each run is not guaranteed to be the same, because the occurrence of the events to trigger the codelets highly depends on runtime and hardware system. Our work in fine-grain FFT algorithm makes use of the changeable execution order of codelets to improve the memory bandwidth usage.

In general, the whole codelet graph is not required to statically construct. It is started by setting up a part of codelets and their dependencies statically and then the other subgraphs can be created dynamically by the running codelets. Our fine-grain FFT algorithms produce the codelet graph at the beginning of the execution when the input data size is known. Our work in locality exploitation in the codelet model also takes advantage of a purely static codelet graph.

The codelet model can be mapped to corresponding abstract parallel computer hardware and software system called abstract machine. As shown in Figure 2.4, the codelet abstract machine is organized hierarchically with heterogeneous elements. It consists of many nodes which are linked together by an interconnection network. Each node has one or more many-core chips and shared node memory. Each chip contains many clusters connected together by a chip interconnect and shared chip memory. Each cluster contains a collection of computing units (CU) and one or more synchronization and scheduling units (SU) which are linked together by an on-chip interconnect and share cluster memory. Computing units, which can be any type of core, are in charge of computation *i.e.* executing codelets. Synchronization units are responsible for scheduling the codelets to available computing units based on the scheduling rules to achieve performance and energy saving. Synchronization units also handle exceptions, hardware interruption, memory request from or to out-of-cluster location, etc. Each computing unit or synchronization unit has its own local memory. The hierarchical feature of the codelet abstract machine can improve locality in programs by organizing the codelets into specific levels of the machine.

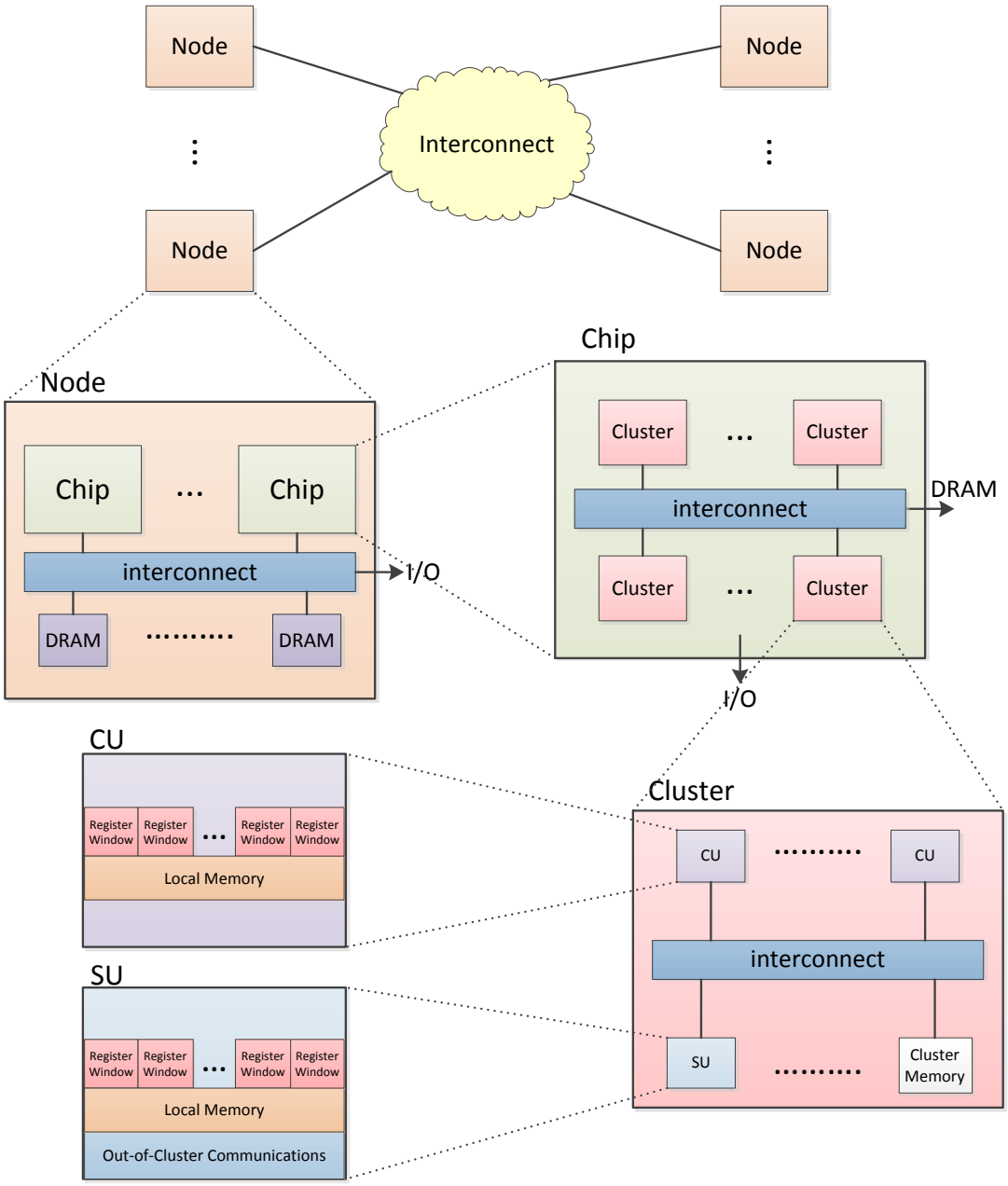


**Figure 2.4:** Abstract machine for the codelet model. It is organized hierarchically with heterogeneous elements. At cluster level, computing units (CU) execute ready codelets and synchronization units (SU) schedule the codelets to available computing units. There is share memory at each level to achieve the locality.

# Chapter 3

# MEMORY WORKLOAD BALANCE IN CODELET PXM

In this chapter, we demonstrate that the fine-grain execution models can improve more balanced workload on not only computation but also memory bandwidth usage than the coarse-grain execution models can. We implement FFT algorithm on the IBM Cyclops-64 many-core architecture to show the advantage of the fine-grain execution model by using codelet model. The related work is also provided.

## 3.1 Methodology

This section introduces and analyzes the methodology of our work. Section 3.1.1 uses an example to demonstrate the opportunity provided by the codelet model to improve the memory bandwidth usage. Section 3.1.2 describes coarse-grain, fine-grain, and guided fine-grain FFT algorithms with 64-point task size. Section 3.1.3 presents our way to randomize the off-chip memory addresses in order to achieve balanced memory bandwidth usage.

### 3.1.1 Motivating Example

We have described the coarse-grain FFT algorithm on Cyclops-64 in Section 2.3. This algorithm shows very good performance when using on-chip memory [15]. However, when we extend this algorithm to using off-chip memory to store the data and twiddle factors, it does not work well. We found that the problem is caused by the unbalanced memory accesses to the off-chip memory banks.

As described in Section 2.2, there are only 4 ports connecting 4 off-chip memory banks respectively on a Cyclops-64 node. From Figure 3.1, we can see that the memory accesses only in the last few stages (about last 1/3 of the execution time) are evenly

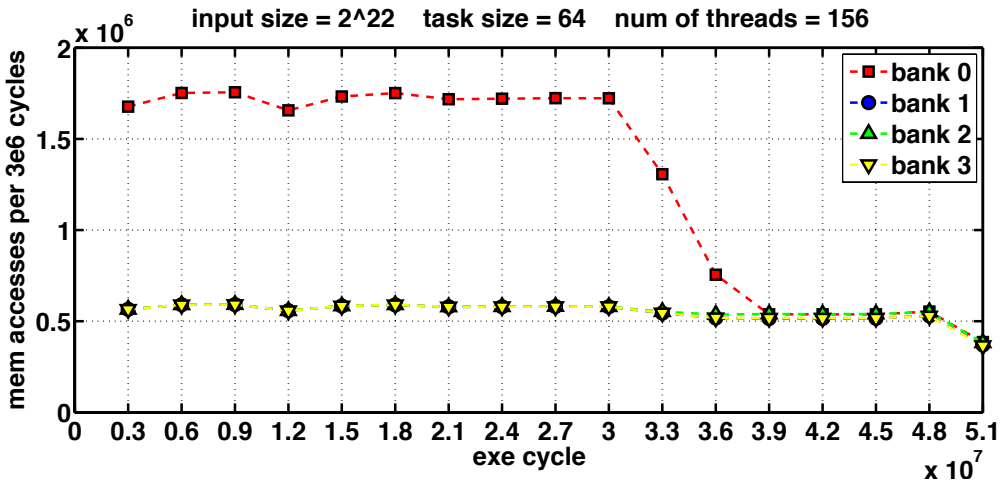


**Figure 3.1:** Access rates of the 4 off-chip memory banks in the coarse-grain FFT algorithm. In early stages, *Bank 0* is accessed three times more than the other banks. It causes contention on *Bank 0* and wastes the bandwidth resource on other banks.

distributed to the four memory banks in the coarse-grain FFT algorithm. In early stages (about first 2/3 of the execution time), *Bank 0* is accessed three times more than the other banks. It causes contention on *Bank 0* and wastes the bandwidth resource on other banks. Because the accessed elements in the twiddle factor array in early stages have a stride that is a multiple of 64 in the memory address and the four off-chip memory banks are interleaved on 64-byte boundaries as well (In a Cyclops-64 node, the first 64 bytes of data are stored on *bank 0*), the next 64 bytes on *bank 1*, and so on so forth.), the accesses on the twiddle factor array in early stages always go to *bank 0*. However, because the stride of the accessed addresses in the last few stages is less than 64, the access rates of the four memory banks are more balanced.

In fine-grain execution models, it is possible for a task in a later stage to be executed prior to a task in an early stage. Therefore, the execution order of the tasks can be guided to get more balanced workload on the memory banks. Some tasks in early stages which only need the data on *bank 0* can be delayed execution in order to relieve the burden on *bank 0*. On the other hand, some tasks in late stages which

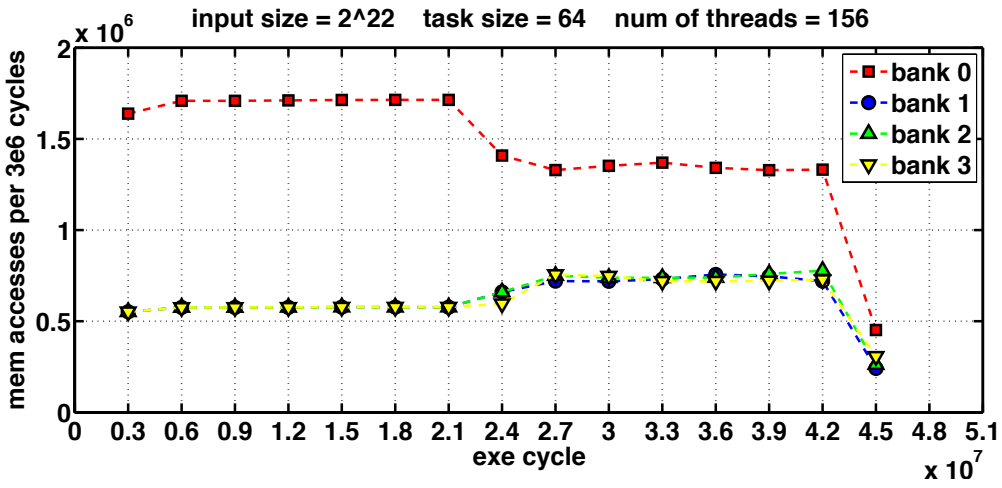


**Figure 3.2:** Access rates of the 4 off-chip memory banks in the fine-grain FFT algorithm. Compared to Figure 3.1, the memory access pattern is much more balanced starting from the second half of the execution time. Contention reduces on *Bank 0* and the bandwidth resource on other banks can be made use of earlier.

need the data on *bank 1, 2, 3* can be advanced to execute so that the bandwidth of *bank 1, 2, 3* can be made use of when free. Figure 3.2 demonstrates the access pattern of the off-chip memory banks in our designed fine-grain FFT algorithm. We can see that the access pattern is more balanced by the access rate of *bank 0*/*bank 1, 2, 3* decreasing/increasing earlier (at about 1/2 of the execution time).

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 method of randomizing the memory addresses introduces big overhead. On the other hand, the feature of randomizing the memory addresses by hardware is not supported by all architectures, because such a feature breaks locality for regular applications. For comparison, we also implement FFT algorithm with hashed twiddle factor array.

### 3.1.2 Three FFT Implementations

#### Coarse-Grain Algorithm

**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);
    end for
  else
    for t_id = 0 to N/64 − 1 in parallel do
      FFT_last_stage_kernel(D,W,stage,t_id);
    end for
  end if
  barrier;
end for
```

The 64-point FFT coarse-grain algorithm can be viewed as an extension of Chen *et al.*'s 8-point FFT algorithm that is described in Section 2.3.

The pseudo code of the coarse-grain 64-point FFT algorithm is shown in Algorithm 1. As other Cooley-Tukey based FFT algorithms, the parallel bit-reversal process before butterfly computations is required to change the input data position to guarantee the output data in correct positions. Then, the whole FFT computation is divided into $\lceil log_2 N/6 \rceil$ stages, where $N$ is the number of the input data. Without loss of generality, we assume that $N$ is a power of two because in FFT algorithm we always append amount of zero data to make the input data size as a power of two. 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 from the off-chip memory to scratchpad, applies butterfly computation on 6 levels (the intermediate data are stored in scratchpad), and stores 64 computed data back to the off-chip memory in place. At the end of each stage, all the threads are synchronized by 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 $log_2 N$ mod 6 levels of butterfly computation.

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

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

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

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

**Fine-Grain Algorithm**

We can see that each task only needs 64 data and 63 pre-computed twiddle factors as input in the coarse-grain FFT algorithm,. The 64 inputs are generated by the output of the 64 parent tasks in the previous stage. For example, a task in *stage 1* can execute once its 64 parent tasks in *stage 0* have completed. It provides an chance to remove the barriers in the coarse-grain FFT algorithm. Therefore, we propose the fine-grain FFT algorithm as shown in Algorithm 2.

The codelet model is used to represent the fine-grain FFT algorithm. Each task in the coarse-grain FFT algorithm is a codelet in the fine-grain FFT algorithm. If the total number of codelets are much more than the total number of threads, the computation workload is balanced. A counter is associated with a codelet instead of using a barrier to achieve the synchronization by updating the number of satisfied dependencies. A concurrent codelet pool is used to store all the codelets that are ready to be executed *i.e.* all the dependencies are satisfied. Initially, all the codelets in *stage 0* are put in the pool since their input data are already available. 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 free thread then takes the next codelet from the codelet pool. Once all the threads finish their work and there are no more codelets in the pool, the whole computation completes.

**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)};
end for
for each element e in cnt do
  e←0;
end for
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);
      end if
    end for
  else
    FFT_last_stage_kernel(D,W,stage,t_id);
  end if
end while
```

Assume the parent codelet be the $i$th codelet in stage $j$, and its $k$th child be the $l$th codelet in stage $j+1$, then

$$l = \lfloor \frac{i}{64^{j+1}} \rfloor \times 64^{j+1} + i \bmod 64^{j+1} \bmod 64^{j} + k \times 64^{j} \tag{3.1}$$

In this algorithm, every 64 children codelets share the same 64 parent codelets. 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 i.e.* $i$ 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 share a synchronization counter.

**Guided Fine-Grain Algorithm**

According to the property of the codelet execution model, the fine-grain FFT algorithm is determinate (see Section 2.4). However, the execution order of the codelets is not guaranteed in different runs. Both the initial arrangement of the codelets in the codelet pool and the execution time of the codelets in runtime influence the execution order. A good execution order may achieve more balanced memory accesses to the off-chip memory banks. From the observations, the codelets in the early stage has heavy memory contention on *bank 0* and the codelets in the last few stages (especially the last stage) have a balanced workload of memory bandwidth. Therefore, we should break the stage order to make the late-stage codelets executed as early as possible. However, due to the data dependency, children codelets can only be executed after their parent and ancestor codelets complete.

We design a guided fine-grain FFT algorithm to guide the execution order to improve the memory bandwidth usage. Firstly, we divide 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. In the first step, the codelets in the *early stages* are executed. Then a barrier is used to ensure the completion of all codelets in *early stages*. In the second step, a last-in-first-out (LIFO) codelet pool is used to store the codelets of stage $i + 1$ to arrange a proper order that make the codelets in the last stage satisfy their dependencies as soon as possible. In such a way, the codelets in the last stage are more likely to be executed earlier. As a result, more balanced workload on the 4 DRAM banks are achieved. Algorithm 3 shows our guided fine-grain FFT algorithm using the last two stages as the *late stage*.

### 3.1.3 Randomization on DRAM Bank Accesses by Hashing

An alternative solution to balance the memory workload is to randomize the memory addresses of the elements in the twiddle factor array $W$. Note that the data are always accessed in a balanced pattern on the 4 banks by codelets in each level. 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 $i$th 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:

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

Figure 3.3 shows the access rates of the 4 off-chip memory banks by randomizing of the twiddle factor addresses using the bit reversal function. We can see that the memory accesses on the 4 memory banks are balanced. However, because of the overhead of the hash function, the memory address randomization method doesn't always achieve better performance. The detailed experimental results will be shown in Section 3.2.

## 3.2 Experiment

In this section, we provide and analyze the experimental results of different implementations of FFT algorithm which are described in Section 3.1 .

### 3.2.1 Task Size and Theoretical Peak Performance

In this section, we calculate the theoretical peak performance of the FFT algorithm on the Cyclops-64 node. We assume that the input data size is so large that both

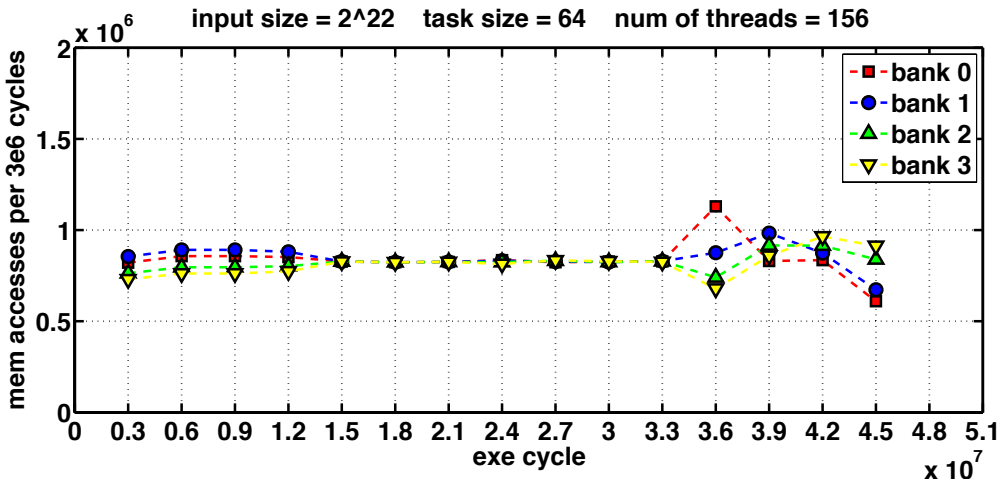


**Figure 3.3:** Access rates of the 4 off-chip memory banks in the fine-grain FFT algorithm with randomized twiddle factor addresses. By using the hash function, all banks are accessed in a uniform manner. The bandwidth resource on each bank can be fully used.

the data array and twiddle factor array are stored in the off-chip memory. The task or codelet size affects the theoretical peak performance because larger size leads to less amount of off-chip memory accesses. We choose the task or codelet size as 64 point butterfly computation, since too much on-chip memory is required and the scratchpad limit is exceeded for larger task or codelet size.

Figure 3.4 shows the best execution time of the fine-grain FFT algorithm using 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:

$$\begin{aligned} peak &= \frac{\#\ of\ floating\ point\ operations}{theoretical\ exectime} \\ &= \frac{5 \times N \times log_2 N}{exectime\ per\ task \times \#\ of\ tasks} \end{aligned} \tag{3.2}$$

$$\#\ of\ tasks = \frac{N}{64} \times \lceil \frac{log_2 N}{log_2 64} \rceil \tag{3.3}$$

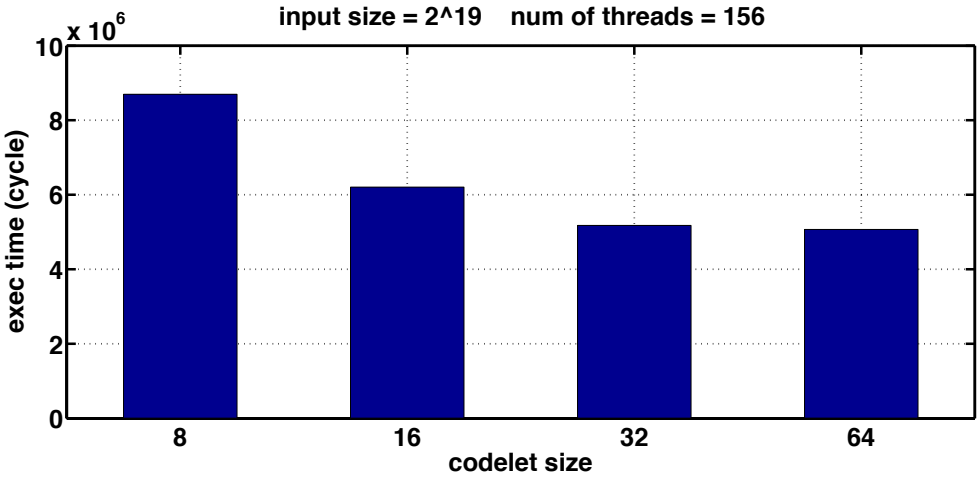


**Figure 3.4:** The best execution time of the fine-grain FFT algorithm for 156 threads units running in parallel and a global input data set of $2^{19}$. 64-point FFT codelets perform best. The $Y$ axis shows the execution time in cycles. The $X$ axis shows the number of data points given as input to each codelet.

$$execTime\ per\ task = \frac{(64 + 64 + 63) \times 16Bytes}{DRAM\ bandwidth} \tag{3.4}$$

where $N$ is the data size. To simplify the computation, we remove the ceiling function in Equation 3.3. The removal does not reduce the theoretical peak performance because it will decrease the denominator in Equation 3.2. Equation 3.4 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. If the off-chip memory contention does not happen, we get the best execution time of a task as shown in Equation 3.4. The DRAM bandwidth on Cyclops-64 is 16 GB/sec as shown in [37]. So we get the following theoretical peak performance from Equations 3.2, 3.3, and 3.4.

$$\begin{aligned} peak &= \frac{5 \times N \times log_2 N \times 64 \times 6 \times 16G}{N \times log_2 N \times (64 + 64 + 63) \times 16} \\ &= 10\,GFLOPS \end{aligned} \tag{3.5}$$

As shown in Equation 3.5, the theoretical peak performance of the FFT algorithm on Cyclops-64 is 10 GFLOPS when the data array and twiddle factor array are stored in the off-chip memory.

### 3.2.2 Experimental Setup

We implement FFT on the FAST simulator [15] which is a functionally-accurate simulator. It models the memory hierarchy of the Cyclops-64 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 3.1: `coarse`, `coarse hash`, `fine` (divided between `fine worst` and `fine best`), `fine hash` and

| Name | Description |
|---|---|
| `coarse` | Coarse-grain synchronization |
| `coarse hash` | Coarse-grain synchronization with hashed twiddle factor array |
| `fine worst` | Worst execution time for fine-grain synchronization |
| `fine best` | Best execution time for fine-grain synchronization |
| `fine hash` | Fine-grain synchronization with hashed twiddle factor array |
| `fine guided` | Guided fine-grain synchronization |

**Table 3.1:** Description of the various methods used to perform FFT on Cyclops-64. `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.

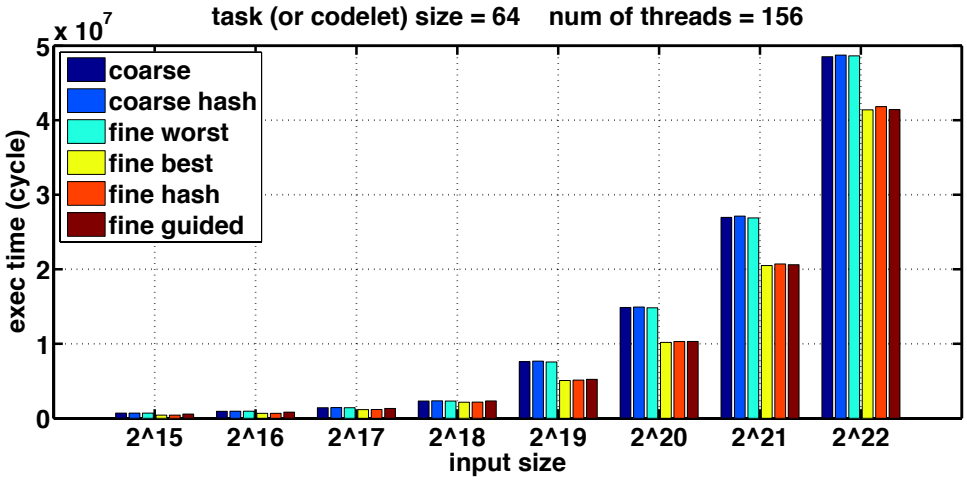


**Figure 3.5:** The execution time of 5 versions of FFT algorithms on Cyclops-64. The $X$ axis shows the various input sizes of FFT. The $Y$ axis shows the execution time. Lower is better.

`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 Figure 3.5 and 3.6 as `fine worst` and `fine best`, respectively.

### 3.2.3 Major Observations

From our experimental results, the major observations are shown as below:

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 experimental results and analysis are described in the following sections.

### 3.2.4 Performance of the Various FFT Algorithms

Figure 3.5 shows the performance of the 5 versions when the input data size varies from $2^{15}$ to $2^{22}$. From Figure 3.5, we see that:

1. `fine guided` always performs between `fine best` and `fine worst` and close to `fine best`. The reason is that `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`. The reason is that 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.

### 3.2.5 Scalability and Speedup

Figure 3.6 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 3.1. `coarse` suffers from memory contention on *bank 0* in the early stages of the computation. `coarse hash`, however, suffers from the

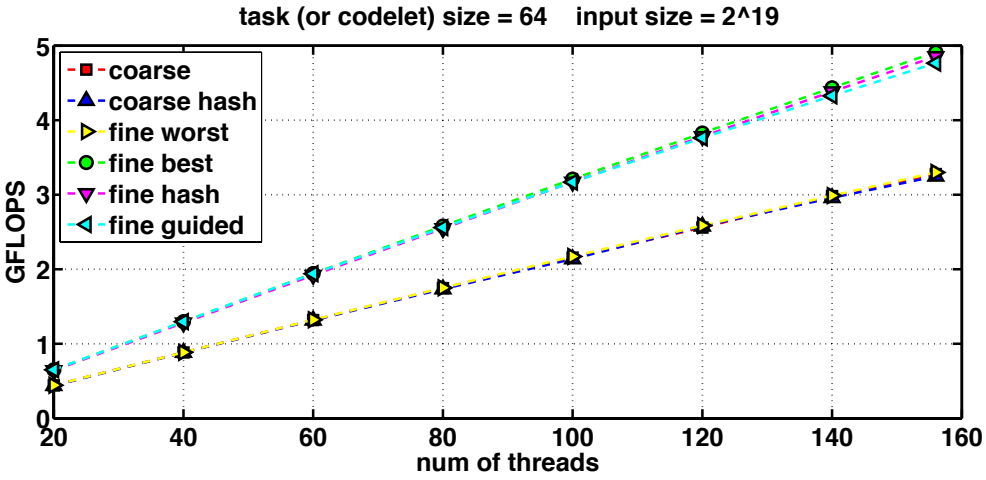


**Figure 3.6:** Performance of 5 versions of FFT algorithms on Cyclops-64 for an input size of $2^{19}$ data elements and 64-point butterfly codelets. The $X$ axis represents the number of thread units used in the computation. The $Y$ axis features the resulting performance (in GFLOPS). Higher is better.

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.

### 3.3 Related Work

Sequential and parallel implementations of FFT have been widely studied in the past. FFTW [31, 30, 32] is one of the most famous work on FFT implementation. In FFTW, a planner is used to efficiently divide the whole one- or multi- demensional FFT computation into codelets. Although the name is the same, the concept of the codelet in FFTW is different from that is described in section 2.4. The codelets in FFTW are only some straight-line code that performs a part of the whole FFT computation. However, the codelets in the codelet model is based on dataflow and involve relationship

represented by data dependencies among them to synchronize the execution. Based on FFTW, UHFFT [55, 3], and PFFT [62] are developed to make the FFT computation work efficiently on multi- and many- core systems.

Choi *et al.* described the parallel implementation of FFT algorithm on a specific multi-core system which includes an array of SIMD processing elements [17] to achieve better performance and energy consumption. Takahashi demonstrated an implementation of the block two-dimensional FFT algorithm using vectorization by SIMD instructions to reduce cache miss on general purpose multi-core processors [70]. Franchetti *et al.* presented a multi-core extension of Spiral program generator to achieve workload balance and avoid false sharing [29].

FFT algorithm have also been studied in Network-on-Chips (NoC) environment. Bahn *et al.* presented parallel FFT implementations to maximize the data parallelism and minimize the communication overhead by exploiting data locality in NoC [17]. Therefore, their algorithms improve workload balance. Mattson *et al.* implemented a two-dimensional FFT kernel on Intel 80-core NoC Terascale Processor based on Pease FFT algorithm [54].

The research of FFT computation on GPU is performed extensively. Moreland *et al.* demonstrated an early study of FFT implementation for GPU before Cuda and OpenCL emerge [56]. They used index magic technique *i.e.* bit reversal permutation as described in Section 2.3 to reduce amount of data copying and frequency compression technique for real input data to improve the computation efficiency. Volkov *et al.* presented an implementation of FFT algorithm on G80 architecture. The hardware resources *e.g.* large register files on the GPU and small on-chip local memory are fully used to achieve good performance. Govindaraju and Lloyd implement hierarchical mixed radix FFT algorithms on GPU [53, 41]. In order to efficiently use memory bandwidth, their algorithms are based on a radix-2 Stockham formulation of FFT to avoid bit-reversal process. Garland *et al.* used Cuda programming model to implement parallel algorithms including FFT on NVIDIA GPU [39]. Dotsenko *et al.* designed an auto-tuning FFT kernel to accommodate different input parameters by also using Cuda

on NVIDIA GPU [26].

The implementation of FFT algorithm on IBM Cell Broad Engine have been studied in [5, 7, 18, 52]. The memory hierarchy of IBM Cell Broad Engine is very similar with that of Cyclops-64. Therefore, both need the data transfer between global and local memory. Bader *et al.* implemented FFT iteratively on IBM Cell Broad Engine by applying barriers using chain-like inter-SPE communication at every stage. The input data are stored in the main memory and fetched into local store of each SPE for computation by DMA transfer in a non-blocking manner. Therefore, memory latency can be hidden by computation by applying double-buffering [5].

FFT algorithm have also been ported on FPGA systems [8, 46]. Kamalizad *et al.* mapped the FFT computation onto M2 efficiently by making use of its unique hardware features [46].

Following their work on UHFFT, Franchetti *et al.* gave an overview of implementing FFT algorithm on various multi- and many-core architectures, including IBM Cell, GPUs, FPGAs, and Intel-compatible multi-cores [28]. They also provided the approaches to optimize the performance for each architecture.

All the works described above map the computation units to hardware statically and use coarse-grain synchronization by barriers to ensure the data integrity between stages. Therefore, the workload tends to be imbalance among the computation elements. On the other hand, our approach takes advantage of the fine-grain codelet model to execute asynchronously regarding to data dependencies among tasks *i.e.* 64-point butterfly computations as well as hardware availability. Besides, our implementation is based on dynamic scheduling at run-time which makes the utilization of computation resource more efficiently as well. Other three works which are very related to our study are introduced as below.

Saybasili *et al.* gave a solution to implement FFT based on the radix-2 Cooley-Tukey algorithm on fine-grain XMT architecture [66]. The fine-grain concept in their work is different from that in our approach. Their fine-grain method is based on the XMT support for short threads *i.e.* not operating system threads. On the other hand,

our fine-grain approach relies on the codelet model that removes barriers by applying event-driven synchronization mechanism. In their algorithm, two-dimensional FFT is implemented by applying parallel one-dimensional FFT on each dimension. However, their parallel one-dimensional FFT computation still uses coarse-grain synchronization by applying barriers at the end of every stage. Besides, the XMT architecture supports hardware randomization on whole memory address space while our work use software hashing on the twiddle factor array. Therefore, the memory workload on XMT is balanced. Note that the FPGA prototype of XMT only supports fixed-point arithmetic and uses on-chip memory.

Thulasiraman *et al.* designed and compared two fine-grain approaches to implement FFT algorithm based on the EARTH model [71] which also finds its roots in the classical dataflow model and can be viewed as the ancestor of the codelet model [75]. The major difference of the two approaches: Receive-Initiated and Sender-Initiated in [71] 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. Furthermore, our work not only guarantees good workload-balancing, but also efficient memory access balancing.

Finally, Chen *et al.* effort on optimizing FFT on Cyclops-64 has already been discussed in Section 2.3. In order to compute FFT for large input data size, our work extended their work from calculating 8-point to 64-point FFT kernel using off-chip instead of on-chip memory for data storage. We also use scratchpad instead of registers for intermediate data. Furthermore, our work uses fine-grain synchronization while their work relies on coarse-grain synchronization.

### 3.4 Discussion

In this thesis, we show that by achieving more balanced workload on computation as well as memory bandwidth the fine-grain codelet execution model provides better performance than the coarse-grain approach does. In fact, the codelet model

is also able to improve the energy efficiency. The overall energy consumption includes static and dynamic parts. The static energy consumption is decided by the performance *i.e.* the execution time. Therefore, the codelet model consumes less static energy than the coarse-grain model. The experiment is based on the Cyclops-64 FAST simulator which can only give the program execution time. However, we can not compare the dynamic energy consumption which depends on the number and type of the executed instructions by this simulator.

Fortunately, we have fsim simulation platform which simulates the Intel straw-man architecture [48] in the UHPC project. Although this simulator is not cycle-accurate, it can provide the dynamic energy consumption of a program. In future, we will implement FFT using the fine-grain codelet and coarse-grain model on fsim and prove that the codelet model also achieves better dynamic energy efficiency. According to the hardware feature, there may be other optimization can be made in the Intel straw-man architecture based on the codelet model.

**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 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;
last_early_stage←last_stage − 2;
for t_id = 0 to N/64 − 1 do Q ← Q ∪ {(0, t_id)}; end for
for each element e in cnt do e←0; end for
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);
      end if
    end for
  end if
end while
barrier;
for every 64 codelets t_id0,...t_id63 in (last_stage − 1) that have the same child codelets do
  Q ← Q ∪ {(last_stage − 1, t_id0), ..., (last_stage − 1, t_id63)};
end for
for each element e in cnt do e←0; end for
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);
      end if
    end for
  else
    FFT_last_stage_kernel(D,W,stage,t_id);
  end if
end while
```

# Chapter 4

# LOCALITY EXPLOITATION IN CODELET PXM

In this chapter, we introduce the locality exploitation problem and our solution in the codelet model. Three different algorithms are provided to automatically exploit locality in the codelet model. On an emulation platform of Cyclops-64 architecture, we apply and compare these three algorithms on various applications including matrix multiply, merge sort, and random generated codelet graphs with reasonable assumptions. Our approach provides maximum locality exploitation as well as maximum parallelism. The related work is also provided.

## 4.1 Methodology

In this section, we present and analyzes the locality exploitation problem in the codelet model and the methodology of our work. Section 4.1.1 gives an example to motivate the locality exploitation in the codelet model. In Section 4.1.2, the locality exploitation problem is formalized as the Best Scheduling Problem. Section 4.1.3 provides our approach to solving the Best Scheduling Problem.

### 4.1.1 Motivating Example

Figure 4.1 gives a simple example to illustrate the locality exploitation in the codelet model. There are 6 codelets which are connected based on their data dependencies. The starting and ending codelets which are used for initialization and cleanup respectively do not affect the locality exploitation. The 4 codelets *i.e.* $A$, $B$, $C$ and $D$ are working codelets. The edges from $A$ and $B$ to $C$ and $D$ indicate data dependencies between the codelets. According to the fire rule in the codelet model, a codelet can be executed when the data dependencies are satisfied. For example, $C$ is able to run

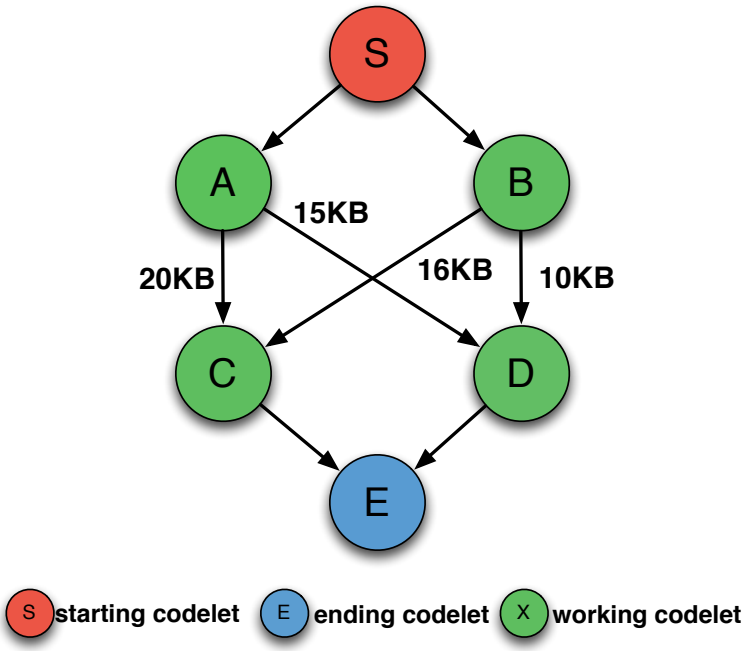


**Figure 4.1:** A motivating example of locality exploitation in the codelet model. The weight on an edge represents the amount of locality that can be exploited by scheduling the two ends of the edge on the same core. The best plan exploits 31KB locality by scheduling $AD$ to one core and $BC$ to another.

after the completion of both $A$ and $B$. The weight on each edge provides the amount of data *i.e.* locality information produced by the upstream codelet and consumed by the downstream one. For example, the edge between $A$ and $C$ means that $A$ generates 20KB data for $C$ to use. In general, $A$ has to store this 20KB data into the shared memory, since it guarantees that $C$ is able to access the data no matter where $C$ is executed. However, if $A$ and $C$ are scheduled to the same core, $A$ does not need to store the 20KB data into the shared memory. Instead, $A$ is able to store the data into the local memory of the core for future access of $C$. In such a way, we exploit the locality between $A$ and $C$.

Figure 4.2 shows a simplified codelet program written in SWARM [49]. The program creates the codelet graph in Figure 4.1 without the starting and ending codelets. In the program, line 1 and 2 set up the dependencies for $C$ and $D$ that have two parents

```
/*Define # of parents for codelets C & D*/
1:  dep_t depC=swarm_Dep_INITIALIZER(2,&C,...);
2:  dep_t depD=swarm_Dep_INITIALIZER(2,&D,...);

3:  CODELET_IMPL_BEGIN(A) //Begin of codelet A
4:      ... //A's work;
        /*Satisfy A's dependencies for C & D*/
5:      swarm_Dep_satisfyOnce(&depC);
6:      swarm_Dep_satisfyOnce(&depD);
7:  CODELET_IMPL_END; //End of codelet A

8:  CODELET_IMPL_BEGIN(B) //Begin of codelet B
9:      ... //B's work;
        /*Satisfy B's dependencies for C & D*/
10:     swarm_Dep_satisfyOnce(&depC);
11:     swarm_Dep_satisfyOnce(&depD);
12: CODELET_IMPL_END; //End of codelet B

13: CODELET_IMPL_BEGIN(C) //Begin of codelet C
14:     ... //C's work;
15: CODELET_IMPL_END; //End of codelet C

16: CODELET_IMPL_BEGIN(D) //Begin of codelet D
17:     ... //D's work;
18: CODELET_IMPL_END; //End of codelet D
```

**Figure 4.2:** A simplified SWARM codelet program corresponding to the codelet graph in Figure 4.1. Note that starting and ending codelets are omitted. The SWARM runtime handles the codelet graph creation (line 1 and 2) and dependency satisfaction (line 5,6,10,and 11).

respectively. Line 5 and 6 satisfy the dependencies of $A$'s children *i.e.* $C$ and $D$ as soon as $A$ completes. Similarly, line 10 and 11 satisfy the dependencies of $B$'s children *i.e.* $C$ and $D$, when $B$ finishes. According to the program, SWARM runtime synchronizes codelets by handling the dependencies, and schedules the ready codelet to the free core dynamically.

Normally, we are able to make use of locality in different ways. For example, if there are two cores, $AC$ can be scheduled on one core and $BD$ on the other. In this case,

30KB locality in total is exploited. However, the best scheduling plan for this example is to schedule $AD$ on one core and $BC$ on the other. Therefore, we can exploit 31KB locality. In a complicated codelet graph involving many codelets and dependencies, exponential amount of scheduling plans may exist. In general, it is impossible for a programmer to manually design a schedule plan to guarantee exploiting maximum locality. We will present and analyze three algorithms which automatically generate an optimal or nearly optimal schedule plan for locality exploitation in Section 4.2.

### 4.1.2 Problem Statement

In this section, we formalize the problem of locality exploitation as the Best Scheduling Problem. We assume that the codelet graph is created statically and the locality information among the codelets is known. All the codelets in the static graph are divided into several groups with respect to the locality information. A group of codelets must schedule to the same core. Note that the execution of the codelets in the same core is must to be ordered by the data dependencies among them. Hence, the static scheduling method must be used in the runtime system based on the generated schedule plan which aims to exploit the maximum amount of locality. In order to achieve locality, the data generated by a codelet (only the part of data which can be reused by its downstream codelet) is temporary stored in the local memory of the same core for the downstream codelet. After consuming the data in local memory, the codelet can vacuum the local memory space for saving data which can be used by its downstream codelet. In this way, both the latency of the memory access and energy consumption are reduced because local memory access spends much less time and energy than global memory access. Therefore, the Best Scheduling Problem can be defined as below:

*(Best Scheduling Problem)* Given a weighted codelet graph $G =< V, E, W >$ and a positive integer $n$, where $V$ represents the codelets, $E$ represents the dependencies, $W$ represents the potential locality, and $n$ represents the total number of cores, find a mapping

$$f : V \rightarrow \{1, \ldots, n\}$$

to satisfy the following requirement:

$$\begin{aligned} Maximize &: \{\sum W(v_1, v_2) | f(v_1) = f(v_2) \wedge v_1 \leftrightarrow v_2\} \\ Subject\ to &: \forall f(v_1) = f(v_2), v_1 \xrightarrow{P} v_2 \vee v_2 \xrightarrow{P} v_1 \end{aligned}$$

,where $v_1 \leftrightarrow v_2$ means that $v_1$ and $v_2$ are adjacent (executed one after the other) in the same group, and $v_1 \xrightarrow{P} v_2$ means that there exists a path in $G$ from $v_1$ to $v_2$.

### 4.1.3 Solution

We propose three algorithms to solve the Best Scheduling Problem. The three algorithms have different trade-off in the algorithmic complexity, locality exploitation, program performance, energy efficiency, and required computation resources. The features of the three algorithms are as follows:

- **Min-cost flow based algorithm:** This algorithm converts the Best Scheduling Problem to a min-cost flow problem. It guarantees an optimal solution. The time complexity is $O(knmlog(n))$ where $k$ is the number of cores, $n$ is the number of codelets, and $m$ is the number of dependencies in the codelet graph.
- **Max first algorithm:** This is a heuristic algorithm that provides a nearly optimal solution in practice. Its time complexity is $O(nlog(n) + m)$ which is the lowest in the three algorithms.
- **Graph partitioning based algorithm:** This algorithm converts the Best Scheduling Problem to a graph partitioning algorithm. The time complexity is $O(mlog(k))$ which is lower than the min-cost flow based algorithm.

We will discuss the three algorithms in detail in Section 4.2.

## 4.2 Algorithm

In this section, three algorithms used to exploit locality automatically in the codelet model are explained in Section 4.2.1, 4.2.2, and 4.2.3, respectively.

### 4.2.1 Min-cost Flow Based Algorithm

The Best Scheduling Problem can be converted to a min-cost flow problem. Given the weighted codelet graph, a flow network can be created such that: (1) Each scheduling plan can be viewed as a flow in the flow network; and (2) The sum of the available weights in a scheduling plan associates with the cost of the corresponding flow in a anticorrelated manner. Therefore, a min-cost flow algorithm can be used to find the corresponding scheduling plan which has the maximum sum of available weights *i.e.* maximum locality exploitation among all possible scheduling plans.

Algorithm 4 demonstrates how to convert a given weighted codelet graph $G$ into a flow network $N$ and how to map the solution of the min-cost flow problem to the solution of the Best Scheduling Problem. The steps are shown as follows:

In the algorithm, two super vertices $src_1$ and $src_2$ are created in the network. Then create an edge from $src_1$ to $src_2$. The capacity of the edge is set as the total amount of computation resource *i.e.* the number of cores, which indicates that the network cannot over consume the computation resource. The cost of the edge is 0.

For each vertex $v$ in the codelet graph, create two corresponding vertices in the network as $v_1$ and $v_2$. Then create an edge from $v_1$ to $v_2$. The capacity of the edge is set as 1, which guarantees that each codelet can be scheduled to only one processor to be executed only once. The cost of the edge is set to $-\infty$, which guarantees that the codelet $v$ must be executed since min-cost flow must go through such a low cost edge. In practice, we use a large negative number $-M$ to replace $-\infty$. It does not affect the optimal solution.

For each edge $(u, v)$ in the codelet graph, create an edge $(u_2, v_1)$ in the network. The capacity of the edge will be set as 1, which indicates that the reuse data between codelet $u$ and $v$ will happen at most once. The cost of the edge is set as a negative number $-w(u, v)$, where $w(u, v)$ is the weight of the edge $(u, v)$ in the codelet graph. It means that the solution actually gains $w(u, v)$ data reuse if $u$ and $v$ are assigned to the same processor.

For each node $v$ in the codelet graph, create an edge $(src_2, v_1)$ in the network.

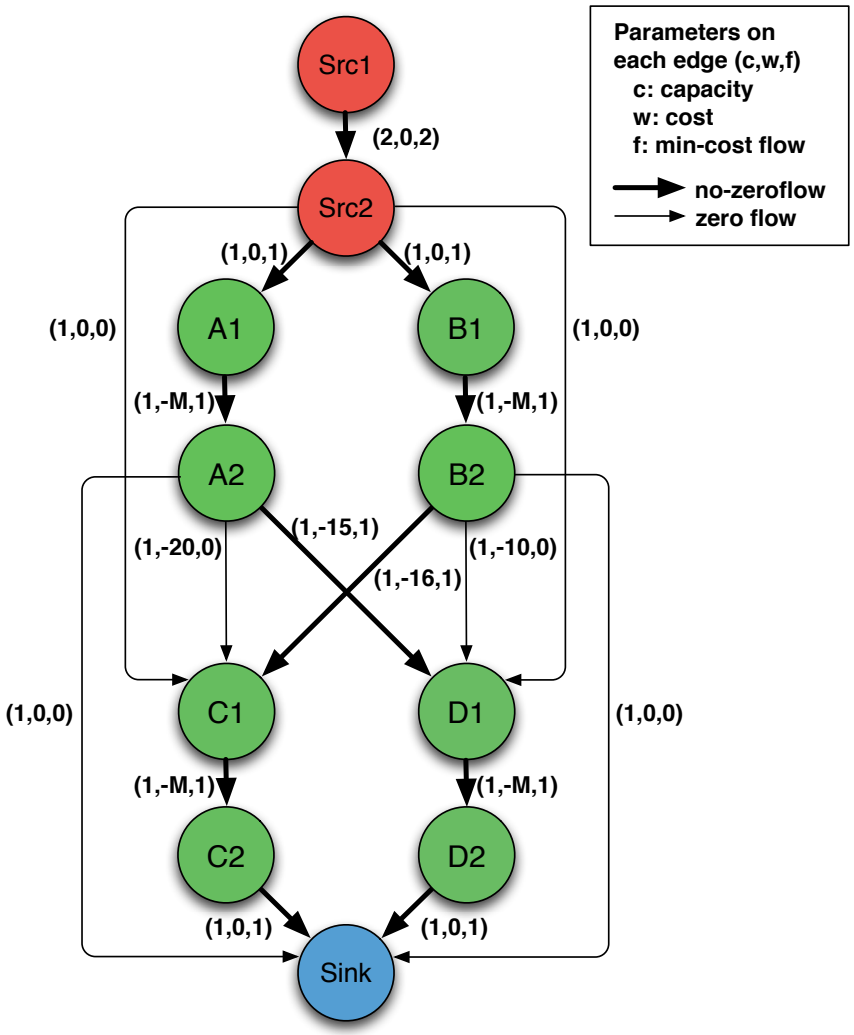


**Figure 4.3:** The flow network converted from the codelet graph in figure 4.1. The resulting min-cost flow is consist of two paths: $Src_1Src_2A_1A_2D_1D_2Sink$ and $Src_1Src_2B_1B_2C_1C_2Sink$. The two paths correspond to the best scheduling plan that schedules codelet $AD$ on one core and $BC$ on the other.

The capacity of the edge is set as 1, which indicates that codelet $v$ will be issued at most once. If the capacity is reached, the codelet $v$ is the first codelet on some core. The cost of the edge is 0.

Create a super vertex *sink* in the network. Then for each node $v$ in the codelet graph, create an edge $(v_2, sink)$ in the network. The capacity of the edge is set as

1, which indicates that codelet $v$ will be terminated at most once. If the capacity is reached, the codelet $v$ is the last codelet on some core. The cost of the edge is 0.

On the network, we will apply the min-cost network flow algorithm to find the optimal solution. There are a number of existing min-cost network flow algorithms such as cycle canceling and shortest path. We choose shortest path algorithm due to its lower time complexity.

Figure 4.3 shows the flow network converted from the codelet graph of Figure 4.1. The min-cost flow is represented by thick arrows. The min-cost flow goes through two paths: $(src_1, src_2, A_1, A_2, D_1, D_2, sink)$ and $(src_1, src_2, B_1, B_2, C_1, C_2, sink)$. That means the corresponding best scheduling plan will schedule $AD$ on one core and $BC$ on the other when the number of cores equals two.

From Figure 4.3, we can see that the min-cost flow based algorithm supports any given number of cores by setting up the edge capacity on $(src_1, src_2)$. Note that the optimal solution corresponds to the given number of cores. For different given numbers of cores, the optimal solutions and scheduling plans might be different.

### 4.2.2 Max First Algorithm

In this section, we introduce a heuristic algorithm called max first algorithm to provide nearly optimal solution for the Best Scheduling Problem. The main idea of the algorithm is to choose the two codelets with maximum potential locality to be executed contiguously on the same core at every step.

The algorithm is shown in Algorithm 5. Initially, all the edges are put into an edge pool. Then the algorithm selects the edge with largest weight from the pool. The two vertices of the edge are scheduled to the same core in some adjacent position. This scheduled edge is removed from the edge pool. The process of selecting and removing repeats until the edge pool is empty.

As a heuristic algorithm, the max first algorithm does not guarantee providing an optimal solution. As shown in Figure 4.1, the max first algorithm exploits 30KB locality by scheduling $AC$ on one core and $BD$ on the other. However, the optimal

**Algorithm 4** Using min-cost flow to solve the Best Scheduling Problem

**input:** Total number of cores $n$
A weighted codelet graph $G =< V, E, W >$

**output:** A vector $par$ stores the parent of each codelet. Codelets $par[i]$ and $i$ will be scheduled to the same core and executed one after the other. If codelet $i$ is the first codelet scheduled to some core, then $par[i]$ equals -1.

**Data:** $N =< V, E, W, C >$ is the flow network that corresponds to $G$, where
$W$ is the weight *i.e.* cost of each edge
$C$ is the capacity of each edge in $N$
$-M$ is a big negative number to represent $-\infty$
$F$ is the min-cost flow

**PSEUDO CODE:**

$N \leftarrow \emptyset$;
$N.V \leftarrow N.V \cup src_1 \cup src_2 \cup sink$;
$N.E \leftarrow N.E \cup (src_1, src_2)$;
$N.W(src_1, src_2) \leftarrow 0$;
$N.C(src_1, src_2) \leftarrow n$;
**for** each $v \in G.V$ **do**
  $N.V \leftarrow N.V \cup v_1 \cup v_2$;
  $N.E \leftarrow N.E \cup (v_1, v_2) \cup (src_2, v_1) \cup (v_2, sink)$;
  $N.W(v_1, v_2) \leftarrow -M$;
  $N.W(src_2, v_1) \leftarrow 0$;
  $N.W(v_2, sink) \leftarrow 0$;
**end for**
**for** each $(u, v) \in G.E$ **do**
  $N.E \leftarrow N.E \cup (u_2, v_1)$;
  $N.W(u_2, v_1) \leftarrow -w(u, v)$;
**end for**
**for** each $(u, v) \in N.E - (src_1, src_2)$ **do**
  $C(u, v) \leftarrow 1$;
**end for**
$F \leftarrow$ MinCostFlow$(N)$;
**for** each $v \in G.V$ **do**
  $par[v] \leftarrow -1$;
**end for**
**for** each $u, v \in G.V$ **do**
  **if** $F(u_2, v_1) == 1$ **then**
    $par[v] \leftarrow u$;
  **end if**
**end for**

solution can exploit 31KB locality by scheduling $AD$ on one core and $BC$ on the other. In our experiments, we observe that the max first algorithm always gives nearly optimal solution (no more than 7.0% worse). We will dissuss the details in Section 4.3.

**Algorithm 5** Max first algorithm

**input:** A weighted codelet graph $G(V, E, W)$
**output:** A vector *par* stores the parent of each codelet. Codelets $par[i]$ and $i$ will be scheduled to the same core and executed one after the other. If codelet $i$ is the first codelet scheduled to some core, then $par[i]$ equals -1.
**Data:** $P$ is an edge pool stored in a binary heap
**PSEUDO CODE:**

```
P ← G.E;
for each v in G.V do
    par[v] ← −1;
end for
while P ≠ ∅ do
    (u, v) ← MaxElement(P);
    P ← P − (u, v);
    par[v] ← u;
    for (u, z) in P do
        P ← P − (u, z);
    end for
    for (z, v) in P do
        P ← P − (z, v);
    end for
end while
```

However, the max first algorithm has lower time complexity than the min-cost flow based algorithm. If we use a heap as the data structure to store the edge pool, its time complexity is $O(nlog(n) + m)$ where $n$ is the total number of codelets and $m$ is the total number of dependencies.

Note that the max first algorithm does not support any given number of cores. This is because the number of the generated groups is fixed for a given codelet graph no matter how many cores are available.

### 4.2.3 Graph Partitioning Based Algorithm

The Best Scheduling Problem can also be converted to a graph partitioning problem. A graph partitioning algorithm [47] partitions the vertices of a weighted graph into multiple groups with respect to minimizing the sum of inter-group weights *i.e.* the weights of edges that go across groups. The graph partitioning algorithm can be used to partition a codelet graph into $n$ groups of codelets. $n$ is the total number

of cores. All codelets in the same group are scheduled to the same core. In such a way, the inter-core locality is minimized

As the max first algorithm, the graph partition algorithm also can not guarantee giving an optimal solution. The reason is that it only aims to minimizing the inter-core locality instead of maximizing the intra-core locality. In our experiments, we find that the graph partition based algorithm always exploits least locality among the three algorithm. Moreover, the graph partition algorithm does not guarantee to utilizing the parallelism in the codelet graph. The reason is that it is possible to classify the codelets with no dependencies into the same group in its solution. Therefore, some meaningless dependencies have to be forced to fix the execution order on the same core. On the other hand, the graph partition based algorithm is able to handle arbitrary number of cores.

## 4.3 Experiment

We evaluate the three algorithms on the Cyclops-64 architecture using the codelet model. In Section 4.3.1, we introduce the experimental design. In Section 4.3.2, the major experimental observations are presented. In section 4.3.3, the experimental results are provided and analyzed.

### 4.3.1 Experimental Design

Figure 4.4 shows the overview of our experiment design. In the system, there are the two modules explained as follows.

- **Scheduling plan generator:** This module uses the three algorithms in Section 4.2 to automatically generate the codelet scheduling plan for locality exploitation. The inputs of the module are the static codelet graph, the potential locality information among the codelets, and the total number of cores. According to Cyclops-64 architecture, we assume that each core executes only one codelet at a time. The output of the module are the scheduling plans generated by the three algorithms.
- **Runtime scheduling emulator:** This module emulates the codelet runtime that schedules the codelet on a Cyclops-64 node. The emulator uses either the default scheduling approach or the input scheduling plan. The default scheduling

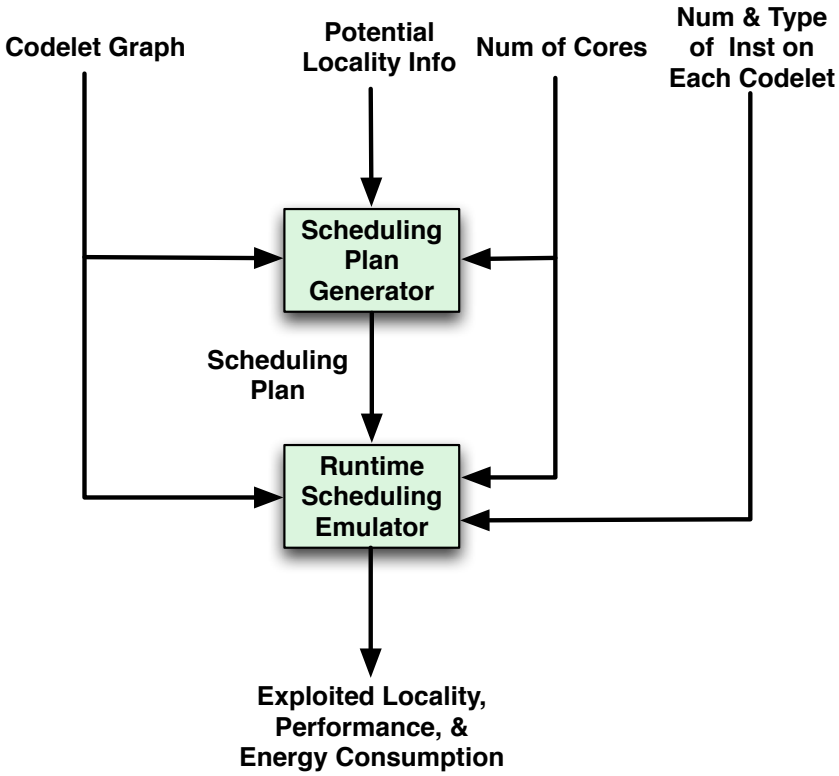


**Figure 4.4:** Experiment design. The system consists of two components. Scheduling plan generator uses the three algorithms in Section 4.2 to automatically generate the codelet scheduling plan for locality exploitation. Runtime scheduling emulator emulates the codelet runtime that schedules the codelet on a Cyclops-64 node.

approach focuses on workload balancing but it is not aware of locality, which matches the dynamic codelet scheduling approaches. The input scheduling plan is able to exploit locality to some extent depending on the algorithm that generates the plan. The other inputs of the module are the codelet graph, the total number of cores, and the numbers and types of instructions in each codelet. The output of the module are the exploited locality, performance, and energy consumption by using the different scheduling plan on the codelet graph, respectively.

We implement the following six applications in our experiments:

- `mm` (matrix multiplication kernel): This application is based on the previous study of matrix multiplication on Cyclops-64 [38]. It computes $C = A \times B$ where $A$,$B$, and $C$ are all $192 \times 192$ matrices whose elements are double precision floating point numbers. $C$ is further partitioned into many $6 \times 6$ tiles. Each codelet

works on the multiplication of a $6 \times 192$ matrix and a $192 \times 6$ matrix to generate a $6 \times 6$ tile in $C$. Therefore, there are 1024 codelets corresponding to 1024 tiles in $C$. Each codelet executes $6 \times 6 \times 192 = 6912$ float multiply-add and $192 \times 6 \times 2 = 2304$ load instructions on $A$ and $B$ in total. If a codelet shares the data *e.g.* a $6 \times 192$ matrix in $A$ with its precedent codelet on the same core, it can load the shared data from local memory instead of global memory by $6 \times 192 = 1152$ load instructions. Therefore, the potential locality between them is $8Bytes \times 192 \times 6 = 9216Bytes$. Note that Matrix multiplication is a very special case, because There are no dependencies but potential locality among codelets. It is obvious to find an solution to achieve the maximum locality exploitation. In our case, all three scheduling algorithms divide the`mm` codelet graph into exactly the same execution flows. Therefore, from Figure 4.5, Figure 4.6, Figure. 4.7, and Figure 4.8, we can see that all the three scheduling algorithms perform the same. Note that `mm` is the most suitable case for `GP` on the six applications, because `GP` algorithm divides codelets into groups regardless of the codelet dependencies while the inter-group locality is minimized.

- `ms` (merge sort kernel): The merge sort application is implemented on $10K$ integers using 7 merging levels. Therefore, the codelet graph is constructed as a binary tree with 7 levels. In the graph, there are $2^7 - 1 = 127$ codelets. If a codelet is in level $l$ ($0 \leq l \leq 6$), it needs to execute $10K/2^l$ integer comparison, load, and store instructions respectively. The data that a codelet in level $l$ requires comes from its two leaf codelets (each produces half of the data) in level $l+1$. Therefore, if a codelet in level $l$ is scheduled to the same core as one of its leaf codelet in level $l+1$, the amount of the potential locality is $4Bytes \times 10K/2^{(l+1)}$.

- `rt_ci` (random tree with computation-intensive codelets): In this application, a codelet graph is randomly generated as a tree with 160 codelets and 160 dependency edges. Each codelet is computation-intensive. That means the amount of computation instructions is 6 times of the amount of memory access instructions in a codelet. Because `mm` is a typical computation-intensive application, it also matches this ratio.

- `rt_mi` (random tree with memory-intensive codelets): This application also produces tree-structure codelet graph with 160 codelets 160 dependency edges in a random way. However, each codelet is memory-intensive which means the amount of computation instruction equals to the amount of memory access instructions in a codelet. As `ms` is a typical memory-intensive application, it also matches this ratio.

- `rg_ci` (random graph with computation-intensive codelets): This application is similar to `rt_ci`. However, the codelet graph is a randomly generated graph with 160 codelets and 320 dependency edges. In our observation, most codelet graphs have low average fanout *e.g.*, around 2 for a codelet graph that represents a parallel *for* loop. That is why we set the average fanout to be 2.

- `rg_mi` (random graph with memory-intensive codelets): This benchmark is similar to `rt_mi`. However, the codelet graph is a randomly generated graph with 160 codelets and 320 dependency edges.

We assume that the data of all the applications is stored in the off-chip memory initially. In computation-intensive applications *i.e.* `mm`, `rt_ci`, and `rg_ci`, we assume that the latency of memory accesses can be fully hidden by computation. Therefore, each memory access instruction only takes one cycle to issue and zero cycle delay. In memory-intensive applications *i.e.* `ms`, `rt_mi`, and `rg_mi`), we assume that the latency of memory accesses can not be hidden by computation. Therefore, according to the Cyclops-64 instruction timing, a load instruction from the scratchpad memory takes 1 cycle issue and 2 cycles delay and from the off-chip memory takes one cycle issue and 57 cycles delay. Since a store instruction on Cyclops-64 does not need acknowledgement, it takes one cycle issue and zero cycle delay. We also assume that all the computation instruction delay in a codelet can be hidden *i.e.* it takes cycle issue and zero cycle delay to execution a computation instruction.

In our experiments, we tested 4 scheduling algorithms. They are described in Table 4.1: `Base`, `MCF`, `MF`, and `GP`. `Base` focuses on workload balancing but are not aware of locality exploitation, which matches the dynamic codelet scheduling approaches. In `Base`, the codelet runtime maintains a global codelet queue. When a codelet satisfies all the dependencies, the codelet runtime puts it into the codelet queue. The runtime selects an available core to execute the ready codelet. If more than one available cores

**Table 4.1:** The description of the four algorithms used to schedule codelets on Cyclops-64

| Name | Description |
|---|---|
| `Base` | Basic scheduling without locality exploitation |
| `MCF` | Min-cost flow based algorithm (see Section 4.2.1) |
| `MF` | Max-first algorithm (see Section 4.2.2) |
| `GP` | Graph partitioning based algorithm (see Section 4.2.3) |

exist, the runtime pick one in a random way. The other three algorithms have different tradeoffs on locality exploitation, algorithm complexity, program execution time, and energy efficiency. We assume that the codelet scheduling takes very small overhead, because in general the execution time of a codelet is much larger than the overhead of the scheduling.

### 4.3.2 Major Observations

From our experimental results, the major observations are shown as below:

- `MCF` always provides best locality exploitation. It reduces up to 59.7% of global memory accesses. `MF` is in the second place and within 7.0% of difference comparing to `MCF`.
- `MCF` outperforms the other scheduling algorithms on all the applications. `MCF` achieves up to 68.1% of performance improvement comparing to `Base`. `MF` is in the second place and within 9.1% of difference comparing to `MCF`.
- `MCF` provides best energy saving on both overall and dynamic energy consumptions. It reduces up to 40.7% overall energy and 59.2% dynamic energy comparing to `Base`. `MF` is in the second place and within 8.5% of difference on overall energy and 3.6% on dynamic energy comparing to `MCF`.

### 4.3.3 Experimental Result

In this section, we present and analyze the experimental results of the four scheduling algorithms from locality exploitation, performance, and energy efficiency aspects.

**Locality exploitation**

Figure 4.5 shows the best locality exploitation of three algorithms including `MCF`, `MF`, and `GP` used on the six applications. The result of `Base` is not shown, because it is not aware of locality. In the figure, the x-axis represents the six applications. The y-axis represents the locality exploitation by the percentage of global memory accesses that have been reduced by buffer in local memory. From Figure 4.5, we can see that:

- `MCF` exhibits best locality exploitation among the three algorithms. It reduces up to 59.7% of global memory accesses. The reason is that `MCF` guarantees optimal solution that maximizes the locality exploitation. The other two algorithms may not reach optimal solution because they are heuristic algorithms.

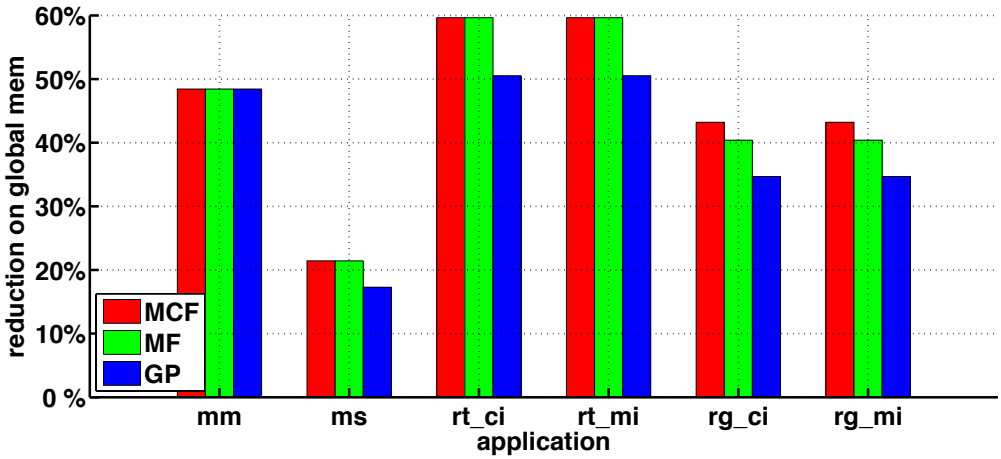


**Figure 4.5:** The best locality exploitation on the six applications by using the three algorithms from Section 4.2. X-axis represents the six applications. Y-axis represents the percentage of global memory accesses that have been reduced by each algorithm. Higher is better. `MCF` exhibits best locality exploitation. It reduces up to 59.7% of global memory accesses on application `rt_ci` and `rt_mi`. `MF` provides a good quality solutions (within 7.0% of difference comparing to `MCF`). `GP` is the worst.

- `MF` provides nearly optimal solutions. The difference of locality exploitation is within 7.0% between `MF` and `MCF`. The worst case of `MF` happens on `rg_ci` and `rg_mi` because the codelet graphs are too complicated. The other four applications have simpler codelet graphs (either tree structure or uniformed weight). Therefore, `MF` finds optimal or nearly optimal solutions for them.

- `GP` finds worst solutions among the three algorithms. The reason is that `GP` tends to minimize the inter-core locality at the expense of the intra-core locality which is the real target of locality exploitation.

**Performance**

Figure 4.6 shows the performance of the four algorithms used on the six applications. The x-axis represents the six applications. The y-axis represents the normalized execution time of each application by using the four scheduling algorithms, respectively. To make the comparison fair, all the algorithms use the same number of cores. We set the amount to be equivalent to the requirement of `MF` because it is the only algorithm that does not support any given number of cores. From Figure 4.6, the observations are shown as below:

- `MCF` performs the best among the four algorithms. It achieves up to 68.1% of performance improvement comparing to `Base` on `rg_mi`. The latency of global

memory *i.e.* DRAM accesses are much longer than that of local storage *i.e.* scratchpad memory accesses. For memory-intensive applications, the latency of memory can not be hidden by computation. Therefore, we can take advantage of the locality exploitation to reduces execution time of each codelet greatly. The performance of these applications are improved remarkably. However, we can see that the performance of computation-intensive applications is not improved by sheduling the codelets based on locality exploitation. This is because the latency of memory accesses no matter on DRAM or scratchpad memory can be hidden completely. Therefore, DRAM and scratchpad memory access take the same execution time *i.e.* one cycle to issue and zero cycle delay.

- `MF` is in the second place. The reason is that its locality exploitation is worse than `MCF`. In special cases, better locality exploitation may not guarantee better performance. However, in our experiments we haven't observed such a special case.

- `GP` performs worse than `MCF` and `MF`. There are two reasons: (1) `GP` performs the worst in locality exploitation among the three algorithms; and (2) `GP` does not consider the dependencies during partitioning and may introduce unnecessary dependencies into the codelet graph, which may reduce the parallelism of the application. However, `GP` still outperforms `Base` for memory-intensive applications because the locality exploitation reduces the latency of memory accesses. For computation-intensive applications, `GP` may perform worse than `Base` due to the parallelism reduction *e.g.* `rg_ci`.

**Energy efficiency**

**Table 4.2:** Energy consumption per instruction.

| Instruction | Energy (pJ/Operation) |
|---|---|
| lddram | 48924.10 |
| stddram | 51488.99 |
| lddsram | 964.65 |
| stdsram | 548.31 |
| mov | 105.48 |
| lddspm | 535.065 |
| stdspm | 326.895 |
| fmad | 245.27 |
| add | 127.65 |

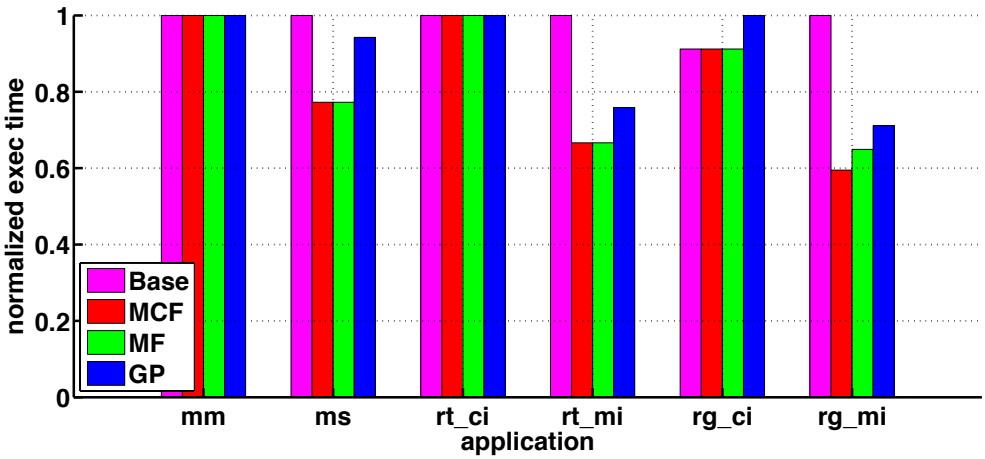


**Figure 4.6:** The performance of the four algorithms on the six applications. X-axis represents the six applications. Y-axis features the normalized execution time of the applications by using the four scheduling algorithms. Lower is better. `MCF` outperforms the other algorithms (up to 68.1% of performance improvement comparing to `Base`). `MF` is the second best (no more than 9.1% slower comparing to `MCF`). `Base` is the worst in most of the cases but it outperforms `GP` on `rg_ci`.

The overall energy consumption of an application consists of static and dynamic energy consumptions. The static energy consumption is determined by the execution time. On Cyclops-64, it is 64.11W as explained in [36]. The dynamic energy is determined by the number and type of the executed instructions. Table 4.2 shows the energy consumption of various instructions on Cyclops-64. Most of the data comes from the early energy study on Cyclops-64 [36]. `lddram`, `lddsram`, and `lddspm` are double-word load instructions on DRAM memory, SRAM memory, and SPM, respectively. Correspondingly, `stddram`, `stdsram`, and `stdspm` are the store instructions. `mov` is the access on a double-word register. `fmad` is the multiple and add computation on double precision floating point numbers. We use `add` to represent integer and logical operations because they consume almost the same amount of energy. However, the energy consumption of scratchpad memory operations *i.e.* `lddspm` and `stdspm` is not previously provided in [36]. The following formula is used to estimate them.

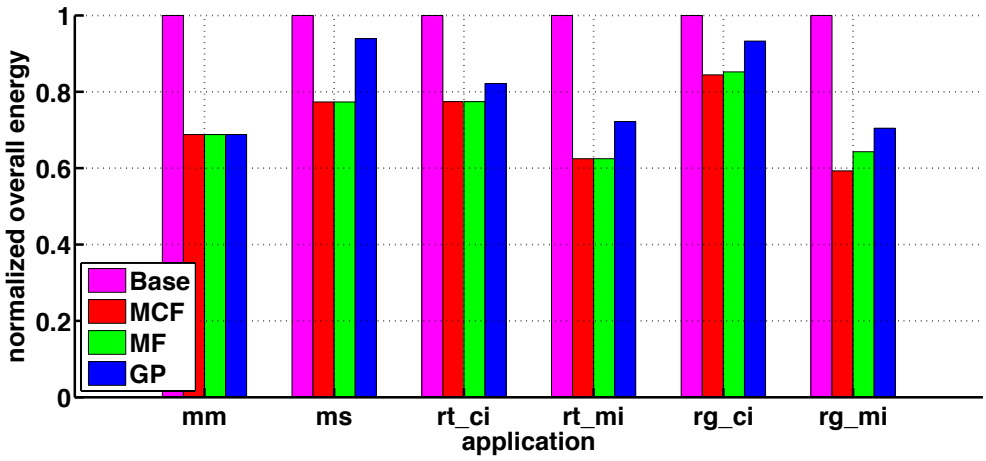


**Figure 4.7:** The overall energy consumption of the six applications by using the four algorithms. X-axis represents the six applications. Y-axis represents the normalized overall energy consumption of each algorithm applied to the six applications. Lower is better. `MCF` achieves the most efficient energy consumption. It reduces up to 40.7% of overall energy comparing to `Base`. `MF` is in the second place (within 8.5% difference comparing to `MCF`).

$$lddspm\ (or\ stdspm) = (lddsram\ (or\ stdsram) - mov) \times ratio + mov$$

We choose *ratio* as 1/3 because the energy consumption of SPM accesses is closer to register accesses than SRAM accesses.

Figure 4.7 and Figure 4.8 show the normalized overall and dynamic energy consumption of the six applications by using the four algorithms, respectively. From the two figures, the observations are made as below:

- `MCF` performs the most efficiently on energy consumption. It reduces up to 40.7% of overall energy and 59.2% of dynamic energy compared to `Base`. The reason is that `MCF` provides the best locality exploitation as well as the best performance.
- `MF` is in the second place because it always produces a nearly optimal solution for locality exploitation as well as performance.
- `GP` is worse than the other two algorithm because it gives the worst solution for locality exploitation as well as performance.

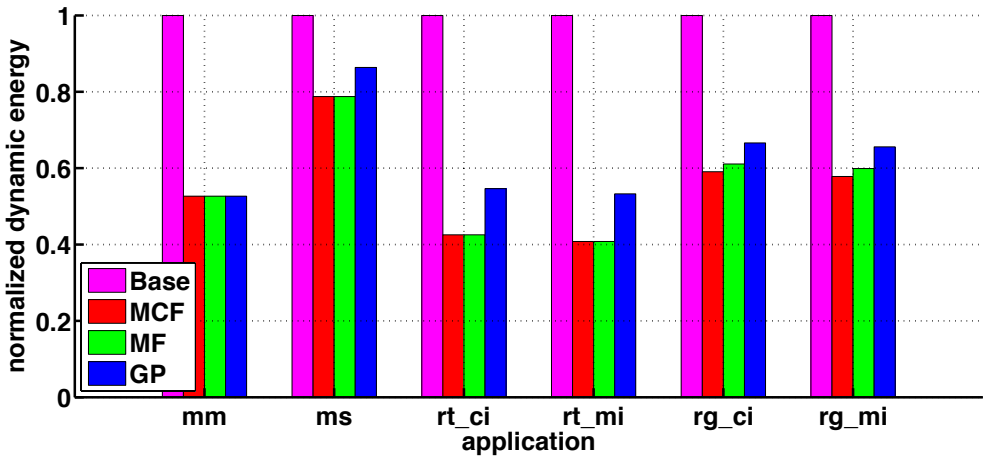


**Figure 4.8:** The dynamic energy consumption of the six applications by using the four algorithms. X-axis represents the six applications. Y-axis represents the normalized dynamic energy consumption of each algorithm applied to the six applications. Lower is better. `MCF` achieves the most efficient dynamic energy consumption. It reduces up to 59.2% of dynamic energy comparing to `Base`. `MF` is in the second place (within 3.6% difference comparing to `MCF`).

Note that if the application involves more memory accesses and the corresponding codelet graph gets more complicated the performance gap among the algorithms gets bigger.

## 4.4 Related Work

The codelet execution model [75] is inspired by the classical dataflow model proposed by Dennis [22]. It is hybrid model which incorporates the advantages of macro-dataflow [68] and the Von Neumann model [25]. The codelet execution is driven asynchronously by events which can be data dependency and resource requirement. That is main difference between the codelet model and its ancestor *i.e.* the EARTH system [71] which expresses the parallelism in a data-driven manner. All codelets are organized as a DAG to represent these relations. Inside a codelet, the program is executed in a control flow manner. The codelet execution model provides an opportunity to

maximize the parallelism while minimize the overhead of resource utilization. As multi and many-core systems are becoming popular, codelet based execution model is drawn more and more attention. The works based on the codelet model include ParalleX execution model [33], SWift Adaptive Runtime Machine (SWARM) [49], TIDeFlow [59], and FreshBreeze [24]. The proposed scheduling algorithm to exploit locality can be applied on these systems to achieve better performance and energy saving. To properly exploit the techniques proposed, these systems would require a adequate facilities to analyze the generated dataflow graph and schedule them accordingly. Moreover there are multiple efforts to exploit the fine-grained parallelism offered by dataflow. The EU's Teraflux project has explored Data-Driven Multithreading (DDM) where a collection of instructions called data-driven threads are connected by data dependencies to form a Synchronization Graph [19]. In addition there are multiple pragma based dataflow systems including StarSs [63] and OpenMP-based OpenStream [64]. These models also pose to benefit from our proposed technique.

Lee *et al.* propose utilizing static scheduling in the synchronous dataflow model [51]. However, synchronous dataflow is not equivalent to the codelet model. Synchronous dataflow requires a priori knowledge of the number of inputs on a single arc before invoking an actor. It is typically used in signal processing applications and is not related to our work.

There exists a very similar problem called program allocation in the data-flow processor design [23, 43, 61]. The aim is to maximize inherent parallelism while minimizing communication overhead. To solve the problem, Lee *et al.* provide a heuristic approach based on static scheduling [50]. In this work, the compromised allocation plan is generated by making use of execution time and communication cost information. Our approach is different from their work by applying to the codelet model instead of the original dataflow model. The codelets are more complicated than the dataflow actors. The execution time of a codelet may vary in each run, because of the contention on hardware resources *e.g.* memory. Our approach can also guarantees to exploit maximum concurrency in a application even if the execution time information

for all codelets in a graph is not provided.

The partitioned global address space (PGAS) is a parallel programming model that supports locality exploitation to improve the performance in an abstracted share address space. Based PGAS, several languages are developed to address locality: Unified Parallel C (UPC) [9], Co-Array Fortran (FAF) [57], and Titanium [73] are PGAS extensions to C, Fortran, and java, respectively. In these three languages, the references to global and local memory are explicitly distinguished by the type system. As High Productivity Computing Systems program (HPCS) languages, X10 [11] and Chapel [10] use a variant of the PGAS model, asynchronous partitioned global address space (APGAS) to support locality optimization in shared data structures. The locality abstractions *i.e.* X10's places[11] and Chapel's locales[10] are provided to associate computation elements with data locations. As NUMA is prevalent in multi-core and many-core systems, locality optimization can greatly improve the performance. Huang *et al.* propose the extension to OpenMP with locality awareness [45]. In addition, locality optimization also involves in work stealing in SLAW [42], CATS [16]. Locality exploitation can also be utilized in compiler optimization to improve the performance. Gao *et al.* and Govindarajan *et al.* propose the approach to reuse the register and cache with the help of compiler [35, 40].

## 4.5 Discussion

In this thesis, we introduce three scheduling algorithms to exploit locality by using the codelet graph and locality information. The experiment is based on a runtime scheduling emulator which emulates the codelet runtime that schedules the codelet on a Cyclops-64 node. To make the experiment more convincing, we should deploy and compare the three scheduling algorithms on the real codelet runtimes *e.g.* SWARM [49] and DARTS [69]. To explore the locality, the codelet runtimes must support static scheduling based on the plan generated by the algorithms. Between two adjacent codelets on the same core, the codelet runtimes must provide a locale to store the

reusable data and a mechanism to pass the address of the locale. The locale management *e.g.* allocation and de-allocation is better to be handled by the codelet runtimes instead of users.

The three algorithms can only work on the static codelet graph. To make them more useful, the algorithms should be extended to the case that the codelet graph can not be statically determined *e.g.* recursion. At least, the algorithms can be used in a Thread Procedure (TP) which is static and acyclic in such a case.

# Chapter 5

# CONCLUSION AND FUTURE WORK

In this thesis, we focus on the memory optimization on memory workload balance and locality exploitation in the codelet execution model. As an case study, various versions of FFT algorithms are implemented on Cyclops-64 to demonstrate that the fine-grain codelet execution model is able to execute the codelets that involve different workload on the memory bandwidth in a guided order to achieve good memory usage as well as performance. The experiment result shows that our fine-grain guided algorithm achieves up to 46% performance improvement comparing to a state-of-the-art implementation on Cyclops-64. To exploit locality in codelet execution, three scheduling algorithms are proposed and compared. They have different trade-offs in algorithmic complexity, locality exploitation, program execution time, and energy efficiency. We test and analyze the three algorithms on various applications on Cyclops-64. The experiment result on shows that our algorithms reduce up to 59.7% of global memory access and improve up to 68.1% performance improvement and 40.7% energy saving comparing to the state-of-the-art codelet scheduling approach.

In the future, we intend to extend the codelet execution model from multi-core and many-core system to cluster computing system for distributed processing of large data sets. By taking the advantage of the codelet model, the cluster computing system tends to achieve better task synchronization and resource utilization including CPU and memory than other existing systems *e.g.* Hadoop [72], Spark [74], etc. Therefore, the system performance can be improved. Moreover, we also plan to extend the traditional MapReduce programming style to more flexible codelet graph programming style that is able to implement a complicated problem in a more efficient way. Hence, the codelet

model could be a good solution for both batch and real time processing for large data sets.

## BIBLIOGRAPHY


[1] IBM Cell Broadband Engine, *http://www-01.ibm.com /chips/techlib/techlib.nsf/products/cell_broadband_engine*.

[2] A. Agarwal. The Tile Processor: A 64-core Multicore for Embedded Processing. In *Proceedings of the 11th Annual Workshop on High Performance Embedded Computing (HPEC-07) (Presentation)*, Lexington, MA, USA, September 18-20, 2007.

[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, Seattle, Washington, USA, June 17-21, 2007.

[4] J. R. Allen, B. M. Bass, C. Basso, R. H. Boivie, J. L. Calvignac, G. T. Davis, L. Frelechoux, M. Heddes, A. Herkersdorf, and A. Kind et al. IBM PowerNP Network Processor: Hardware, Software, and Applications. *IBM Journal of Research and Development*, 47(2.3):177–193, 2003.

[5] D. A. Bader and V. Agarwal. FFTC: Fastest Fourier Transform for the IBM Cell Broadband Engine. In *Proceddings of the 14th International Conference on High Performance Computing (HiPC-07)*, pages 172–184, Goa, India, December 18-21, 2007.

[6] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, 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 *Proceedings of IEEE International Solid-State Circuits Conference (ISSCC-08), Digest of Technical Papers*, pages 88–598, San Francisco, CA, USA, February 3-7, 2008.

[7] P. Bientinesi, N. P. Pitsianis, and X. Sun. Parallel 2D FFTs on the Cell Broadband Engine. *Department of Computer Science Duke University Technical Report CS-2007-03*, 2007.

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

[9] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. *Introduction to UPC and Language Specification*. IDA Center for Computing Sciences Technical Report CCS-TR-99-157, 1999.

[10] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[11] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: An Object-Oriented Approach to Non-uniform Cluster Computing. *ACM SIGPLAN Notices*, 40(10):519–538, 2005.

[12] C. Chen, Y. Wu, J. Suetterlein, L. Zheng, and G. R. Gao. Towards An Energy-Efficient Scheduler in the Codelet Model. In *IEEE Symposium on Low-Power and High-Speed Chips (IEEE COOL Chips XVI) (Poster)*, Yokohama, Japan, April 17-19, 2013.

[13] C. Chen, Y. Wu, J. Suetterlein, L. Zheng, M. Guo, and G. R. Gao. Automatic Locality Exploitation in the Codelet Model. In *Proceedings of the 11th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA-13)*, pages 853–862, Melbourne, Australia, July 16-18 2013.

[14] C. Chen, Y. Wu, S. Zuckerman, and G. R. Gao. Towards Memory-Load Balanced Fast Fourier Transformations in Fine-grain Execution Models. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW-13)*, pages 1607–1617, Washington, DC, USA, May 24, 2013.

[15] L. Chen, Z. Hu, J. Lin, and Guang R. Gao. Optimizing the Fast Fourier Transform on a Multi-core Architecture. In *Proceedings of Workshop on Performance Optimization for High-Level Languages and Libraries in the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPSW-07)*, pages 1–8, Long Beach, CA, USA, March 26-30, 2007.

[16] Q. Chen, M. Guo, and Z. Huang. Cats: Cache Aware Task-Stealing based on Online Profiling in Multi-socket Multi-core Architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS-12)*, pages 163–172, New York, NY, USA, June 25-29, 2012.

[17] J. Choi, J. Kim, and C. Kim. Parallel Implementation of the FFT Algorithm Using a Multi-core Processor. In *2010 International Forum on Strategic Technology (IFOST)*, pages 19 –22, University of Ulsan, Ulsan, South Korea, October 13-15, 2010.

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

[19] C. Christofi, G. Michael, P. Trancoso, and P. Evripidou. Exploring HPC Parallelism with Data-Driven Multithreating. In *Proceedings of Data-Flow Execution Models for Extreme Scale Computing (DFM-12)*, pages 10–17, Minneapolis, MN, USA, September 19-23, 2012.

[20] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl. Evaluation and Improvements of Programming Models for the Intel SCC Many-core Processor. In *Proceedings of International Conference on High Performance Computing and Simulation (HPCS-11)*, pages 525–532, Istanbul, Turkey, July 4-8, 2011.

[21] J. Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. TiNy Threads: a Thread Virtual Machine for the Cyclops64 Cellular Architecture. In *Proceedings of the Fifth Workshop on Massively Parallel Processing, in conjuction with 19th International Parallel and Distributed Processing Symposium (IPDPSW-05)*, pages 8–15, Denver, CO, USA, April 4-8, 2005.

[22] J. B. Dennis. First Version of a Data Flow Procedure Language. In *Programming Symposium Lecture Notes in Computer Science*, volume 19, pages 362–376. Springer, 1974.

[23] J. B. Dennis. Data Flow Supercomputers. *Computer*, 13(11):48–56, 1980.

[24] J. B. Dennis. Fresh Breeze: a Multiprocessor Chip Architecture Guided by Modular Programming Principles. *ACM SIGARCH Computer Architecture News*, 31(1):7–15, 2003.

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

[26] Y. Dotsenko, S. S. Baghsorkhi, B. Lloyd, and N. K. Govindaraju. Auto-tuning of Fast Fourier Transform on Graphics Processors. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP-11)*, pages 257–266, San Antonio, TX, USA, February 12-16, 2011.

[27] 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. *IEEE Journal of Solid-State Circuits*, 47(1):220 –231, 2012.

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

[29] F. Franchetti, Y. Voronenko, and M. Püschel. FFT Program Generation for Shared Memory: SMP and Multicore. In *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing (SC-06)*, pages 51–65, Tampa, FL, USA, November 11-17 2006.

[30] M. Frigo. A Fast Fourier Transform Compiler. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-99)*, pages 169–180, Atlanta, GA, USA, May 1-4, 1999.

[31] M. Frigo and S. G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP-98)*, pages 1381–1384, Seattle, WA, USA, May 12-15 1998.

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

[33] G. R. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu. Parallex: A study of a new parallel computation model. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS-07)*, pages 1–6, Long Beach, CA, USA, March 26-30 2007.

[34] G. R. Gao, J. Suetterlein, and S. Zuckerman. Toward an Execution Model for Extreme-Scale Systems - Runnemede and Beyond. Technical Memo 104, April 2011.

[35] R. G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective Loop Fusion for Array Contraction. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing (LCPC-92)*, pages 281–295, New Haven, CT, USA, August 3-5, 1992.

[36] E. Garcia, D. Orozco, and G. R. Gao. Energy Efficient Tiling on a Many-core Architecture. In *Proceedings of the 4th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-11)*, Heraklion, Crete, Greece, January 23, 2011.

[37] 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 ACM International Conference on Computer Frontiers (CF-12)*, pages 245–248, Cagliari, Italy, May 15-17, 2012.

[38] E. Garcia, I. E. Venetis, R. Khan, and G. R. Gao. Optimized Dense Matrix Multiplication on a Many-core Architecture. In *Proceedings of International European Conference on Parallel and Distributed Computing (EuroPar-10)*, pages 316–327, August 31-September 3, 2010.

[39] 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. *IEEE Micro*, 28(4):13–27, 2008.

[40] R. Govindarajan, H. Yang, J. N. Amaral, C. Zhang, and G. R. Gao. Minimum Register Instruction Sequence Problem: Revisiting Optimal Code Generation for DAGs. In *Proceedings 15th International Parallel and Distributed Processing Symposium (IPDPS-01)*, pages 8–15, San Francisco, CA, USA, April 23-27, 2001.

[41] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High Performance Discrete Fourier Transforms on Graphics Processors. In *Proceedings of the ACM/IEEE Conference on High Performance Computing (SC-08)*, pages 2–13, Austin, TX, USA, November 15-21 2008.

[42] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. SLAW: a Scalable Locality-aware Adaptive Work-stealing Scheduler. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS-10)*, pages 1–12, Atlanta, GA, USA, April 19-23 2010.

[43] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester Prototype Dataflow Computer. *Communications of the ACM*, 28(1):34–52, 1985.

[44] M. Harris. GPGPU: General-purpose Computation on GPUs. *SIGGRAPH 2005 GPGPU COURSE*, 2005.

[45] L. Huang, H. Jin, L. Yi, and B. Chapman. Enabling Locality-aware Computations in OpenMP. *Scientific Programming*, 18(3):169–181, 2010.

[46] A. H. Kamalizad, C. Pan, and N. Bagherzadeh. Fast Parallel FFT on a Reconfigurable Computation Platform. In *Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD-03)*, pages 254–259, Sao Paulo, Brazil, November 10-12, 2003.

[47] G. Karypis and V. Kumar. Multilevel Algorithms for Multi-constraint Graph Partitioning. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC-98)*, pages 1–13, Orlando, FL, USA, November 7-13 1998.

[48] R. Knauerhase, R. Cledat, and J. Teller. For Extreme Parallelism, Your OS is Sooooo Last-millennium. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism (HotPar-12)*, pages 3–3, Berkeley, CA, USA, June 7-8, 2012.

[49] C. Lauderdale and R. Khan. Towards a Codelet-based Runtime for Exascale Computing: Position Paper. In *Proceedings of the 2nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT-12)*, pages 21–26, London, United Kingdom, March 3, 2012.

[50] B. Lee, A. R. Hurson, and T. Y. Feng. A Vertically Layered Allocation Scheme for Data Flow Systems. *Journal of Parallel and Distributed Computing*, 11(3):175–187, 1991.

[51] E. A. Lee and D. G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, C-36(1):24 –35, 1987.

[52] 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.

[53] D. B. Lloyd, C. Boyd, and N. Govindaraju. Fast Computation of General Fourier Transforms on GPUs. In *Proceedings of the IEEE International Conference on Multimedia and Expo (ICME-08)*, pages 5–8, Hannover, Germany, June 23-26, 2008.

[54] 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 ACM/IEEE Conference on High Performance Computing (SC-08)*, pages 38–48, Austin, TX, USA, November 15-21, 2008.

[55] 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, Santa Fe, NM, USA, May 8-11, 2000.

[56] 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, San Diego, CA, USA, July 26-27 2003.

[57] R. W. Numrich and J. Reid. Co-array Fortran for Parallel Programming. *ACM SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[58] D. Orozco, E. Garcia, R. Khan, K. Livingston, and G. R. Gao. Toward High-throughput Algorithms on Many-core Architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):49:1–21, 2012.

[59] D. Orozco, E. Garcia, R. Pavel, R. Khan, and G. R. Gao. TIDeFlow: The Time Iterated Dependency Flow Execution Model. In *Proceedings of the 1st Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM-11)*, pages 1–9, Galveston, TX, USA, October 10, 2011.

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

[61] G. M. Papadopoulos and D. E. Culler. Monsoon: an Explicit Token-Store Architecture. *ACM SIGARCH Computer Architecture News*, 18(3a):82–91, 1990.

[62] M. Pippig. An Efficient and Flexible Parallel FFT Implementation Based on FFTW. In *Proceedings of International Conference on Competence in High Performance Computing (CiHPC-10)*, pages 125–134, Schloss Schwetzingen, Germany, June 22-24, 2012.

[63] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical Task-Based Programming with StarSs. *International Journal of High Performance Computing Applications*, 23(3):284–299, 2009.

[64] A. Pop and A. Cohen. OpenStream: Expressiveness and Data-Flow Compilation of OpenMP Streaming Programs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):53:1–53:25, 2013.

[65] 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. *IEEE Journal of Solid-State Circuits*, 46(4):757 –766, 2011.

[66] A. B. Saybasili, A. Tzannes, B. R. 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 on Parallel and Distributed Computing and Systems (PDCS-09)*, pages 107–113, Cambridge, MA, USA, November 24, 2009.

[67] 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 Transactions on Graphics*, 27(3):1–15, 2008.

[68] J. Silc, B. Robic, and T. Ungerer. Asynchrony in Parallel Computing: From Dataflow to Multithreading. *Journal of Parallel and Distributed Computing Practices*, 1(1):3–30, 1998.

[69] J. Suettlerlein, S. Zuckerman, and G. R. Gao. An Implementation of the Codelet Model. In *Proceedings of the 19th International European Conference on Parallel and Distributed Computing (EuroPar-13)*, Aachen, Germany, August 26-30, 2013.

[70] D. Takahashi. Implementation and Evaluation of Parallel FFT Using SIMD Instructions on Multi-core Processors. In *Proceedings of International Workshop on Innovative Architecture for Future Generation High-performance Processors and Systems (IWIA-07)*, pages 53–59, Maui, HI, USA, January 11-13, 2007.

[71] K. B. Theobald. *EARTH: An Efficient Architecture for Running Threads*. PhD thesis, McGill University, Montreal, Quebec, Canada, May 1999. AAINQ50269.

[72] T. White. *Hadoop: the definitive guide*. O'Reilly, 2012.

[73] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A High-performance Java Dialect. *Concurrency Practice and Experience*, 10(11-13):825–836, 1998.

[74] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud-10)*, pages 10–16, Boston, MA, USA, June 22-25, 2010.

[75] S. Zuckerman, J. Suetterlein, R. Knauerhase, and Guang 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, San Jose, CA, USA, June 5, 2011. ACM.