Parallelize streaming applications on Microgrid CPUs: A novel application on a scalable, multicore architecture.

Thesis

Presented in partial Fulfillment of the Requirements for the Degree of Master of Science in the Graduate School of The Ohio State University

By

Abhishek Mishra

Graduate Program in Computer Science and Engineering

The Ohio State University

2014

Thesis Committee

Dr. Rajiv Ramnath, Advisor

Dr. Radu Teodorescu
Abstract

With the continuing necessity of moving to many core architectures, it has become imperative for programming paradigms to evolve. Programming languages, compilers and architectures need to work together to make it easier to extract as much concurrency as possible from an application with little increase in programmer effort.

A Microgrid architecture is one such scalable architecture. Currently optimization for this architecture happens at a low level and is only limited to ‘for’ loops that have minimal or no dependence between their iterations. However, the physical arrangement of CPUs and their interconnection on this architecture make it highly suited for streaming applications.

For this project, the Microgrid architecture was analyzed by studying its performance for a simple benchmark application considered representative of streaming applications. Necessary modifications were made to it so this benchmark would be able to take advantage of the concurrency and scalability inherent to microgrids. This thesis reports on this analysis and its results.
Dedicated to my parents. For their support and belief in me.
Acknowledgments

I would like to thank Dr. Rajiv Ramnath for his support and trust. I would also like to thank Dr. Radu Teodorescu for his extremely useful guidance and patience. A special thanks to Raphael ‘Kena’ Poss from University of Amstredam for helping me every step of my project, from understanding the tools to shaping the research.
Vita
May 2006 .......................... B-Tech, Jawaharlal Nehru Technological University
Sep 2006 to Aug 2011 .......................... System Software Engineer, IBM India
Sep 2011 to May 2013 ..........Graduate Teaching Assistant, The Ohio State University

Fields of Study

Major Field: Computer Science and Engineering
# Table of Contents

Abstract................................................................................................................................. ii

Dedication............................................................................................................................. iii

Acknowledgments................................................................................................................. iv

Vita......................................................................................................................................... v

List of Tables........................................................................................................................... viii

List of Figures........................................................................................................................ ix

Chapter 1: Introduction......................................................................................................... 1
  1.1 Organization of Thesis................................................................................................. 1
  1.2 Problem Statement....................................................................................................... 1
  1.3 Background.................................................................................................................. 2
  1.4 Motivation..................................................................................................................... 3

Chapter 2 Related Research................................................................................................. 6
  2.1 Architecture design....................................................................................................... 6
  2.2 Streaming Applications............................................................................................... 7
  2.3 Streaming applications on new architectures............................................................. 8

Chapter 3 Solution Design................................................................................................. 10
  3.1 MicroGrid architecture............................................................................................... 11
List of Tables

Table 1: New instructions added to the ISA..............................15
Table 2: CPU pipeline utilization comparison..........................29
List of Figures

Figure 1: CPU interconnect. .......................................................13
Figure 2: Register file layout. .....................................................14
Figure 3: Sample code using SL directives..................................17
Figure 4: Transformations for the Livermore 3 kernel...............19
Figure 5: Naive implementation with two levels of thread families....21
Figure 6: Producer consumer interaction with shared register........24
Figure 7: Comprehensive approach..........................................25
Figure 8: Chart comparing CPU pipeline utilization for naive and
strategies.................................................................28
Chapter 1: Introduction

1.1 Organization of Thesis

This thesis is structured into 5 chapters. In chapter 2 we discuss the state of the art of related research in this area. Related projects are listed and their contributions are contrasted.

In chapter 3, the solution design is described in detail. First the Microgrid architecture is discussed in detail followed by the limitations in its design that need to be fixed in order to support a novel class of applications. Finally, necessary modifications to the architecture are described.

In Chapter 4 the results from experimental runs are analyzed and chapter 5 describes the future direction that this project could take.

1.2 Problem Statement

The work done for this thesis strives to modify an existing many-core architecture to be able to run a novel class of application. The architecture studied is called Microgrids [1] and necessary modifications
are made to it so streaming applications can take advantage of the low level parallelism that this architecture provides.

1.3 Background

Computing applications have continued to become ever more demanding for faster and more efficient computing systems. Two factors have largely helped the semiconductor technology to keep up with this increasing demand. One, by constantly scaling up the number of components that can be fit on to a chip at a rate which has been more or less dictated by Moore’s law, which predicts that the number of transistors on a chip would double every two years. And two, by increasing the clock frequency at which these components work and interact with each other.

While the transistor count has been increasing at the same pace, utilizing them for scaling a single CPU’s performance has become increasingly more difficult. The frequency scaling is pushing the limits of heat dissipation. Not only does this affect the direct power consumption but it also increases the cost of cooling the equipment. At the same time, gains by throwing more logic components into a single CPU have been effected with diminishing returns.

To better utilize the larger number of components now available on a chip, CPU design has moved toward incorporating multiple independent
processors on a single chip. All processors available in the market today are multi-core in design with the number of cores ranging from 2 to 16 and even beyond.

From the software perspective, this design causes the least amount of disruption since support for multiple processors was already built into most operating systems as well as multi threaded program libraries.

While in a multitasking scenario such as desktop computing, this design is quite useful as long as there are enough tasks that need CPU resources. However, any single application cannot run any faster than before. To draw an analogy, it is more like waiting at the grocery checkout with a cart full of items. It is going to take you the same amount of time regardless of how many other checkout lanes are open and free.

To solve this problem, a single application needs to be decomposed into smaller activities that could be done in parallel. Existing techniques for extracting such parallelism include using multithreaded programming libraries that create coarse grain threads which are able to run in parallel and communicate with each other when needed.

1.4 Motivation

Extracting scalable parallelism in an application using only coarse
grain multi-threading has its limits. Firstly, it is a labor intensive process which is also highly error prone. The programmer needs to make sure they don’t leave a possibility of a race condition or a deadlock.

Even when done properly, multi threaded programs like this do not scale well because they are still coarse grained and quite big in size to spread across hundreds of cores. Also, they utilize a very high level mechanism for communication and synchronization between parallel threads which goes through the primary memory even though the cores sit next to each other on the same chip.

In order to fix these issues, new architecture design becomes imperative which provides a thread creation mechanism that is inexpensive and also provides a synchronization mechanism which by passes the traditional memory based methods and works through direct communication between the cores. Along with these changes in the architecture, there is a need to incorporate changes at the compiler as well as the programming language level to make the programmer’s life easier. The ultimate goal is to hide all the details of parallelizing an application from the programmer.

While such newer architectures try to extract parallelism from low-level constructs in a general program, they can be further optimized for specific classes of applications so they can take advantage of available
concurrency much more effectively. One such class is streaming applications. There is evidence that streaming media applications are consuming most of the cycles on consumer machines like smartphones. At the same time such applications are fundamental to high performance domains like network routing and cryptography[7].

This project tries to fit streaming applications on a scalable distributed CMP architecture called Microgrids to take advantage of the similarity in structure of the applications with the architecture design.
2.1 Architecture design

New architecture design for many core CPUs has been approached from several directions.

The graphics processor (GPU) has been a popular design. The GPU on today's commodity video cards has evolved into an extremely powerful and flexible processor. Modern graphics architectures provide tremendous memory bandwidth and computational horsepower, with dozens of fully programmable shading units that support vector operations and IEEE floating point precision [13]. Programming environments like CUDA have emerged which let programmers take advantage of this hugely parallel architecture. Applications that perform massive vector operations can realize many orders-of-magnitude improvement in performance over a traditional processor. However, they lack explicit support for communication between the streaming multiprocessors (SMs) on the GPU. Such communication can occur via the global memory of a GPU, but it then requires a barrier synchronization across the SMs of the GPU in order to complete the communication between SMs[14]
Then there are architectures which provide much more lower level support for synchronization primitives by incorporating on-chip networks. One such architecture is Voltron which extends traditional multicore systems by providing a dual-mode scalar operand network to enable efficient inter-core communication and lightweight synchronization [15]. This design is quite flexible from the programmer’s point of view since it places no restriction on which CPUs can communicate with each other. However, this same flexibility requires all CPUs to be connected with each other which makes the scalability of this design difficult.

2.2 Streaming Applications

There is evidence that streaming media applications are already consuming a majority of the CPU cycles on most consumer machines [16]. This kind of prevalence has motivated development of language and compiler support specifically meant for streaming applications.

William Thies et al [7] have developed the StreamIt language with two goals: first, to provide high-level stream abstractions that improve programmer productivity and program robust-ness within the streaming domain, and second, to serve as a common machine language for grid-based processors. The StreamIt compiler also aims to perform stream-specific optimizations to achieve the performance of an expert programmer.
StreamIt differs from other languages in that it imposes a well-defined structure on the streams; all stream graphs are built out of a hierarchical composition of Pipelines, SplitJoins, and FeedbackLoops. The comparison of StreamIt’s structure with arbitrary stream graphs could be likened to the difference between structured control flow and GOTO statements. Though sometimes the structure restricts the expressiveness of the programmer, the gains in robustness, readability, and compiler analysis are immense.

2.3 Streaming applications on new architectures

Efficient and effortless parallelization of streaming applications has received a lot of attention. Multiple efforts have been made to fit this class of applications better on specialized novel architectures.

Manjunath Kudlur, et al [8] describe a compiler technique for planning and orchestrating the execution of streaming applications on multicore platforms. They have targeted the Cell Broadband Engine (BE) architecture. The CellBE [17] includes a 64-bit multithreaded PowerPC processor element (PPE) and eight synergistic processor elements (SPEs), connected by an internal high-bandwidth Element Interconnect Bus (EIB). They describe an integrated unfolding and partitioning technique that unfolds data parallel actors as needed and maximally packs actors onto cores. Next, the actors are assigned to pipeline stages in such a way that all communication is maximally overlapped with
computation on the cores.

Similar work has been done by Udupa et al [9] targeting GPUs programmable using the CUDA framework. They have formulated the problem of scheduling and assignment of pipeline stations to processors as an efficient integer linear program (ILP), which is then solved using ILP solvers. They work around the limitations of the GPU layout, such as limited synchronization within threads and severe penalties for global memory access, to orchestrate pipeline station threads to achieve scalable performance reaching speedups between 1.87X and 36.83X over a single threaded CPU.

When compared to the architectures targeted by the efforts described in this section, the Microgrid architecture not only provides much better mechanism for thread synchronization, but the physical arrangement of the CPUs and their interconnect is a much better match for streaming applications. This distinctive feature of the architecture is the main motivation behind this project.
Chapter 3 Solution Design

For this project, we will use technologies and tools developed as part of the “Apple-CORE” project which stands for *Architecture Paradigms and Programming Languages for Efficient programming of multiple COREs*. Apple-CORE is a project that intends to make multi-core computing mainstream by providing general-purpose, multi-functional multi-cores that are easy to program and work with [12].

To harness the potential of chip multi-processors(CMPs) for scalable, energy-efficient performance in general-purpose computers, the Apple-CORE project has co-designed a general machine model and concurrency control interface with dedicated hardware support for concurrency management across multiple cores. Its SVP interface combines dataflow synchronization with imperative programming, towards the efficient use of parallelism in general-purpose workloads. Its reflection in hardware provides logic able to coordinate single-issue, in-order multi-threaded RISC cores into *computation clusters on chip*, called Microgrids[3]. This hardware design is implemented as a cycle accurate simulator called MGSim. Relevant details about each of these components will be
3.1 MicroGrid architecture

The microgrid architecture is a scalable, parallel CMP design which implements multiple independent CPUs with low level provisions for inter CPU communication. The execution model for this design is such that sequential code is fragmented at compile time and executed out of order while maintaining in-order sequence within each fragment. The CPUs are single issue, in-order execution units. This avoids the complexity in instruction issue and eliminates complex circuit design for speculative execution. The idea being that once large amount of concurrency is available in software, one can scale back on the complexity per core and multiply the number of cores to increase the throughput [3].

Thread families exhibit a parent child relationship. Each executing thread can create a new family of threads which will execute related fragments of code. The only constraint that limits the execution of threads in a family is the dependencies between them. To manage these dependencies, the architecture provides extremely efficient on chip communication

The CPUs are connected using two on chip communication systems. One is a global broadcast bus. This bus is mainly used at thread creation time when the parent thread needs to pass on setup information, initial
parameters and loop invariants to all the threads at one time. It can also be used to collect back aggregated outcomes from the child thread family after all threads finish execution. Since it is a global and a shared resource, it suffers from contention and needs to be arbitrated. But this contention does not affect the scalability or the efficiency of the design a lot because of the relatively infrequent use of the bus.

The second network is a shared-register ring network connecting all CPUs but it only allows communication between adjacent CPUs. There is no addressing or routing on this bus. An outgoing message from one CPU will always be consumed by the next CPU and so on. Fig. 1 shows how the CPUs are connected using these two networks.
Each CPU can interleave multiple threads from several families simultaneously. This is achieved by providing a register file in each CPU that is over provisioned by several times and partitioned on run time between all the threads currently being interleaved on the CPU.

Multiple threads of the same family executing on the same CPU, are allocated overlapping partitions of the register file. The overlap makes it possible for two threads to communicate in the same fashion as two threads communicating between two CPUs using the ring network. This arrangement is shown in Fig.2.
Each such shared register is associated with state information which lets the consumer thread know if the data is ready for consumption. The state information is modified within a clock cycle, so this barrier can be used as an atomic operation providing full protection from race conditions.

Figure 2: Register file layout. The register file within each CPU gets partitioned among all active threads. The partitions overlap so that adjacent threads from the same family share registers.

It's important for the scope of this project to understand how the MicroGrid architecture deals with concurrency control. The architecture adds to the existing ISA a small list of instructions which are the key to concurrency control. Table 1 lists the new instructions added to the ISA.
Table 1: New instructions added to the ISA

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instruction behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cre</td>
<td>Creates a new family of threads</td>
</tr>
<tr>
<td>Swch</td>
<td>Causes a context switch to occur</td>
</tr>
<tr>
<td>Kill</td>
<td>Terminates the thread being executed</td>
</tr>
<tr>
<td>Bsync</td>
<td>Waits for all other threads to terminate</td>
</tr>
<tr>
<td>Brk</td>
<td>Terminates all other threads</td>
</tr>
</tbody>
</table>

The *Cre* instruction broadcasts the thread creation parameters, the global input values and information about the shared parameters between the threads to all the CPUs participating in this family. This is matched with the *Kill* instruction which terminates all the threads in a family. It causes the threads to release all resources allocated to it.

In order to achieve interleaving, each CPU maintains a list of active threads that are ready to execute. With the *Swch* instruction, a thread can explicitly yield the CPU in favor of another thread. This instruction is acted upon in the very first stage of the CPU pipeline, effectively achieving a cycle-by-cycle interleaving. The compiler can explicitly insert a request for a context switch in case of a read operation since these might fail if there is a cache miss. It can also do something similar after a longer operation like a floating point multiplication.
The only way of synchronization between threads in a family is through shared registers in overlapping register partitions. The synchronization is performed using two state bits that are associated with each register which differentiate between the following states: full, empty, waiting-local and waiting-remote [1]. At the start of a thread execution all registers are “wiped” by moving them to the EMPTY state. When the consumer thread issues a read and finds the register still in EMPTY state, it yields the pipeline and gets added to the list of threads waiting on this register.

Context switching and successful synchronization have no overhead in terms of additional pipeline cycles [1]. In Fact all the new instructions can be completed in the first stage of the pipeline as they only control the action of the scheduler [2].

3.2 The SVP model

The Self-adaptive Virtual Processor (SVP) is a programming model based on concurrent and hierarchical composition of homogeneous families of blocking microthreads [3]. This model defines all concurrency and communication in abstract terms and all mapping and scheduling of the threads created is transparent to the programmer. SVP also defines mechanisms for dynamic control of concurrently executing programs and its implementation may be considered as an operating system kernel [3].
#include <stdio.h>

// This thread program ’foo’ increments the value it receives over its input channel, propagates the incremented value to its output channel, then prints a digit on the C standard output.

sl_def( foo, sl_shparm(int, a)) {
    sl_setp(a, sl_getp(a) + 1);
    printf("%d", sl_getp(a));
} sl_enddef

int main(void) {
    // Create a dependent family of 10 threads.
    sl_create(, 0, 10, 1, 0, , foo, sl_sharg(int, x));
    // Push 0 as source value for the dataflow channel.
    sl_seta(x, 0);
    // Wait on termination of the family.
    sl_sync();
    // After the family terminates, the channel endpoint reveals the final value.
    printf("%d\n", sl_geta(x));
    return 0;
}

Figure 3: Sample code using SL directives. A sample “hello world” like program is listed which uses basic SL primitives. A family of threads which communicate with each other using a shared variable

The SL intermediate language provides extensions to C to represent the semantics of an SVP model. It provides the low level primitives for concurrency creation, synchronization, the definition of data flow communication channels and the communication over these channels[4]. The figure shows some of these calls in a simple “hello-world” program for thread creation, synchronization and data flow. As a C extension, SL supports most programs that are also valid in C. Fig. 3 shows a sample program which demonstrates some of SL primitives.
3.3 The MGSim Simulator

MGSim provides a discrete event, cycle-accurate, full-system simulation kernel where all component behaviors down to individual pipeline stages, register file ports, arbitrators, functional units, FIFOs, DDR controller, etc. have their own detailed model. This enables a slightly higher level, and thus faster simulation than a circuit-level simulation [5].

The simulator implements the SVP model in the instruction set of an in-order issue Alpha core. A ‘place’ in this architecture comprises a number of processors configured into a ring network to implement the SVP actions (create, sync, etc) and to support register sharing between adjacent processor’s register files. All delegation in the microgrid is performed by a chip wide network, linking all processors (global bus) [4].

The implementation is done in an object oriented manner which makes it easy to modify or add new modules to the simulator. The simulator also provides multiple debugging tools to step through either the simulator code or the program running on top of the simulator.
3.4 Current Application

Microthreads are currently used to parallelize loop iterations in serial code. Using the SL language extensions, the code is transformed such that a new family of threads is created where each thread executes one iteration of the loop. Such transformation can result in concurrency in not only loops with no intra-iteration dependencies but also when the loop contains loop-carried dependencies. One such transformation of the Livermore 3 kernel is shown in the Fig.4. This transformation is a reduction which can be executed with full concurrency using microthreads.

```c
#define N 65536
int x[N];
int y[N];
int i;
int sum = 0;
for (i = 0; i < N; i++) {
    sum += x[i] * y[i];
}

#define N 65536
int x[N];
int y[N];
family fid;
int sum = 0;
create(fid; 0; N - 1; i)
    sum_loop(sum, x, y)
thread void sum_loop(shared int s,
    const int* x, const int* y)
{
    index i;
    s += x[i] * y[i];
}

#define N 65536
#define N2 64
#define N1 (N / N2)
int x[N];
int y[N];
family fid;
int sum = 0;
create(fid; 0; N2 - 1; i)
    outer_loop(sum, x, y)
thread void outer_loop(shared int s,
    const int* x, const int* y)
{
    index j;
    family fid;
    int local_sum = 0;
    create(fid; local; j; N - 1; N1)
        inner_loop(local_sum, x, y)
    s += local_sum;
}

thread void inner_loop(shared int s, const int* x, const int* y)
{
    index i;
    s += x[i] * y[i];
}
```

Figure 4: Transformations for the Livermore 3 kernel. The first transformation doesn’t introduce any concurrency due to presence of an aggregation. This is overcome in the second transformation using two levels of thread families.
3.5 Pipelined streaming applications

Stream programming is one model that has wide applicability in the multimedia, graphics, and signal processing domains. Streaming models execute as a set of independent actors that explicitly communicate data through channels. Each station acts on an independent piece of data and passes along its output to the next stage like cars in an assembly line. Such applications are inherently concurrent and scale almost perfectly. The scale of concurrency coupled with the forward only data flow makes these applications greatly suited for the Microgrid architecture.

3.6 Naive Implementation of a streaming application on MG

To fit streaming applications on the microgrid architecture out of the box, two levels of thread families need to be created. Individual threads in the first level family will consume one data element from the incoming stream of data. Each such thread will in turn spawn another family of threads. Individual threads belonging to this second level family act as one pipeline station for the streaming application. Fig 5. shows this arrangement. Each station processes the data element and passes along its output to the adjacent thread through the ring network or by extension, the overlapping register segments in the register file. The next thread waits on this incoming data from the previous thread. Once it receives the data, it can go ahead and do its part of the processing.
At any given time, only one thread of each second level family can be running. The previous threads would have finished their execution and the subsequent threads are all waiting for input from the previous threads. Effectively all second level thread families are completely serialized. The concurrency in this method is achieved from placing second level threads on different CPUs. For this reason the scaling is limited to the number of physical CPUs present.

3.7 Comprehensive implementation

The naive approach works as a simple extension to the current use
case of microgrid architecture without any modifications to it. But as illustrated in the previous section, it suffers from some crippling limitations.

In order to truly reflect the streaming paradigm, each pipeline station must be able to act on a different data element at the same time and pass along the output to the next stage. This way true parallelism can be achieved even when all stations are not on distinct CPUs.

However, the design of the microgrid architecture in its current state is unable to support this paradigm completely. The data flow channels for shared data between adjacent threads only synchronizes the consumer thread when the producer thread hasn’t yet produced its output. The producer thread on the other hand never waits before trying to write to the shared register.

This design works because on the SVP platform, a thread family is created every time a for loop is encountered and each thread in the family is responsible for only one iteration of the loop. After the loop is finished the whole thread family is the destroyed. Since the producer never waits to write to the shared space, any more than one transfer between two adjacent threads will have an undetermined result. But since a thread’s life span lasts only one iteration of a loop, the one time usability of the data channel suffices. In fact this limitation makes the design of the architecture
In order for synchronization to work both ways, there were two modifications necessary in the architecture.

- The producer thread must only write in to the shared register if it is in the EMPTY state (earlier this was always assumed to be true)
  - The thread must suspend if the register still hasn’t been consumed.

- After consuming the register, the consumer thread must set the register back to the EMPTY state. In turn waking up the producer thread which might have suspended on it.

These changes are depicted in Figure 6 as a UML sequence diagram.
Figure 6: Producer consumer interaction with shared register: UML sequence diagram showing how the producer and consumer threads interact with the shared register and modify the register's state.

The change is straightforward but it introduces a new structural hazard into the CPU pipeline. Before this change the only stage of the pipeline that could modify the contents of the register file was the Writeback stage. But now since the consumer thread must set the register back to EMPTY after reading the value in from it, the ReadStage of the pipeline can also initiate a modification in the register file.

This problem doesn’t directly affect two adjacent threads of a single thread family since the barriers set by register state check are rigid and not prone
to race condition. However, the microgrid CPU is designed to interleave several threads from various thread families and there could easily be two threads with instructions in proper slots in the pipeline to hit this hazard.

In order to fix this issue, the write port to the CPU’s register file had to be arbitrated. In case of a contention, the write back stage is given preference because the data that is written back could potentially travel through the pipeline forwarding mechanism and serve the consumer thread before it is made to go to a waiting state.

Figure 7: Comprehensive approach. Fully functional pipeline with multiple instantiation for scaling is shown.

With the changes in place, the data forwarding channel can now support
multiple transfers before the thread family is killed and the streaming paradigm can be represented fully in the microgrid architecture. However, now the limit on the scaling is placed by the number of pipeline stages that the streaming application could be divided in. In order to scale more effectively, here too, a two stage policy must be followed. The implementation is described in Fig.6.
Chapter 4: Results and Analysis

4.1 Comparing the naive and comprehensive approaches

The results presented in this section were measured using a cycle-accurate implementation of a cluster of SVP cores. This software emulator is based on the specification of the model as described in the previous section.

The benchmark that was used to measure the performance parameters was a simple program that acted on a series of 1000 floating point data elements. The actions were a mix of floating point operations and integer operations with sufficient loads and stores. The operations were divided into two pipeline stages.

The average pipeline utilization of all CPUs when the benchmark was run in the naive fashion was compared with the same benchmark run using the comprehensive changes to the architecture. Data was collected with number of cores varying between 4, 8, 16 and 32.

The results are presented in table 2
The trend shown in Fig. 7 clearly shows that there is better pipeline utilization of the CPU in case of the comprehensive approach. This increase can be attributed to the fact that once a thread is created for the purpose of one application pipeline, it gets to work on many more data elements instead of just one as was the case with the naïve approach. This increases the CPU utilization by hiding memory latencies more effectively. Also, since the comprehensive approach can extract concurrency even with multiple stages of the application running on the same CPU, it creates more opportunity for thread interleaving which increases the CPU utilization further.
<table>
<thead>
<tr>
<th>Cores / Pipeline utilization</th>
<th>Naive approach</th>
<th>Comprehensive approach</th>
<th>% gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>72.15</td>
<td>69.03</td>
<td>-4.32%</td>
</tr>
<tr>
<td>8</td>
<td>69.19</td>
<td>70.623</td>
<td>2.07%</td>
</tr>
<tr>
<td>16</td>
<td>63.64</td>
<td>71.154</td>
<td>11.81%</td>
</tr>
<tr>
<td>32</td>
<td>64.195</td>
<td>72.039</td>
<td>12.22%</td>
</tr>
<tr>
<td>Average</td>
<td>67.29375</td>
<td>70.7115</td>
<td>5.08%</td>
</tr>
</tbody>
</table>

Table 2: CPU pipeline utilization comparison. Comparing CPU pipeline utilization for naive and comprehensive implementation strategies

4.2 Comparison with GPUs

The current state of the art for streaming applications is to utilize the concurrency presented by GPUs. However GPU programming suffers from inherent problems like branch diversion issue [18] and high cost of synchronization between threads in different SPs [14]. Such issues arise mainly from the fact that processing units in GPUs are not completely independent and within a “warp” are more like SIMD units. The microgrid architecture doesn’t suffer from these issues because each CPU comes with its own instruction pointer and a control unit.

Another issue facing GPUs is the co-processor configuration that they run in. This causes latency issues when data needs to be transferred. Microgrids, on the other hand are capable of running all services of the
operating system including handling interrupts and I/O devices [11]. This helps in making the memory access more direct.
Chapter 5 Future Work and Conclusion

Current work shows that the basic layout and interconnect design are a good match of streaming applications workloads and that with few modifications to the architecture, significant gains in concurrency can be achieved. However, to fit a wider variety of applications, further changes will be required.

5.1 Future work

As described in section 2.2, streaming applications can be reduced to three basic constructs; the pipeline, the split-join and the feedback loop. The current implementation represents the simple pipeline where there is a forward only communication between the producers and consumers. In order to support the full spectrum of streaming applications the other two constructs will have to be incorporated.

With the hierarchical design of thread families, the split join constructs should be fairly straightforward to implement however it might suffer from performance issues similar to the naïve implementation.
The feedback loop on the other hand is possible only with some more changes to the architecture. For example the on chip network could be made to be reconfigurable on the run time.

5.2 Buffered communication

In the current implementation the data flow channel is only one word wide and more importantly can carry only a single message. That is, if the consumer isn’t quick enough to consume the incoming message, the producer cannot proceed further. This way the whole pipeline works at the speed of the slowest station. It will be helpful if the data channel could buffer some messages. This way slow stations can get a chance to catch up and the whole pipeline could run faster.

While the buffered scheme is a definite improvement, it comes with its overhead. There could be cases where non buffered scheme could be perfectly useful and not incur the overhead. The decision to choose between the two schemes could be made at the compile time or even at the run time.

5.3 StreamIT integration

At this time all the transformation of serial code to parallel pipeline stations was done manually. This process could be automated using
modified compilers that use annotated language extensions to detect and optimally parallelize streaming applications. One example of such a compiler is the StreamIt compiler and language extensions [7]. The threaded code from such compiler will need to be ported on to the SVP platform in order to run it on the microgrid architecture.

5.2 Conclusion

As part of this project, the microgrid architecture was modified in ways necessary to make it better suited to scale streaming applications. For the benchmark application, the modifications were non-disruptive with little or no impact on the performance or scalability of the architecture. There were significant gains in resource utilization and the gains scaled well with increasing number of CPUs.
References


[14] W.-C. Feng and S. Xiao, “To GPU Synchronize or not GPU Synchronize?,” in Int'l


