Optimizing The Performance of Streaming Numerical Kernels
On The IBM Blue Gene/P PowerPC 450

Thesis by
Tareq Majed Yasin Malas

In Partial Fulfillment of the Requirements
For the Degree of
Master of Science

King Abdullah University of Science and Technology, Thuwal,
Kingdom of Saudi Arabia
July, 2011
The thesis of Tareq Malas is approved by the examination committee.

Committee Chair (Thesis Supervisor): Prof. David Keyes
Committee Member: Prof. Mikhail Moshkov
Committee Member: Dr. Aron Ahmadia

King Abdullah University of Science and Technology

2011
ABSTRACT

Optimizing The Performance of Streaming Numerical Kernels
On The IBM Blue Gene/P PowerPC 450

Tareq Majed Yasin Malas

Several emerging petascale architectures use energy-efficient processors with vectorized computational units and in-order thread processing. On these architectures the sustained performance of streaming numerical kernels, ubiquitous in the solution of partial differential equations, represents a formidable challenge despite the regularity of memory access. Sophisticated optimization techniques beyond the capabilities of modern compilers are required to fully utilize the Central Processing Unit (CPU).

The aim of the work presented here is to improve the performance of streaming numerical kernels on high performance architectures by developing efficient algorithms to utilize the vectorized floating point units. The importance of the development time demands the creation of tools to enable simple yet direct development in assembly to utilize the power-efficient cores featuring in-order execution and multiple-issue units.

We implement several stencil kernels for a variety of cached memory scenarios using our Python instruction simulation and generation tool. Our technique simplifies the development of efficient assembly code for the IBM Blue Gene/P supercomputer’s PowerPC 450. This enables us to perform high-level design, construction, verifica-
tion, and simulation on a subset of the CPU’s instruction set. Our framework has the capability to implement streaming numerical kernels on current and future high performance architectures. Finally, we present several automatically generated implementations, including a 27-point stencil achieving a 1.7x speedup over the best previously published results.
ACKNOWLEDGMENTS

I owe my deepest gratitude to my mentor, Aron Ahmadia, for his encouragement, guidance, support, and contribution in our research work since the beginning. He enabled me to develop a better understanding of Scientific Computing research. With his amazing leadership skills, he formed our research group and ensured flawless and synergistic teamwork.

Many thanks to John Gunnels, the manager of High Performance Computing at IBM T.J. Watson Research Center. He has made available his support in a number of ways. He provided guidance and support to improve the performance of our research code through our weekly meetings. His long experience in High Performance Computing was essential in achieving our research results and understanding the resulting performance. Moreover, I am grateful for his help in reviewing the draft of this thesis.

I am in debt to Jed Brown from ETH for his contribution, utilizing his professional coding experience, in our research group. His proof-of-concept PowerPC 450 simulator provided great assistance to our research. Also, his bright ideas in the kernel’s algorithm were essential in achieving our successful results.

I am heartily thankful to Professor David Keyes. He provided all the support required to make our research group possible. Even with his commitments as the founding dean of the MCSE Division at KAUST, he generously granted me some of his time through our various meetings. I am honored to be one of his graduate students.
I am most indebted to my parents for all their time and effort raising me. Their sustained encouragement gave me the power to pursue knowledge through higher education. Also, I am grateful for their patience during my busy periods when I could not visit them.

I am indebted to my brothers and sister for helping me in my school courses, an essential factor that enabled me to carry out my undergraduate and graduate education. Moreover, I am thankful for their patience, support, and encouragement during my education in the undergraduate and Masters degree.

Finally, I am grateful to Andy Ray Terrel for his helpful commentary on an early draft of this thesis. I am also indebted to Andrew Winfer for his support in conducting our numerical experiments on the Shaheen Blue Gene/P system at the KAUST Supercomputing Laboratory, and to Vernon Austel for his assistance in running experiments on the Blue Gene/P system at IBM’s Watson Research Center.
TABLE OF CONTENTS

List of Abbreviations ........................................... 10
List of Illustrations ........................................... 11
List of Tables .................................................... 12

I Introduction .................................................... 14
  I.1 Motivation ............................................... 14
  I.2 Background ............................................. 16

II Related Work ................................................ 18

III Implementation Considerations ......................... 21
  III.1 Stencil Operators ...................................... 21
  III.2 PowerPC 450 ........................................... 23
  III.3 Instruction Scheduling Optimization ............... 26

IV Implementation ............................................. 32
  IV.1 STREAM benchmark on Shaheen ....................... 32
  IV.2 C and Fortran .......................................... 33
  IV.3 Synthetic Assembly ..................................... 34
    IV.3.1 Kernels Design ................................... 35
    IV.3.2 Unroll-and-Jam ................................... 37
LIST OF ABBREVIATIONS

API Application Programming Interface
ArBB Array Building Blocks
CPU Central Processing Unit
CUDA Compute Unified Device Architecture
DAG Directed Acyclic Graph
FIFO First In First Out
FMA Floating point Multiply-Add
FPR Floating Point Registers
FPU Floating Point Unit
GPR General Purpose Registers
GPU Graphics Processing Unit
ILP Integer Linear Programming
LRU Least Recently Used
LSU Load Store Unit
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MG</td>
<td>Multi Grid</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface</td>
</tr>
<tr>
<td>NAS</td>
<td>NASA Advanced Supercomputing</td>
</tr>
<tr>
<td>NUMA</td>
<td>Non-Uniform Memory Access</td>
</tr>
<tr>
<td>OpenMP</td>
<td>Open Multi-Processing</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>SIMT</td>
<td>Single Instruction Multiple Thread</td>
</tr>
<tr>
<td>SMPs</td>
<td>Symmetric Processors</td>
</tr>
<tr>
<td>WENO</td>
<td>Weighted Essentially Non-Oscillatory</td>
</tr>
</tbody>
</table>
LIST OF ILLUSTRATIONS

II.1  Unroll-and-jam example for a 3-dimensional array copy operation  . 20
    (a)  Regular loop  . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
    (b)  Unrolled-and-jammed loop  . . . . . . . . . . . . . . . . . . 20

III.1  7-point stencil operator  . . . . . . . . . . . . . . . . . . . . . . . . . 22
III.2  27-point stencil operator  . . . . . . . . . . . . . . . . . . . . . . . 23
III.3  Example of DAG representation of the instructions  . . . . . . . . 27
III.4  A 0/1 array of size $N \times M$ representing the scheduling variables . . . 28

IV.1  SIMD stencil computations on 1D streams  . . . . . . . . . . . . . . 36
IV.2  The components of the code generation framework in this work  . . . 41

V.1  3-point stencil performance with load-copy kernel  . . . . . . . . . . 44
V.2  7-point stencil performance comparison between mutate-mutate and load-copy kernels  . . . . . . . . . . . . . . . . . . . . . 46
V.3  7-point stencil performance comparison  . . . . . . . . . . . . . . . . 47
V.4  27-point stencil performance with mutate-mutate kernel  . . . . . . 48
V.5  27-point stencil performance comparison  . . . . . . . . . . . . . . . . 49
LIST OF TABLES

IV.1 STREAM performance (MB/s) on PowerPC 450 using 1, 2, and 4 threads ........................................... 33
IV.2 STREAM performance (MB/s) on PowerPC 450 running 2 and 4 MPI processes ........................................... 33
IV.3 Resource usage per stencil of mm and lc .......................................................... 37
IV.4 Computational requirements .......................................................... 40
V.1 Predictions vs. observations for in-L1 and streaming performance .......................................................... 51
CHAPTER I

Introduction

I.1 Motivation

As computational science soars past the petascale to exascale, a large number of applications continue to achieve disappointingly small fractions of the sustained performance capability of new machines. Although the traditional focus in scientific computing is on massively parallel or even multicore scalability, in many cases this shortcoming in performance stems from issues at the single processor/thread level. The ‘many-core’ revolution brings simpler, slower, more power-efficient cores with vectorized floating point units in large numbers to a single microprocessor. At each stage of such a design, tradeoffs are made that complicate performance optimization for the compiler and programmer in exchange for improved power and die efficiency from the hardware. For example, out-of-order execution logic allows a microprocessor to reorder instruction execution on the fly to avoid pipeline and data hazards. When out-of-order logic is removed to save silicon and power, the responsibility for avoiding these hazards is returned to the compiler and programmer. In the same vein, a wide vector processing unit can significantly augment the floating point performance capabilities of a processing core at the expense of the single-element mappings between
input and output in performant code. Such vector units also incur greater bandwidth demands for a given level of performance, as measured by percentage of theoretical peak. Graphics processing units provide a set of compute semantics similar to a traditional Single Instruction Multiple Data (SIMD) vector processing unit with Single Instruction Multiple Thread (SIMT), but still execute in-order and require vectorized instruction interlacing to achieve optimal performance. The trend to improve efficiency of performance with wider vector units and in-order execution units is observable in the architectures of the IBM Blue Gene/P PowerPC 450 [1], the Cell Broadband Engine Architecture [2], Intel’s Larrabee architecture [3], and NVIDIA’s Tesla Graphics Processing Unit (GPU) [4].

The characteristics of Blue Gene/P that motivate the work in this thesis will persist in the processor cores of exascale systems. The fundamental challenge of the exascale relative to petascale is electrical power [5]. An optimistic benchmark (goal) for a petascale system is the continuous consumption of somewhat over a MegaWatt of electrical power, which represents the average continuous power consumption of roughly one thousand people in an OECD country (1.4 kW per person). A naïve scaling from peta to exa of the hardware resources known today implies that operation of a single exascale system would displace one million people from the power grid. Power reductions relative to delivered flop/s of factors of one to two orders of magnitude are expected en route to the exascale, which means more threads instead of faster-executing threads, power growing roughly as the cube of the clock frequency. It also means much less memory and memory bandwidth per thread because the movement of data over copper interconnects consumes much more power than the operations on that data in registers. Mathematical formulations of problems and algorithms to implement them will be rewritten to increase arithmetic intensity. Implementations in hardware will have to be made without some of today’s popular power-intensive performance optimizations.
In general terms, we may expect less memory per thread, less memory bandwidth per thread, and more threads per fixed data set size, creating an emphasis on strong scaling within a shared-memory unit. We expect larger grain sizes of SIMDization and high penalization of reads and writes from main memory.

I.2 Background

We define streaming numerical kernels as small, cycle-intensive regions of a program where, for a given $n$ bytes of data accessed in a sequential fashion, $O(n)$ computational operations are required. Streaming numerical kernels are generally considered to be memory-bandwidth bound on most common architectures due to their low arithmetic intensity. The actual performance picture is substantially more complicated on high performance computing microprocessors, with constraints on computational performance coming from software limitations in the expressiveness of standard C and Fortran when targeting SIMD processors and a host of hardware bottlenecks and constraints, from the number of available floating point registers, to the available instructions for streaming memory into SIMD registers, to the latency and throughput of buses between the multiple levels of the memory hierarchy.

In this thesis, we focus upon stencil operators, a subset of streaming kernels that define computations performed over a local neighborhood of points in a spatial multi-dimensional grid. They are commonly found in partial differential equation solver codes as finite-difference discretizations of continuous differential operators. Perhaps the most well-known of these is the 7-point stencil, which usually arises as a finite difference discretization of the Laplace kernel on structured grids. Although modern numerical methods and discretization schemes have diminished this operator’s relevance for many problems as a full-fledged numerical solver, it is still a cycle-intensive subcomponent in several important scientific applications such as Krylov iterations of
Poisson terms in pressure corrections, gravitation, electrostatics, etc. \[6\] and adaptive mesh refinement methods \[7\]. Other important stencils include the 27-point stencil and the 3-point stencil. The 27-point stencil arises when cross-terms are needed such as in the NASA Advanced Supercomputing (NAS) parallel Multi Grid (MG) benchmark, which solves a Poisson problem using a V-cycle multigrid method with the stencil operator \[8\]. The 3-point stencil is a one-dimensional stencil which serves as a building block for the higher-order stencils and is useful on its own in tensor product formulations such as the solution of Riemann kernels in hyperbolic systems of equations.

We concentrate on the Blue Gene/P architecture \[1\] for a number of reasons. The forward-looking design of the Blue Gene series, with its power-saving and ultra-scaling properties exemplifies some of the characteristics that we feel will be common in exascale systems. Blue Gene/P has SIMD registers, a slow clock rate (850 MHz), an in-order, narrow superscalar execution path, and is constructed to be highly reliable and power-efficient, holding several slots in the top 10 of the GREEN500 \[9\] list as recently as 2009. While the system is several years old, it is still used for cutting-edge science, as is evidenced by the continued presence of Blue Gene systems as winners and finalists in the Gordon Bell competition \[10, 11, 12\] as well as receiving awards as prestigious as The National Medal of Technology and Innovation in 2009.
CHAPTER II

Related Work

As stencil operators have been identified as an important component of many scientific computing applications, a good deal of effort has been spent attempting to optimize their performance. Recently, [13] optimized the performance of a 7-point stencil and a Lattice Boltzmann method on CPUs and GPUs over three-dimensional grids. They performed spatial and temporal blocking, named 3.5D blocking, to decrease the memory bandwidth requirements of their memory bound problems. An earlier yet more comprehensive work by Datta [14] constructed an auto-tuning framework to optimize the 7- and the 27-point stencils and the Gauss-Seidel Red-Black Helmholtz kernel. Datta’s work was performed on diverse multicore architectures modern at the time. He achieved good performance employing a search over a variety of algorithmic and architecture-targeting techniques including common subexpression elimination and Non-Uniform Memory Access (NUMA)-aware allocations. It is interesting to note that aside from register blocking and common subexpression elimination, Datta’s techniques were ineffective in improving the performance of stencil operators on the Blue Gene/P platform. This was attributed in part to the in-order-execution architecture of the PowerPC 450 processor.

A common domain-specific technique in practice is time skewing [15] [16]. Unfor-
tunately, time skewing is not generalizable and has limited application as it does not allow other computations between time steps to occur.

There have been several other recent studies of performance improvements in stencil operators in the literature [17-18-19], including core-level optimization techniques. The majority of previous work focuses on spatial and temporal blocking of memory access: Frigo provides a cache-oblivious algorithm for general n-dimensional stencil computation in [20], whereas in [21] Wellein performs stencil multi-core optimizations using cache blocking in time (temporal blocking). Kamil conducted a study on the impact of modern memory subsystems on three-dimensional stencils [22], and proposed two cache optimization strategies for stencil computation in [23], as well as an auto-tuning framework for code generation in [24]. Kamil’s framework accepts the stencil’s kernel in Fortran or then converts it to a tuned version in Fortran, C, or Compute Unified Device Architecture (CUDA). In [25], a software synthesis approach was proposed to generate optimized stencil code. Machine Learning is utilized in [26] to tune the parameters of the optimization techniques for the 7- and the 27-point stencil. In [27] tiling in space is utilized to improve the performance by improving the spatial locality of the stencil computations.

Several emerging frameworks are facilitating the development of efficient high performance code without having to go down to the assembly level, at least not directly. These implementations are largely motivated by the difficulties involved in utilizing vectorized Floating Point Unit (FPU) and other advanced features in the processor. CorePy [28], a Python implementation similar to our approach, provides a code synthesis package with Application Programming Interface (API) to develop high performance applications by utilizing the low-level features of the processor that are usually hidden by the programming languages. Intel has introduced a new dynamic compilation framework, Array Building Blocks (ArBB) [29], a high level approach to automatically using the SIMD units on Intel processors without targeting a specific
Several techniques are developed in the literature to address the alignment problems in utilizing the SIMD capabilities of the modern processors. In [30] the alignment is handled by using an algorithm to reorganize the data in the registers to satisfy the alignment constraints of the processor. A compilation technique for data layout transformation is proposed in [31] to have the data aligned properly to be used with the SIMD without problems.

In this work we focus on using code synthesis to manually make use of unroll-and-jam, an optimization technique that creates tiling on multiply nested loops through a two-step procedure [32, 33]. Unroll-and-jam combines two well-known techniques, loop unrolling on the outer loop to create two inner loops, then loop fusion, or “jamming,” to combine the two inner loops into a single loop. Figure II.1 shows example of unroll-and-jam for a copy operation between two 3-dimensional arrays, where each of the two outer most loops is unrolled once. This technique can work well for 3-dimensional local operators because it promotes register reuse and can increase effective arithmetic intensity, though it requires a large number of registers available to work effectively.

```
for i = 1 to N
    for j = 1 to N
        for k = 1 to N
            A(i,j,k) = R(i,j,k)
        endfor
    endfor
endfor
```

(a) Regular loop

```
for i = 1 to N step 2
    for j = 1 to N step 2
        for k = 1 to N
            A(i,j,k) = R(i,j,k)
            A(i+1,j,k) = R(i+1,j,k)
            A(i,j+1,k) = R(i,j+1,k)
            A(i+1,j+1,k) = R(i+1,j+1,k)
        endfor
    endfor
endfor
```

(b) Unrolled-and-jammed loop

Figure II.1: Unroll-and-jam example for a 3-dimensional array copy operation
CHAPTER III

Implementation Considerations

III.1 Stencil Operators

A 3-D stencil is a linear operator on $\mathbb{R}^{MNP}$, the space of scalar fields on a Cartesian grid of dimension $M \times N \times P$. Apart from some remarks in Chapter [VI] we assume throughout this thesis that the operator does not vary with the location in the grid, as is typical for problems with regular mesh spacing and space-invariant physical properties such as constant diffusion. The input $A$ and output $R$ are conventionally stored in a one-dimensional array using lexicographic ordering. We choose a C-style ordering convention so that an entry $a_{i,j,k}$ of $A$ has flattened index $(iN + j)P + k$, with zero-based indexing. We assume that the arrays $A$ and $R$ are aligned to 16 byte memory boundaries. Some of our experimental kernels assume an “odd” alignment, where the start of either the $A$ or $R$ array is set to an offset of 8 bytes from a 16 byte alignment. We could generalize the alignment requirements to 8 bytes at almost no computational cost by adding some logic to the prologue of the kernels.

Formally, the 3-point stencil operator defines a linear mapping from a weighted sum of three consecutive elements of $A$ to one element in $R$:

$$r_{i,j,k} = w_0 \ast a_{i,j,k-1} + w_1 \ast a_{i,j,k} + w_2 \ast a_{i,j,k+1}$$
We further assume certain symmetries in the stencils that are typical of self-adjoint problems, there that \( w_0 = w_2 \). The effect of this assumption is that we require fewer registers for storing the \( w \) coefficients of the operator, allowing us to unroll the problem further; however, this assumption is not applicable to convective problems.

The 7-point stencil (Figure III.1) defines the result at \( r_{i,j,k} \) as a linear combination of the input \( a_{i,j,k} \) and its six three-dimensional neighbors with Manhattan distance one. The 27-point stencil uses a linear combination of the set of 26 neighbors with Chebyshev distance one. The boundary values \( r_{i,j,k} \) with \( i \in \{0, M-1\} \), \( j \in \{0, N-1\} \), or \( k \in \{0, P-1\} \) are not written as is standard for Dirichlet boundary conditions. Other boundary conditions would apply a different one-sided stencil at these locations which is cheap even without highly optimized code.

![Figure III.1: 7-point stencil operator](image)

The 27-point stencil (Figure III.2) can be seen as the summation over nine independent 3-point stencil operators into a single result. We assume symmetry along but not between the three dimensions, leading to 8 unique weight coefficients. The symmetric 7-point stencil operator has 4 unique weight coefficients.
III.2 PowerPC 450

Designed for delivering power-efficient floating point computations, the nodes in a Blue Gene/P system are four-way Symmetric Processors (SMPs) comprised of PowerPC 450 processing cores [34]. The processing cores possess a modest superscalar architecture capable of issuing a SIMD floating point instruction in parallel with various integer and load/store instructions. Each core has an independent register file containing 32 4-byte general purpose registers and 32 16-byte SIMD floating point registers which are operated on by a pair of fused floating point units. A multiplexing unit on each end of this chained floating point pipeline enables a rich combination of parallel, copy, and cross semantics in the SIMD floating point operation set. This flexibility in the multiplexing unit can enhance computational efficiency by replacing the need for independent copy and swap operations on the floating point registers. To provide backward compatibility as well as some additional functionality, these 16-byte SIMD floating point registers are divided into independently addressable, 8-byte primary and secondary registers wherein non-SIMD floating point instructions operate transparently on the primary half of each SIMD register.

An individual PowerPC 450 core has its own 64KB L1 cache divided evenly into
32 KB for instructions and 32KB for data. The L1 cache uses a round-robin (First In First Out (FIFO)) replacement policy in 16 sets, each with 64-way set-associativity. The L1 cache line is 32 bytes in size.

Every core also has its own private prefetch unit, designated as the L2 cache, “between” the L1 and the L3. In the default configuration, each PowerPC 450 core can support up to 5 “deep fetching” streams or up to 7 shallower streams. These values stem from the fact that the L2 prefetch cache has 15 128-byte entries. If the system is fetching two lines ahead (such settings can be configured at boot time), each stream occupies three positions, one current and two “future,” while a shallower prefetch lowers the occupancy to two per stream. The final level of cache is an 8 MB L3 cache, shared among the four cores which features a Least Recently Used (LRU) replacement policy, with 8-way set associativity and 128-byte lines.

On this architecture the desired scenario in highly performant numerical codes is the dispatch of a SIMD floating point instruction every cycle (in particular, a Floating point Multiply-Add (FMA)), with any load or store involving one of the floating point registers issued in parallel, as inputs are streamed in and results are streamed out. Floating point instructions can be retired one per cycle, yielding a peak computational throughput of one (SIMD) FMA per cycle, leading to a theoretical peak of 3.4 GFlops/s per core. Blue Gene/P floating point load instructions, whether they be SIMD or non-SIMD, can be retired every other cycle, leading to an effective bandwidth to the L1 of 8 bytes a cycle for aligned 16-byte SIMD loads (non-aligned loads result in a significant performance penalty) and 4 bytes a cycle otherwise. As a consequence of the instruction costs, no kernel can achieve peak floating point performance if it requires a ratio of load to SIMD floating point instructions greater than 0.5. It is important to ensure packed “quad-word” SIMD loads occur on 16 byte aligned memory boundaries on the PowerPC 450 to avoid performance penalties that ensue from the resulting hardware interrupt.
An important consideration for achieving high throughput performance on modern floating point units is pipeline latency, the number of cycles that must transpire between accesses to an operand being written or loaded into in order to avoid pipeline hazards (and their consequent stalls). Floating point computations on the PowerPC 450 have a latency of approximately 5 cycles, whereas double-precision loads from the L1 require at least 4 cycles and those from the L2 effectively require 15 cycles [1]. Latency measurements when fulfilling a load request from the L3 or DDR memory banks are less precise: we assume an additional 50 cycle average latency penalty for all loads outside the L1 that hit in the L3.

In the event of an L1 cache miss, up to three concurrent requests for memory beyond the L1 can execute (an L1 cache miss while 3 requests are “in-flight” will cause a stall until one of the requests to the L1 has been fulfilled). Without assistance from the L2 cache, this leads to a return of 96 bytes (three 32-byte lines) every 56 cycles (50 cycles of memory latency + 6 cycles of instruction latency), for an effective bandwidth of approximately 1.7 bytes/cycle. This architectural characteristic is important in our thesis, as L3-confined kernels with a limited number of streams can effectively utilize the L2 prefetch cache and realize as much as 4.6 bytes/cycle bandwidth per core, while those not so constrained will pay the indicated bandwidth penalty.

The PPC450 is an in-order unit with regards to floating point instruction execution. An important consequence is that a poorly implemented instruction stream featuring many non-interleaved load/store or floating point operations will suffer from frequent structural hazard stalls with utilization of only one of the units. Conversely, this in-order nature makes the result of efforts to schedule and bundle instructions easier to understand and extend.
III.3 Instruction Scheduling Optimization

We wish to minimize the required number of cycles to execute a given code block composed of PowerPC 450 assembly instructions. This requires scheduling (reordering) the instructions of the code block to avoid the structural and data hazards, described in the previous section. In this section we present the theoretical formulation for instruction scheduling technologies and constraint to the maximum allowed register allocation for a given code block.

We formulate the Integer Linear Programming (ILP) optimization problem. Our work is based on the formulation of [35] that considers optimizations combining both register allocation and instruction scheduling of architectures with multi-issue pipelines. To account for multi-cycle instructions, we include parts of the formulation at [36]. We consider two minor changes to these approaches. First, we consider the two separate sets of registers of the PowerPC 450, the General Purpose Registers (GPR), and the Floating Point Registers (FPR). Second, we account for the instructions that use the Load Store Unit (LSU) occupying the pipeline for two cycles.

We consider a code block composed of $N$ instructions initially ordered as $I = \{I_1, I_2, I_3, ..., I_N\}$. A common approach to represent the data dependencies of these instructions is to use a Directed Acyclic Graph (DAG). Given a graph $G(V, E)$, the nodes of the graph ($V$) represent the instructions whereas the directed edges ($E$) represent the dependencies between them. Figure III.3 shows an example of a DAG representing a sequence of instructions. Edges represent data dependencies between instructions. Each read-after-write data dependency of an instruction $I_j$ on an instruction $I_i$ is represented by a weighted edge $e_{ij}$. This weighted edge corresponds to the number of cycles needed by an instruction $I_i$ to produce the results required by an instruction $I_j$. The weights associated with Write-after-read and write-after-write data dependencies are set to one.

The PowerPC 450 processor has one LSU and one FPU. This allows the processor
I_1 : ld fa, rx, ry  
I_2 : ld fb, rx, ry  
I_3 : sub fd, fa, fb  
I_4 : ld fb, rt, ry  
I_5 : mul fe, fb, fd  
I_6 : st fe, rz, ry  
I_7 : mul fc, fd, fc  
I_8 : st fc, rv, ry  

(a) Instruction sequence  

Figure III.3: Example of DAG representation of the instructions

to execute a maximum of one floating point operation per cycle and one load/store operation every two cycles (each operation using the LSU consumes 2 cycles). The instructions use either the LSU (I^{LSU} ∈ LSU) or the other computation resources of the processor including the FPU (I^{FPU} ∈ FPU). A critical path, C, in the DAG is a path in the graph on which the sum of its edges’ weights attains the maximum. We can compute the lower bound (L_{bound}) to run the instructions I as follows:

\[ L_{bound}(I) = \max\{C, 2 * I^{LSU}, I^{FPU}\} \quad (III.1) \]

We can compute an upper bound U_{bound}(I) by simulating the number of cycles to execute the scheduled instructions. If U_{bound}(I) = L_{bound}(I) we can generate an optimal schedule.

We define the 0/1 optimization variables x_i^j, having the value 1 to represent that the instruction I_i is scheduled at a given cycle c^j and 0 otherwise. These variables are represented in the array shown in Figure III.4, where M represents the total number
of the cycles.

\[
\begin{array}{cccccc}
I_1 & c^1 & c^2 & c^3 & \ldots & c^M \\
I_2 & x_1 & x_2^1 & x_3^3 & \ldots & x_M^M \\
I_3 & x_1 & x_2^2 & x_3^2 & \ldots & x_M^2 \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
I_N & x_1 & x_2^N & x_3^N & \ldots & x_M^N \\
\end{array}
\]

Figure III.4: A 0/1 array of size \(N \times M\) representing the scheduling variables

The first constraint in our optimization is to force each instruction to be scheduled only once in the code block. Formally:

\[
\sum_{j=1}^{M} x_i^j = 1, \forall i \in \{1, 2, ..., N\}
\]

The PowerPC 450 processor can execute a maximum of one floating point operation every cycle and one load/store operation every two cycles. This imposes the following constraints:

\[
\sum_{i: I_i \in I_{FPU}} x_i^j \leq 1, \forall j \in \{1, 2, ..., M\}
\]

\[
\sum_{i: I_i \in I_{LSU}} (x_i^j + x_i^{j+1}) \leq 1, \forall j \in \{1, 2, ..., M - 1\}
\]

Finally, to maintain the correctness of the code’s results we enforce dependency constraints:

\[
\sum_{k=1}^{M} (k \cdot x_i^k) - \sum_{k=1}^{M} (k \cdot x_j^k) + e_{ij} + 1 \leq 0, \forall i, j : e_{ij} \in E
\]

Instruction latency to write on a GPR is 1 cycle \((e_{ij} = 1)\). The latency for writing on an FPR is 5 cycles for \(I_i \in I_{FPU}\) and at least 4 cycles for \(I_i \in I_{LSU}\). Load instructions have higher latency when the loaded data is present in the L3 cache or the RAM. This can be considered in future formulations to maximize the number of
cycles between loading the data at an instruction $I_i$ and using it by maximizing $e_{ij}$ in the objective.

We wish also to constrain the maximum number of allocated registers in a given code block. All the registers are assumed to be allocated and released within the code block. An instruction $I_i$ scheduled at a cycle $c^j$ allocates a register $r$ by writing on it. The register $r$ is released (deallocated) at a cycle $c^k$ by the last instruction reading from it. The life span of the register $r$ is defined to be from cycle $c^j$ to cycle $c^k$ inclusive.

We define two sets of 0/1 register optimization variables $g_{ij}^i$ and $f_{ij}^i$ to represent the usage of the GPR and the FPR, respectively. Each of these variables belongs to an array of the same size as the array in Figure III.4. The value of $g_{ij}^i$ is set to 1 during the life span of a register in the GPR modified by the instruction $I_i$ scheduled at the cycle $c^j$ and last read by an instruction scheduled at the cycle $c^k$, that is $g_{ij}^i = 1$ when $j \leq z \leq k$ and zero otherwise. The same applies to the variables $f_{ij}^i$ to represent the FPR register allocation.

To compute the values of $g_{ij}^i$ and $f_{ij}^i$, we define the temporary variables $\hat{g}_i^p$ and $\hat{f}_i^p$. Let $K_i$ be the number of instructions reading the from the register allocated by the instruction $I_i$. These temporary variables are computed as follows:

\[
\hat{f}_i^p = K \times \sum_{z=1}^{p} x^z_i - \sum_{\forall j; e_{ij} \in E} \left( \sum_{z=1}^{p} x^z_j \right), \forall i : I_i \text{ writes on FPR} \tag{III.6}
\]

\[
\hat{g}_i^p = K \times \sum_{z=1}^{p} x^z_i - \sum_{\forall j; e_{ij} \in E} \left( \sum_{z=1}^{p} x^z_j \right), \forall i : I_i \text{ writes on GPR} \tag{III.7}
\]

Our optimization variables $g_{ij}^i$ and $f_{ij}^i$ will equal to 1 only when $\hat{g}_i^p > 0$ and $\hat{f}_i^p > 0$, respectively.
respectively. This can be formulated as follow:

\[ f^j_i - \hat{f}^j_i \leq 0 \]  \hspace{1cm} (III.8)
\[ g^j_i - \hat{g}^j_i \leq 0 \]  \hspace{1cm} (III.9)
\[ K \times f^j_i - \hat{f}^j_i \geq 0 \]  \hspace{1cm} (III.10)
\[ K \times g^j_i - \hat{g}^j_i \geq 0 \]  \hspace{1cm} (III.11)

Now we can constrain the maximum number of used registers \( FPR_{max} \) and \( GPR_{max} \) by the following:

\[ \sum_{i=1}^{N} (g^j_i) - GPR_{max} \leq 0, \forall j \in \{1, 2, ..., M\} \]  \hspace{1cm} (III.12)
\[ \sum_{i=1}^{N} (f^j_i) - FPR_{max} \leq 0, \forall j \in \{1, 2, ..., M\} \]  \hspace{1cm} (III.13)

Our optimization objective is to minimize the required cycles to execute the code \( (C_{run}) \). The complete formulation of our optimization is:
Minimize: $C_{run}$

Subject to:

$$\sum_{j=1}^{M} x_i^j = 1, \quad \forall i \in \{1, 2, ..., N\} \quad (\text{III.2})$$

$$\sum_{i : I_i \in \text{FPU}} x_i^j \leq 1, \quad \forall j \in \{1, 2, ..., M\} \quad (\text{III.3})$$

$$\sum_{i : I_i \in \text{LSU}} (x_i^j + x_i^{j+1}) \leq 1, \quad \forall j \in \{1, 2, ..., M - 1\} \quad (\text{III.4})$$

$$\sum_{k=1}^{M} (k \times x_i^k) - \sum_{k=1}^{M} (k \times x_i^k) + e_{ij} + 1 \leq 0, \quad \forall i, j : e_{ij} \in E \quad (\text{III.5})$$

$$\sum_{i=1}^{N} (g_{i}^j) - GPR_{\text{max}} \leq 0, \quad \forall j \in \{1, 2, ..., M\} \quad (\text{III.12})$$

$$\sum_{i=1}^{N} (f_{i}^j) - FPR_{\text{max}} \leq 0, \quad \forall j \in \{1, 2, ..., M\} \quad (\text{III.13})$$

$$\sum_{j=1}^{M} j \times x_i^j - C_{run} \leq 0, \quad \forall i : I_i \text{ has no successors (sink node)} \quad (\text{III.14})$$

This optimization problem is known to be NP-complete [37], so there is no known algorithm to solve it in polynomial time. In our current implementation, we use a greedy algorithm which yields an approximate solution in a practical time, as we describe in IV.3.3.
CHAPTER IV

Implementation

IV.1 STREAM benchmark on Shaheen

STREAM benchmark \cite{stream} is a program to measure the sustainable memory bandwidth for a set of basic vector computations on the target processor. Given arrays $A$, $B$, and $C$ and a constant $q$, these vector computations are: Copy ($A = B$), Scale ($A = q \times B$), Sum ($A = B + C$), and Triad ($A = B + q \times C$).

The theoretical peak of the processor does not provide practical upper bounds of the memory bandwidth. We use the STREAM benchmark to get practical upper bounds on the memory bandwidth of our target platform, PowerPC 450. We performed our experiments on Shaheen using the XL compiler with different configurations and number of cores to explore the set of configurations achieving the best performance.

We generated the results shown in Table IV.1 on a single node for different numbers of Open Multi-Processing (OpenMP) threads using optimization flag -O5 without using the Message Passing Interface (MPI).

Furthermore we examined the performance of the STREAM benchmark when independent MPI processes run on the same 4-cores chip as shown in Table IV.2.
<table>
<thead>
<tr>
<th>Threads #</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>total</td>
<td>per thread</td>
<td>total</td>
<td>per thread</td>
</tr>
<tr>
<td>Copy</td>
<td>3871</td>
<td>7538</td>
<td>3769</td>
<td>8718</td>
</tr>
<tr>
<td>Scale</td>
<td>3621</td>
<td>7074</td>
<td>3537</td>
<td>9812</td>
</tr>
<tr>
<td>Add</td>
<td>3701</td>
<td>7353</td>
<td>3677</td>
<td>9405</td>
</tr>
<tr>
<td>Triad</td>
<td>3707</td>
<td>7405</td>
<td>3703</td>
<td>9819</td>
</tr>
</tbody>
</table>

Table IV.1: STREAM performance (MB/s) on PowerPC 450 using 1, 2, and 4 threads.

<table>
<thead>
<tr>
<th>Mode (processes)</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Per process</td>
<td>Total</td>
<td>Per process</td>
</tr>
<tr>
<td>Copy</td>
<td>7656</td>
<td>3828</td>
<td>9992</td>
<td>2498</td>
</tr>
<tr>
<td>Scale</td>
<td>7200</td>
<td>3600</td>
<td>9904</td>
<td>2476</td>
</tr>
<tr>
<td>Add</td>
<td>7372</td>
<td>3686</td>
<td>10284</td>
<td>2571</td>
</tr>
<tr>
<td>Triad</td>
<td>7412</td>
<td>3706</td>
<td>10620</td>
<td>2655</td>
</tr>
</tbody>
</table>

Table IV.2: STREAM performance (MB/s) on PowerPC 450 running 2 and 4 MPI processes.

We have two observations from the STREAM benchmark results on a four PowerPC 450 cores chip. First, running four separate MPI processes achieved better performance than running one process with four OpenMP threads. Second, when we use less than four cores, the achieved bandwidth per process in MPI or per thread in OpenMP is more than the theoretical peak of a single core when using all the 4 cores.

These results motivate the use of MPI processes in our experiments. In addition, we run our experiments using four MPI processes on the four cores chip to avoid misleadingly optimistic results.

**IV.2 C and Fortran**

We implement three steaming kernels in C and Fortran and utilize published results as a benchmark to gain a better understanding of the performance of the general class of streaming numerical kernels on the IBM PowerPC 450. Our first challenge is in expressing a SIMDized mapping of the stencil operator in C or Fortran. Neither
language natively supports the concept of a unit of SIMD-packed doubles, though Fortran’s complex type comes very close. Complicating matters further, the odd cardinality of the stencil operators necessitates the use of clever tactics to properly SIMDize. Regardless of unrolling strategy, one of every two loads is unaligned or requires clever register manipulation. It is our experience that standard C and Fortran implementations of our stencil operators will compile into exclusively scalar instructions that cannot attain greater than half of peak floating point performance. For example, with a naive count of 53 flops per 27-point stencil, this is 31.5 Mstencil/s on a single core of the PowerPC 450. Register pressure and pipeline stalls further reduce the true performance to 21.5 Mstencil/s. We note that Datta [39] was able to improve this to 25 Mstencil/s using manual unrolling, common subexpression elimination, and other optimizations within C.

The XL compilers support intrinsics which permit the programmer to specify which instructions are used, but the correspondence between intrinsics and instructions is not exact enough for our purposes. The use of intrinsics can aid programmer productivity, as this method relies upon the compiler for such tasks as register allocation and instruction scheduling. However, our methods required precise control of such aspects of the produced assembly code, making this path unsuitable for our needs.

IV.3 Synthetic Assembly

The challenges in writing fast kernels in C and Fortran motivate us to program at the assembly level, a (perhaps surprisingly) productive task when using our code synthesis framework: an experienced user was able to design, implement, and test efficient kernels for a stencil operator in one day using the framework. Initially, we design two small, naively scheduled, 3-point kernels: mutate-mutate and load-copy. The
relative balance between load/store and floating point cycles consumed distinguishes the kernels from each other.

IV.3.1 Kernels Design

We find that efficiently utilizing SIMD units in stencil computations is a challenging task. To fill the SIMD registers, we pack two consecutive data elements from the fastest moving dimension, $k$, allowing us to compute two stencils simultaneously, as in [40]. Computing in this manner is semantically equivalent to an unrolling by 2 in the $k$ direction. Since $i$ and $j$ are static for any given stream, we denote the two values occupying the SIMD register for a given array by their $k$ indices, e.g., SIMD register $a_{34}$ contains $A_{i,j,3}$ in its primary half and $A_{i,j,4}$ in its secondary half.

The stencil operators map a subset of adjacent $A$ elements with odd cardinality to each $R$ element, as is illustrated in the left half of Figure IV.1, which depicts the SIMD register contents and computations mid-stream of a 3-point kernel. This prevents a direct mapping from SIMD input registers to SIMD output registers. Note that aligned SIMD loads from $A$ easily allow for the initialization of registers containing $a_{23}$ and $a_{45}$. Similarly, the results in $r_{34}$ can be safely stored using a SIMD store. The register containing $a_{34}$, unaligned elements common to the aligned registers containing adjacent data, requires a shuffle within the SIMD registers through the use of additional load or floating point move instructions.

We introduce two kernels, which we designate as mutate-mutate (mm) and load-copy (lc), as two different approaches to form the packed data into the unaligned SIMD registers while streaming through $A$ in memory. The mutate-mutate kernel, as the name indicates, mutates the value of the SIMD register by replacing the older half with the next element of the stream. In our example in Figure IV.1 we start with $a_{23}$ loaded in the SIMD register, then after the first computation we load $a_{4}$ into the primary part of the SIMD register so that the full contents are $a_{43}$. 
The load-copy kernel instead combines the unaligned values in a SIMD register from two consecutive aligned quad-words loaded in two SIMD registers by copying, via an inter-register move, the primary element from the second register to the primary element of the first. In our example $a_{23}$ and $a_{45}$ are loaded in two SIMD registers. Then, after the needed computations involving $a_{23}$ have been dispatched, a floating point move instruction replaces $a_2$ with $a_4$ to form $a_{43}$.

The two kernels use an identical set of floating point instructions, visually depicted in the right half of Figure IV.1 to accumulate the computations into the result registers. The two needed weight coefficients are packed into one SIMD register. This packing is enabled by the copy feature provided by the floating point unit. The first
floating point operation is a cross copy-primary multiply instruction, multiplying two copies of the first weight coefficient by the two values in $a_{23}$, then placing the results in the SIMD register containing $r_{34}$ (Figure IV.1a). Then, the value of $a_{23}$ in the SIMD register is mutated to become $a_{43}$, replacing one data element in the SIMD register. The second floating point operation is a cross complex multiply-add instruction, performing a cross operation to deal with the reversed values (Figure IV.1b). Finally, the value $a_{45}$, which has either been preloaded by load-copy or is created by a second mutation in mutate-mutate, is used to perform the last computation (Figure IV.1c).

We list resource requirements of the load-copy and the mutate-mutate kernels in Table IV.3. The two kernels are at complementary ends of a spectrum. The mutate-mutate kernel increases pressure exclusively on the load pipeline while the load-copy kernel incurs extra cycles on the floating point unit. The two strategies can be used in concert, using mutate-mutate when the floating point unit is the bottleneck and the load-copy when it is not.

Table IV.3: Resource usage per stencil of mm and lc

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Operations</th>
<th>Cycles</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ld-st FPU</td>
<td>ld-st FPU</td>
<td>Input</td>
</tr>
<tr>
<td>mm</td>
<td>2-1 3</td>
<td>6 3</td>
<td>1</td>
</tr>
<tr>
<td>lc</td>
<td>1-1 4</td>
<td>4 4</td>
<td>2</td>
</tr>
</tbody>
</table>

### IV.3.2 Unroll-and-Jam

The 3-point kernels are relatively easy to specify in assembly, but the floating point and load/store instruction latencies will cause pipeline stalls if they are not unrolled. Further unrolling in the $k$-direction is a possible solution that we do not explore in this thesis. Although this solution would reduce the number of concurrent memory streams, it would also reduce data reuse for the other stencils and require even more intricate programming.
Unrolling and jamming once in transverse directions provides independent arithmetic operations to hide instruction latency, but interleaving the instructions by hand produces large kernels that are difficult to understand and modify. To simplify the design process, we constructed a synthetic code generator and simulator with reordering capability to interleave the jammed FPU and load/store instructions to minimize pipeline stalls. The synthetic code generator also gives us the flexibility to implement many general stencil operators, including the 7-point and 27-point stencils using the 3-point stencil as a building block. This implies that when we improve performance for the 3-point stencil we almost transparently improve performance for the entire class of stencil operators.

Unroll-and-jam serves a second purpose for the 7-point and 27-point stencil operators due to the overlapped data usage among adjacent stencils. When applied, it eliminates the redundant loads of the common data elements among the jammed stencils, reducing pressure on the memory subsystem by increasing the effective arithmetic intensity of the kernel.

This can be quantified by comparing the number of input streams, which we refer to as the “frame size,” with the number of output streams. For example a $2 \times 2$ jam for the 27-point stencil produces $2 \cdot 2 = 4$ output streams using a frame size of $4 \cdot 4 = 16$ while the same jam for the 3-point stencil has a frame size of just $2 \cdot 2 = 4$.

We use mutate-mutate and load-copy kernels to construct 3-, 7-, and 27-point stencil kernels over several different unrolling configurations. Table IV.4 lists the register allocation requirements for these configurations and provides per-cycle computational requirements. In both the mutate-mutate and load-copy kernels the 27-point stencil is theoretically FPU bound because of the high reuse of loaded input data elements across streams.

The mutate-mutate kernel allows more unrolling for the 27-point stencil than the load-copy kernel because it uses fewer registers per stencil. The number of allocated
registers for input data streams at the mutate-mutate kernel is equal to the number of the input data streams, while the load-copy kernel requires twice the number of registers for its input data streams.

Using unroll-and-jam enables the reuse of loaded input data across streams, increasing the FPU to load/store instruction ratio. The mutate-mutate kernel outperforms the load-copy kernel when using unroll-and-jam, allowing complete utilization of the FPU unit when the stencil operator has high reuse across the data streams.

IV.3.3 PowerPC 450 Simulator

The initial implementation of our simulator is created by Jed Brown. It optimizes our several hundred synthesized assembly instructions for in-order execution. First, we produce a list of non-redundant instructions using a Python generator. The simulator reflects an understanding of the instruction set, including semantics such as read and write dependencies, instruction latency, and which execution unit it occupies. It functions as if it were a PowerPC 450 with an infinite-lookahead, greedy, out-of-order execution unit. On each cycle, it attempts to start an instruction on both the load/store and floating point execution units while observing instruction dependencies. If this is not possible, it provides diagnostics about the size of the stall and what dependencies prevented another instruction from being scheduled. The simulator both modifies internal registers that can be inspected for verification purposes and produces a log of the reordered instruction schedule. The log is then rendered as inline assembly code which can be compiled using the XL or GNU C compilers.

Our code generation framework is shown in Figure IV.2. Python code is prepared to produce the instruction sequence of our code block. First, registers are allocated by assigning them variable names. Next, a list of instructions’ objects is created, utilizing the variables to address the registers. Enabling the instruction scheduler allows the simulator to execute the instructions out of order; otherwise they will be
<table>
<thead>
<tr>
<th>Configuration</th>
<th>Registers</th>
<th>Instructions</th>
<th>Bandwidth</th>
<th>Kernel Cycles</th>
<th>Frame Cycles</th>
<th>Stencils/Iteration</th>
<th>Input Weight</th>
<th>Result Weight</th>
<th>ld-st FPU Cycles</th>
<th>ld-st FPU Utilization %</th>
<th>FPU Cycles</th>
<th>FPU Utilization %</th>
</tr>
</thead>
<tbody>
<tr>
<td>27-mm-1x1</td>
<td>9</td>
<td>2</td>
<td>18-1</td>
<td>71.1</td>
<td>100</td>
<td>80</td>
<td>80</td>
<td>80</td>
<td>36.2</td>
<td>75</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>27-mm-1x2</td>
<td>12</td>
<td>4</td>
<td>24-2</td>
<td>66.7</td>
<td>100</td>
<td>71.1</td>
<td>100</td>
<td>100</td>
<td>34.7</td>
<td>75</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>27-mm-1x3</td>
<td>15</td>
<td>6</td>
<td>30-3</td>
<td>56.8</td>
<td>100</td>
<td>71.1</td>
<td>100</td>
<td>100</td>
<td>31.7</td>
<td>75</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>27-mm-2x1</td>
<td>12</td>
<td>4</td>
<td>40-4</td>
<td>66.7</td>
<td>100</td>
<td>71.1</td>
<td>100</td>
<td>100</td>
<td>34.7</td>
<td>75</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>27-mm-2x2</td>
<td>15</td>
<td>6</td>
<td>64-6</td>
<td>66.7</td>
<td>100</td>
<td>71.1</td>
<td>100</td>
<td>100</td>
<td>31.7</td>
<td>75</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>27-mm-2x3</td>
<td>18</td>
<td>8</td>
<td>80-8</td>
<td>66.7</td>
<td>100</td>
<td>71.1</td>
<td>100</td>
<td>100</td>
<td>34.7</td>
<td>75</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>27-mm-2x4</td>
<td>21</td>
<td>12</td>
<td>96-12</td>
<td>66.7</td>
<td>100</td>
<td>71.1</td>
<td>100</td>
<td>100</td>
<td>31.7</td>
<td>75</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>3-lc-2x1</td>
<td>1</td>
<td>2</td>
<td>1-1</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>3-lc-2x2</td>
<td>2</td>
<td>4</td>
<td>2-2</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>3-lc-2x3</td>
<td>4</td>
<td>8</td>
<td>4-4</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>3-lc-2x4</td>
<td>6</td>
<td>12</td>
<td>6-6</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>
executed in order when we turn the scheduler off. The instruction simulator uses virtual GPR, FPR, and memory to simulate the pipeline execution and to simulate the expected results of the given instruction sequence. A log is produced by the instruction simulator to provide the simulation details, showing the cycles at which the instructions are scheduled and the encountered data and structural hazards, if there are any. Also, the log can contain the contents of the GPR, the FPR, and the memory, at any cycle, to debug the code for results correctness. The C code generator takes the simulated instructions and appends their equivalent inline assembly code in a provided C code template. To provide more clarity in the generated assembly code, we associate each generated line with a comment showing the mapping between the used registers' numbers and their corresponding variables' names in the Python code.

Figure IV.2: The components of the code generation framework in this work
CHAPTER V

Performance

Code synthesis allows us to easily generate and verify performance of 3-, 7-, and 27-point stencil operators against our predictive models over a range of unrolling-and-jamming and inner kernel options. We use individual MPI processes mapped to the four PowerPC 450 cores to provide 4-way parallelization and ascertain performance characteristics out to the shared L3 and DDR memory banks.

The generated assembly code, comprising the innermost loop, incurs a constant computation overhead from the prologue, epilogue, and registers saving/restoring. The relative significance of this overhead is reduced when larger computations are performed at the innermost loop. This motivated us to decompose the problem’s domain among the four cores along the outermost dimension for the 3- and 7-point stencils, resulting in better performance. However, the 27-point stencil has another important factor, dominating the innermost loop overhead cost. Its computations inhibit relatively high input data points sharing among neighbor stencils. Splitting the innermost dimension allows the shared input data points to be reused by consecutive middle dimension iterations, where the processor will likely have them in the L1 cache, resulting in faster computations. On the other hand, if the computation is performed using a large innermost dimension, the shared input data points will no longer be in
the L1 cache at the beginning of the next iteration of the middle loop.

All three studies are conducted over a range of cubic problems from size $14^3$ to $362^3$. The problem sizes are chosen such that all variations of the kernels can be evaluated naively without extra code for cleanup.

Each sample is computed in a loop to reduce noise from startup costs and timer resolution, and the samples’ average is used. To have improved measurements, we repeat this experiment over another loop, selecting the fastest experiment.

All performance results are given per-core, though results were computed on an entire node with a shared L3 cache. Full node performance can be reasonably obtained by multiplying by 4.

During the course of our experiments on the stencils, we noticed performance problems for many of the stencil variants when loading data from the L3 cache. The large number of concurrent hardware streams in the unroll-and-jam approach overwhelms the L2 streaming unit, degrading performance. This effect can be amplified in the default optimistic prefetch mode for the L2, causing wasted memory traffic from the L3. We make use of a boot option that disables optimistic prefetch from the L2 and compare against the default mode where applicable in our results. We distinguish the two modes by using solid lines to indicate performance results obtained in the default mode and dashed lines to indicate results where the optimistic prefetch in L2 has been disabled.

\section*{V.1 3-Point Stencil Computations}

We begin our experiments with the 3-point stencil, our computational building block for the other experiments in our work (Figure V.1). The 3-point stencil has the lowest arithmetic intensity of the three stencils studied, and unlike its 7-point and 27-point cousins, does not see an increase in effective arithmetic intensity when unroll-and-
jam is employed. It is clear from Section IV.3 that the load-copy kernel is more efficient in bandwidth-bound situations, so we use it as the basis for our unroll-and-jam experiments. We see the strongest performance in the three problems that fit partially in the L1 cache (the peak of 224 Mstencil/s is observed at 26^3), with a drastic drop off as the problem inputs transition to the L3. The most robust kernel is the 2×1 jam, which reads and writes to two streams simultaneously, and can therefore engage the L2 prefetch unit most effectively. The larger unrolls (2×2, 2×3, and 2×4), enjoy greater performance in and near the L1, but then suffer drastic performance penalties as they exit the L1 and yet another performance dip near 250^3. Disabling L2 prefetch does not seem to have any large effect on the 2×1 kernel, though it unreliably helps or hinders the other kernels.

![3-point stencil performance with load-copy kernel](image)

Figure V.1: 3-point stencil performance with load-copy kernel
The 3-point kernel seems to be an ideal target on the PowerPC 450 for standard unrolling in the fastest moving dimension, $k$, a technique we did not attempt due to its limited application to the larger problems we studied. Unroll-and-jam at sufficient sizes to properly cover pipeline hazards overwhelms the L2 streaming unit due to the large number of simultaneous streams to memory. Unrolling in $k$ would cover these pipeline hazards without increasing the number of streams.

V.2 7-Point Stencil Computations

Our next experiment focuses on performance of the 7-point stencil operator (Figure V.2). We directly compare the mutate-mutate and load-copy kernels using the same unroll configurations, although we note that the mutate-mutate kernel can support a slightly larger unroll on this problem with a compressed usage of general purpose registers that were only implemented for the 27-point stencil. Again, we notice strong performance within the L1, then a dropoff as the loads start coming from the L3 instead of the L1. The performance drop near $256^3$ is caused when the $2 \times 3$ kernel’s frame size of $(2 + 2)(3 + 2) - 4 = 16$ multiplied by the length of the domain exceeds the size of L1. For smaller sizes, neighbors in the $j$ direction can reside in L1 between consecutive passes so that only part of the input frame needs to be supplied by streams from memory. With up to 16 input streams and $2 \cdot 3 = 6$ output streams, there is no hope of effectively using the L2 prefetch unit. We notice that performance improves with the L2 optimistic prefetch disabled slightly within the L3, and drastically when going to the DDR banks. The load-copy kernel shows better performance than the mutate-mutate kernel, it is clear here that load/store cycles are more precious than floating point cycles. It is likely that this kernel could attain better results with the use of cache tiling strategies, though we note that without any attempts at cache tiling this result is commensurate with previously reported results for the PowerPC 450 that
focused on cache tiling for performance tuning.

![Graph showing performance comparison]

Figure V.2: 7-point stencil performance comparison between mutate-mutate and load-copy kernels

In Figure V.3 we compare our results with Datta’s findings in [14]. Both our results and Datta’s results achieved perfect scaling from one to two and four cores. It is clear that Datta’s performance is compute bound resulting in the same performance for both small problems fitting in the L1 cache and large problems streaming from the L3 cache. Our work improved the computation performance, resulting in 2.2x speedup for small problems fitting in the L1 cache. We got similar performance for large problems because the performance becomes memory bound when the data is streamed from the L3 cache.
V.3 27-Point Stencil Computations

The 27-point stencil should be amenable to using a large number of jammed unrolls due to the high level of reuse between neighbor stencils. Indeed, we see nearly perfect scaling in Figure V.4 as we increase the number of jams from 1 to 6 using the mutate-mutate kernel. Although there is a gradual dropoff from the peak of 54 Mstencil/s (85% of arithmetic peak) as the problem sizes increase to the point that there is little reuse from the L1 cache, the kernel consistently sustains an average of 45 Mstencil/s (70% of arithmetic peak), even when the problem sizes greatly exceed the L3 cache.

Despite the L2-overwhelming frame size \((2 + 2)(3 + 2) = 20\) input streams and \(2 \cdot 3 = 6\) output streams), the jammed stencil achieves good performance with no blocking largely due to the high level of reuse of input data afforded by the unrolls in \(i\) and \(j\).

We compare with Datta’s results of the 27-point stencil in Figure V.5. Similar to the 7-point stencil results, Datta’s performance is compute bound for both small problems fitting in the L1 cache and large problems streaming from the L3 cache. This
resulted in the same performance. By improving the computation performance, we achieved improvements for both small problem size, achieving 2.2x speedup, and large problem size, achieving 1.7x speedup. The performance of the large problem size is improved because of the high arithmetic density in the 27-point stencil computations.

V.4 Model Validation

As we utilized a simulator which incorporates a model of the architecture’s performance characteristics to produce our kernels, we sought to validate our performance model by comparing the implicit predictions of our generative system to the empirical results seen in Table V.1.
Since performance within a core is often considerably easier to predict than when one must go beyond the core for memory access, we divide our comparisons into those on-core (L1) and those that go off the core, to L3 or main memory (streaming).

Our modeling of the 27-point stencils can be seen to be highly accurate. Inside the L1 cache the disparity between predicted and actual performance is consistently less than 1.5%. Shifting our attention to the streaming predictions, our accuracy can be seen to be considerably degraded. This is not surprising, given that our simulator was largely targeted to model the L1 domain. However, the relative error is less than 15% in all cases; this appears to be sufficient for producing highly efficient code. This shortcoming appears to stem directly from the level of detail with which we model the shared L3 cache and main memory subsystem and we are working to correct this in our simulator.

The match between predicted and witnessed performance for the 7-point stencil shows the same pattern. When modeling performance inside the L1 our relative error is less than 5%, but when extending our prediction to the components of the system
shared by all four cores, our error is as great as 17.5%. That our greatest error in this instance is an under-prediction that is probably attributable to a fortuitous alignment of the continuous vectors in the k-direction, as staggering these carefully often result in bandwidth benefits on the order of 10-15%.

Finally, while our simulator proved useful in generating code with tolerable performance characteristics for the 3-point stencil, it is apparent that our model does not accurately reflect some of the characteristics of the hardware. From some further experimentation, we are reasonably certain that the chief reason for our lack of accuracy in predicting the performance both within the L1 cache and beyond stems from the bandwidth that must be shared between the multiple write streams and our failure to account for this in our model. As the L1 cache is write-through in Blue Gene/P, this same characteristic likely accounts for both sets of results. It is most apparent in the 3-point stencil predictions as the ratio of write streams to either read streams or floating point operations is highest in this case.
### Table V.1: Predictions vs. observations for in-L1 and streaming performance

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Instruction limits</th>
<th>Bandwidth limits</th>
<th>Streaming limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>Predicted</td>
<td>Observed</td>
<td>In-L1</td>
</tr>
<tr>
<td>Simulated</td>
<td>Predicted</td>
<td>Observed</td>
<td>Streaming</td>
</tr>
<tr>
<td>Bandwidth</td>
<td>Predicted</td>
<td>Observed</td>
<td></td>
</tr>
<tr>
<td>Limits</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>27-mm-1x1</th>
<th>27-mm-1x2</th>
<th>27-mm-1x3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>27-mm-2x1</td>
<td>27-mm-2x2</td>
<td>27-mm-2x3</td>
</tr>
<tr>
<td></td>
<td>27-mm-2x4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 100.81 | 136.76 | 24.488 | 233.72 | 130.76 | 338.72 | 130.76 | 393.72 | 425   |
| 110.02 | 136.76 | 24.115 | 233.72 | 130.76 | 338.72 | 130.76 | 393.72 | 425   |
| 113.12 | 136.76 | 31.22  | 233.72 | 130.76 | 338.72 | 130.76 | 393.72 | 425   |
| 120.03 | 136.76 | 19.16  | 233.72 | 130.76 | 338.72 | 130.76 | 393.72 | 425   |
| 123.81 | 136.76 | 17.29  | 233.72 | 130.76 | 338.72 | 130.76 | 393.72 | 425   |
| 60.46  | 136.76 | 17.29  | 233.72 | 130.76 | 338.72 | 130.76 | 393.72 | 425   |
| 71.04  | 136.76 | 17.29  | 233.72 | 130.76 | 338.72 | 130.76 | 393.72 | 425   |
| 72.25  | 136.76 | 17.29  | 233.72 | 130.76 | 338.72 | 130.76 | 393.72 | 425   |
| 78.59  | 136.76 | 17.29  | 233.72 | 130.76 | 338.72 | 130.76 | 393.72 | 425   |
| 80.88  | 136.76 | 17.29  | 233.72 | 130.76 | 338.72 | 130.76 | 393.72 | 425   |
| 427    | 338.72 | 130.76 | 393.72 | 425   |
| 427    | 338.72 | 130.76 | 393.72 | 425   |
| 427    | 338.72 | 130.76 | 393.72 | 425   |
| 427    | 338.72 | 130.76 | 393.72 | 425   |

| 27-mm-2x3 | 27-mm-2x4 |
|           |           |
| 31.32     | 136.76    |
| 338.67    | 130.76    |
| 393.72    | 425       |
| 425       |           |
| 425       |           |
| 425       |           |
| 425       |           |
CHAPTER VI

Concluding Remarks

VI.1 Conclusion

The main contribution of this thesis is the reuse of registers in constant coefficient linear operators. The loads of the input vector elements and stores of the output vector elements are minimized and the fraction of multiply-adds among all cycles is maximized. This is achieved by using our novel streams computation algorithms mutate-mutate and load-mutate. This was possible, without data layout reordering, because of the SIMD-like instructions of the PowerPC 450 core.

The recommended research agenda for libraries in the exascale domain includes the fusion of library routine implementations as well as the creation of frameworks that enable the optimal instantiation of a given routine when supplied with architectural information [41]. We feel that the work presented here is a contribution to that end. Further, the nature of our simulator allows us to optimize our code as measured by other metrics such as bandwidth or energy consumption, given a simple model of the cost. We also demonstrate performance comparable to advanced cache tiling approaches for the 7-point stencil, despite the fact that we make no effort to optimize for cache reuse.
VI.2 Future Work

While the three individual problems considered (3-point stencils in one dimension and 7-point and 27-point stencils in three dimensions) all with constant coefficients and symmetry within each spatial dimension (but not across them) are important, there are also two generalizations of importance: higher-order stencils and chained iterative passes over the vectors. To set upper bounds for the generalization of the performance enhancement techniques of this thesis to these other important kernels, we characterize their volumes of reads and writes of floating point and integer arguments relative to the number of floating point multiply-adds performed.

Higher-order stencils expand the number of adjacent input vector elements that enter into a single output vector element, in successive steps of semi-width one in each of the spatial dimensions $i$, $j$, and $k$, with an additional weight coefficient corresponding to each additional increment of semi-width in each dimension. This is a modest generalization. Higher-order discretization increases register pressure because of the larger number of inputs that combine in each output. Opportunities for reuse of input elements expand with the stencil width up to the ability to keep them resident. In a $P$-point regular stencil (regardless of number of spatial dimensions) each input element is operated upon with a pre-stored weight $P$ times: once in the stencil centered upon it, and once in each neighboring stencil with which its own stencil overlaps. Floating point arithmetic intensity increases in proportion to $P$. That is, if there are $N$ elements in the input or output array, there are $PN$ floating point multiply-adds per $N$ floating reads and $N$ floating writes. Explicit methods for nonlinear systems, especially with high-order discretization techniques such as Weighted Essentially Non-Oscillatory (WENO) or discontinuous Galerkin [42], have similar properties, including a larger number of input streams, but with much higher arithmetic intensity.

$S$-stage chaining (as in the simultaneous accumulation of $Ax$, $A^2x$, $A^3x$, \ldots $A^sx$)
allows the output vector to be fed back as input before being written. Per output
der vector of $N$ floating point writes, there are $N/S$ reads and $PN$ floating point multiply-
adds. Therefore, up to the ability to keep the additional operands cached, both higher-
order operators and chained operations improve the potential for the transformations
described here.

Two other generalizations tend to make the work of this thesis less relevant. Ir-
regular stencils require integer reads, in addition to floating point reads to determine
which elements of the input vector go with each row of the matrix. This further
dilutes advantages that lead to the great breakthroughs in stencils per second de-
scribed here. Stencil operations with constant coefficients and sparse matrix-vector
multiplies with general coefficients are similar when counting floating operations, but
very different when it comes to data volume.

Spatially varying coefficients require the loading of additional weights, each of
which is used only once, each of which is of the same floating precision of the input
and output vectors, $P$ of them in the production of each output vector element, with
each input vector element being combined with $P$ different weights. While each input
and output element can still be reused up to $P$ times in the execution of one pass
through the overall vector-to-vector map, the dominant array in the workspace is
the coefficient matrix of weights so the benefits of reusing the vectors are minimal.
This situation is typical for nonlinear problems when using Newton-Krylov and linear
multigrid methods. However, when “free flops” are available, the weights can also
be recomputed on the fly as a nonlinear function of a given state and/or scalar
coefficients. In this case, the number of input streams is similar to the linear constant
coefficient case (perhaps larger by a factor of 2 or 3), but the number of floating point
results is several times higher and involves the problem-specific “physics.” Putting
the physics inside the kernel like this suggests that there will be an emphasis on the
ability to quickly develop high-performance kernels for new physics.
APPENDICES
APPENDIX A

Code generation with SimPPC

We describe the code generation process of our framework by a walkthrough example to generate the innermost loop of a 27-point stencil kernel with a $1 \times 2$ unroll-and-jam. We drive our experiments by a Python module presented in section A.1. This module appends the generated code in a C code template described in section A.2. The generated code is prepared by executing the Python functions shown in section A.3. Finally, section A.4 contains the generated C code.

A.1 Driver

This code starts by parsing the arguments passed by the user and preparing a directory for the generated source code file (lines 3-10). Then, the generated code is prepared by the get_template_dict function (line 11). Next, the code reads the text of the C code template, by the get_template function at line 13. Finally, the generated code is appended in the template and the output C file is written by the write_template function in line 14.

```python
#!/usr/bin/env python3

def main():
    args = parse_args()
    build_dir = './build'
    ensure_dir(build_dir)
    gen_params = {'interleave': args.interleave, 'verbose': args.verbose}
    opt_params = ['unroll_i', 'unroll_j']
    for k in opt_params:
        if getattr(args, k) is not None:
            gen_params[k] = getattr(args, k)
    template_dict = get_template_dict(args.kernel, gen_params)
    if not args.dry_run:
        kernel_template = get_template(args.kernel)
```
write_template(args.kernel, kernel_template, template_dict, build_dir)

def get_template(kernel):
    import os
    with open(os.path.join('kernels', kernel + '.ic.t')) as template_file:
        kernel_template = template_file.read()
    return kernel_template

def write_template(kernel, kernel_template, template_dict, build_dir):
    import os
    with open(os.path.join(build_dir, 'gen_kernels.ic'), 'w') as kernel_file:
        kernel_file.write(kernel_template.format(**template_dict))

def get_template_dict(kernel, gen_params):
    import os, importlib, kernels, inspect
    kernel_module = importlib.import_module('kernels.' + kernel, 'kernels')
    generators = [m for m in inspect.getmembers(kernel_module, Generator, inspect.isfunction) if m[0][0:4] == 'gen_']
    g = kernel_module.Generator(**gen_params)
    template_dict = dict()
    for name, method in generators:
        inst, clock = method(g)
        print(name + ': ' + str(clock))
        template_dict[name] = inst
    return template_dict

def parse_args():
    import argparse
    import sys
    from glob import glob
    from os.path import basename
    parser = argparse.ArgumentParser(description='Build.py: Generates kernels')
    parser.add_argument('-k', '––kernel', default='mock',
                        help='kernel to generate')
    parser.add_argument('-i', '––interleave', default='1',
                        help='enable/disable instructions
                        interleaving (default 1 "enabled")')
    parser.add_argument('-v', '––verbose', default='1',
                        help='Show
                        the log of the simulator (default 1 "enabled")')
    parser.add_argument('-n', '––dry-run',
                        help='Do not write the generated kernel',
                        action='store_true')
    parser.add_argument('–unroll_j', type=int, help='specify the
                        unrolling size in J')
    parser.add_argument('–unroll_i', type=int, help='specify the
                        unrolling size in I')
    class ListAction(argparse.Action):
def __call__(self, parser, namespace, values, option_string=None):
    print('[Kernels] \n' + '\n'.join(basename(k).replace('.ic.t', '') for k in glob('kernels/*ic.t')))
    sys.exit(0)
parser.add_argument('−l', '−−list', action=ListAction, nargs=0,
    help='list available kernels')
return parser.parse_args()

def ensure_dir(d):
    try:
        os.makedirs(d)
    except OSError as exc:
        if exc.errno == errno.EEXIST:
            pass
        else: raise

if __name__ == "__main__":
    main()

Listing A.1: The main function in the framework

A.2 Code template

Our C code template example contains the static part of the code and four replacement fields (\{gen_header_defines!\s\}, \{gen_initialize_registers!\s\}, \{gen_prologue!\s\}, and \{gen_inner_iter!\s\}). These fields are replaced by their corresponding generated code. The general format of the replacement field is \{string!\s\}. We use the terminology gen_[name] for the string. Each gen_[name] has a corresponding code generation function in Python, which has the same name. Our framework executes these python functions, starting with gen_, and replaces the replacement fields by the returned text in the C template.

/*
FP Registers allocation:
  0–15: Input data frame (j is fast moving dimension)
  16–19: First set of jam results
  20–23: Second set of jam results
  24–31: weights coefficients

Weights mapping in the stencil operator
frame k=0 k=1 k=2
   j   j   j
  7 4 7 6 2 6 7 4 7
  i 5 3 5 1 0 1 5 3 5
  7 4 7 6 2 6 7 4 7

Weight index: 0 1 2 3 4 5 6 7
Weight register: 24 25 26 27 28 29 30 31
*/
A.3 Code generation

The first step in the code generation is to give variable names to the registers (allocate registers) as shown in Listing A.3. We allocate registers for both GPR and FPR. Several variables address the same register in the case of the registers containing the weight coefficients. This is done to have direct mapping between each weight coefficient’s value and its stencil point instead of doing convoluted mapping between the location and the value of each weight coefficient in the stencil operator.
The \texttt{gen\_header\_defines} function, shown in Listing A.4, defines important values that are specific to the generated stencil code to have correct execution and performance measurements.

We define and initialize our registers at \texttt{gen\_initialize\_registers} function, shown in Listing A.5. It generates the variables definitions and initializations that are required by the generated code. In this function, the number of allocated variables changes according to the required unroll-and-jam size, where larger jams require more registers.

The loop prologue is generated by the \texttt{gen\_prologue} function, as shown in Listing A.6. The generated inline assembly code is responsible for loading the weight coefficients to the registers and perform the prologue’s computations of the loop. The Python’s code appends the instructions’ objects to the array \texttt{istream}. By the end of the function’s execution, the list of instructions are passed to the PowerPC 450 simulator (at line 23) to generate the inline assembly of these instructions and to compute the number of simulated cycles to execute the code. By default, the simulator performs out-of-order scheduling. The user has the option to disable the out-of-order scheduling using runtime parameters.

Finally, the main loop body is generated by the \texttt{gen\_inner\_iter} function, as shown in Listing A.7 starting from line number 19. For each point in the stencil’s operator, we perform the same FMA operation for all jams. Hence, we aggregate these common operations at the function \texttt{fma\_block} starting from line number 1.

```python
1 class Common:
2     def __init__(com_self, self):
3         # create function to deal with blocks of registers
4         com_self.c = get\_core(use\_trace=self.use\_trace)
5         com_self.cv = CViewer()
6
7         # Allocate fp registers
8         FPR\_pool = com_self.c.acquire\_fp\_registers(range(27, -1, -1))
9         # Allocate input data fp registers for the input data frame
10        com_self.streams = [0]*self.FRAME\_SIZE
11        for i in range(self.FRAME\_SIZE): com_self.streams[i] = FPR\_pool.pop()
12
13        # Allocate results fp registers
14        com_self.results = [0]*self.JAM\_SIZE
15        for i in range(self.JAM\_SIZE): com_self.results[i] = FPR\_pool.pop()
16
17        # Allocate 4 fp registers for the weights
18        com_self.w\_compact = com_self.c.acquire\_fp\_registers(range(28, 32))
19        com_self.w\_unique = [0]*8
20        com_self.w\_unique[0] = com_self.w\_compact[0]
21        com_self.w\_unique[1] = com_self.w\_compact[1]
23        com_self.w\_unique[3] = com_self.w\_compact[0]
```
Listing A.3: Python’s register allocation class

def gen_header_defines(self):
  stencil_size = 27
  str = ''
  str+='#define GEN_KERNEL_A_OFFSET (0)\n'
  str+='#define GEN_KERNEL_R_OFFSET (1)\n'
  str+='#define GEN_KERNEL_NAME ("%s")\n' % self.kernel_name
  str+='#define GEN_KERNEL_OPERATOR_SIZE (%.1f)\n' % (stencil_size
  str+='#define GEN_KERNEL_LS_CYCLES (%.1f)\n' % ((self.FRAME_SIZE*2 + self.JAM_SIZE)*2) # *2 cycles
  str+='#define GEN_KERNEL_FPU_CYCLES (%.1f)\n' % (stencil_size*2)
  str+='#define GEN_KERNEL_NI_JAMS (%d)\n' % self.JAM_I
  str+='#define GEN_KERNEL_NJ_JAMS (%d)\n' % self.JAM_J
  str+='#define N_FLOPS (N_STENCILS*%.1f)\n' % (stencil_size*2-1)
Listing A.4: Python’s definitions code generation

```python
def gen_initialize_registers(self):
    init_reg = ''
    for i in range(self.JAM_SIZE):
        init_reg += register double *r%d%d asm("%d");\n' % 
        (i//self.JAM_J, i%self.JAM_J, self.gpr_pointers[i])
    init_reg += '\n'
    # Assign the results strides pointers to the integer registers
    shift = list(range(self.JAM_J))
    for i in range(self.JAM_SIZE):
        init_reg += \
        (i//self.JAM_J, i%self.JAM_J, shift[i//self.JAM_J], shift[i%self.JAM_J])
    # Assign the weights pointer to its integer register
    init_reg += '\n weights = w - 2;\n'
    return init_reg, 0
```

Listing A.5: Python’s registers initialization code generation

```python
def gen_prologue(self):
    com = self.Common(self)
    inst_str = ['']
    istream = []
    clock_count = [0]
    # PROLOGUE
    # load weights registers
    # Weight index : 3 0 5 1 4 2 7 6
    # Weight register: 28 29 30 31
    # Python Index : 0 1 2 3
    # Compact indexing:
    # 7 6 4 2
    # 5 1 3 0
    #
    w_map = [0,1,2,0,2,1,3,3]
    for i in range(8):
        if i in [0,1,2,6]:
            istream += [isa.lfsdx(com.w_compact[w_map[i]],com.w_p,com.
                               sixteen)]
        elif i in [3,5,4,7]:
            istream += [isa.lfdux(com.w_compact[w_map[i]],com.w_p,com.
                               sixteen)]
        # load frame 1/3
        istream += [isa.lfdx(com.streams[i],com.a_ptr,com.a_indexing[i])
            for i in range(self.FRAME_SIZE)]
    self.generate_code_append(istream,com,inst_str,clock_count)
    istream = []
    return inst_str[0],clock_count[0]
```

Listing A.6: Python’s prologue code generation
```python
def fma_block(self, w, streams, result, stream_index, k_index):
    istream = []
    # Determine the current weight coefficient
    weight = w[stream_index%self.FRAME_J + 3*(stream_index//self.FRAME_J) + 9*k_index]
    # generate the indices of the active part of the frame
    active_window = [(stream_index+ i%self.JAM_J + self.FRAME_J*(i//self.JAM_J)) for i in range(self.JAM_SIZE)]
    # Do the FMA's
    for i in range(self.JAM_SIZE):
        ind = active_window[i]
        if k_index == 0:
            istream += [isa.fxpmul(result[ind], weight, streams[ind])]  # override the register at the first write
        else:
            istream += [isa.fcpmadd(result[ind], weight, streams[ind], result[ind])]
        elif k_index == 1: istream += [isa.fxcmxma(result[ind], weight, streams[ind], result[ind])]
        elif k_index == 2: istream += [isa.fcpmadd(result[ind], weight, streams[ind], result[ind])]
    return istream

def gen_inner_iter(self):
    com = self.Common(self)
    istream = []
    # do the FMA's for frame 1/3
    for i in self.block_ind: istream += self.fma_block(com.w, com.streams, com.results, i, self.K0)
    # mute for frame 2/3
    istream += [isa.lfsdxu(com.streams[i], com.a_ptr, com.a_indexing[i]) for i in range(self.FRAME_SIZE)]
    # do the FMA's for frame 2/3
    for i in self.block_ind: istream += self.fma_block(com.w, com.streams, com.results, i, self.K1)
    # mute for frame 3/3
    istream += [isa.lfsdxu(com.streams[i], com.a_ptr, com.a_indexing[i]) for i in range(self.FRAME_SIZE)]
    # do the FMA's for frame 3/3
    for i in self.block_ind: istream += self.fma_block(com.w, com.streams, com.results, i, self.K2)
    # write back result set 1/2
    istream += [isa.stfxdux(com.results[i], com.results_p[i], com.sixteen) for i in range(self.JAM_SIZE)]
    return self.generate_code(istream, com)

Listing A.7: Python's loop body code generation functions
```
This section presents the generated code of our 27-point stencil example. The out-of-order scheduling is disabled in Listing A.8 to provide better clarity of the different stages of the code. Then, we show how the inner loop body appears when our out-of-order scheduling is used in Listing A.9. This code does not show the mapping between the registers’ numbers and their corresponding variables’ names in the Python code to have a more compact view of the code.

```
//
FP Registers allocation:
0−15: Input data frame (j is fast moving dimension)
16−19: First set of jam results
20−23: Second set of jam results
24−31: weights coefficients

Weights mapping in the stencil operator
frame k=0 k=1 k=2
  j   j   j
  i 5 3 5 1 0 1 5 3 5
  7 4 7 6 2 6 7 4 7

Weight index : 0 1 2 3 4 5 6 7
Weight register: 24 25 26 27 28 29 30 31

#define GEN_KERNEL_A_OFFSET (0)
#define GEN_KERNEL_R_OFFSET (1)
#define GEN_KERNEL_NAME ("bgm\−mute\−27\−point 1x2 jam Kernel\−no interleaving")
#define GEN_KERNEL_OPERATOR_SIZE (27.0)
#define GEN_KERNEL_LS_CYCLES (52.0)
#define GEN KERNEL_FPU_CYCLES (54.0)
#define GEN_KERNEL_NI_JAMS (1)
#define GEN_KERNEL_NJ_JAMS (2)
#define N_FLOPS (N_STENCILS*53.0)

#define kernel_reference kernel_reference_27pt
#define fortran_kernel fortran_kernel_27pt

inline void gen_kernel(const double *restrict a, double *restrict r,
 const double *restrict w, int istride, int jstride) {
 register const double *restrict weights asm ("26");
 register int k asm ("31");
 register int sixteen asm ("27");
 register double * a_ptr asm ("3");
 register int next_frame asm ("4");
 register int next_i_jam asm ("5");
 register int next_i_jam asm ("6");

 next_frame = 1 − (istride) * (GEN_KERNEL_NI_JAMS+2−1) − jstride * (GEN_KERNEL_NJ_JAMS+2−1);
```
next_j_jam = 8*istride;
next_i_jam = 8*(istride - jstride * (GEN_KERNEL_NJJAMS+2-1));

a_ptr = (double *)a - istride - 1*jstride - next_frame;
next_frame=8*(next_frame); // first update in the prologue uses
next_frame for index update

sixteen = 16;

// Initialize registers
register double *r00 asm("7");
register double *r01 asm("8");
r00 = (double *)(r + 0*istride + 0*jstride - 1);
r01 = (double *)(r + 0*istride + 1*jstride - 1);
weights = w - 2;

// Prologue
asm volatile("lfsdux 28, %0, %1":"b" (weights):"b" (sixteen));
asm volatile("lfsdux 29, %0, %1":"b" (weights):"b" (sixteen));
asm volatile("lfsdux 30, %0, %1":"b" (weights):"b" (sixteen));
asm volatile("lfdux 28, %0, %1":"b" (weights):"b" (sixteen));
asm volatile("lfdux 30, %0, %1":"b" (weights):"b" (sixteen));
asm volatile("lfdux 31, %0, %1":"b" (weights):"b" (sixteen));
asm volatile("lfdux 31, %0, %1":"b" (weights):"b" (sixteen));
asm volatile("lfxdux 0, %0, %1":"b" (a_ptr):"b" (next_frame));
asm volatile("lfxdux 1, %0, %1":"b" (a_ptr):"b" (next_j_jam));
asm volatile("lfxdux 2, %0, %1":"b" (a_ptr):"b" (next_j_jam));
asm volatile("lfxdux 3, %0, %1":"b" (a_ptr):"b" (next_j_jam));
asm volatile("lfxdux 4, %0, %1":"b" (a_ptr):"b" (next_j_jam));
asm volatile("lfxdux 5, %0, %1":"b" (a_ptr):"b" (next_j_jam));
asm volatile("lfxdux 6, %0, %1":"b" (a_ptr):"b" (next_j_jam));
asm volatile("lfxdux 7, %0, %1":"b" (a_ptr):"b" (next_j_jam));
asm volatile("lfxdux 8, %0, %1":"b" (a_ptr):"b" (next_j_jam));
asm volatile("lfxdux 9, %0, %1":"b" (a_ptr):"b" (next_j_jam));
asm volatile("lfxdux 10, %0, %1":"b" (a_ptr):"b" (next_j_jam));
asm volatile("lfxdux 11, %0, %1":"b" (a_ptr):"b" (next_j_jam));
asm volatile("lfxdux 12, %0, %1":"b" (a_ptr):"b" (next_j_jam));

a_ptr += 1;

// Inner iteration
for (k=1; k < jstride - 2; k+=2) {
    asm volatile("fxpmul 12, 31, 0");
    asm volatile("fxpmul 13, 31, 1");
    asm volatile("fxcpmadd 12, 30, 1, 12");
    asm volatile("fxcpmadd 13, 30, 2, 13");
    asm volatile("fxcpmadd 12, 31, 2, 12");
    asm volatile("fxcpmadd 13, 31, 3, 13");
    asm volatile("fxcpmadd 12, 29, 4, 12");
    asm volatile("fxcpmadd 13, 29, 5, 13");
    asm volatile("fxcpmadd 12, 28, 5, 12");
asm volatile("fxcpmadd 13, 28, 6, 13");
asm volatile("fxcpmadd 12, 29, 6, 12");
asm volatile("fxcpmadd 13, 29, 7, 13");
asm volatile("fxcpmadd 12, 31, 8, 12");
asm volatile("fxcpmadd 13, 31, 9, 13");
asm volatile("fxcpmadd 12, 30, 9, 12");
asm volatile("fxcpmadd 13, 30, 10, 13");
asm volatile("fxcpmadd 12, 31, 10, 12");
asm volatile("fxcpmadd 13, 31, 11, 13");
asm volatile("lfsdux 0, %0, %1":"+b" (a_ptr):"b" (next_frame));
asm volatile("lfsdux 1, %0, %1":"+b" (a_ptr):"b" (next_jam));
asm volatile("lfsdux 2, %0, %1":"+b" (a_ptr):"b" (next_jam));
asm volatile("lfsdux 3, %0, %1":"+b" (a_ptr):"b" (next_jam));
asm volatile("lfsdux 4, %0, %1":"+b" (a_ptr):"b" (next_jam));
asm volatile("lfsdux 5, %0, %1":"+b" (a_ptr):"b" (next_jam));
asm volatile("lfsdux 6, %0, %1":"+b" (a_ptr):"b" (next_jam));
asm volatile("lfsdux 7, %0, %1":"+b" (a_ptr):"b" (next_jam));
asm volatile("lfsdux 8, %0, %1":"+b" (a_ptr):"b" (next_jam));
asm volatile("lfsdux 9, %0, %1":"+b" (a_ptr):"b" (next_jam));
asm volatile("lfsdux 10, %0, %1":"+b" (a_ptr):"b" (next_jam));
asm volatile("lfsdux 11, %0, %1":"+b" (a_ptr):"b" (next_jam));
asm volatile("fxcxma 12, 31, 0, 12");
asm volatile("fxcxma 13, 31, 1, 13");
asm volatile("fxcxma 12, 30, 1, 12");
asm volatile("fxcxma 13, 30, 2, 13");
asm volatile("fxcxma 12, 31, 2, 12");
asm volatile("fxcxma 13, 31, 3, 13");
asm volatile("fxcxma 12, 29, 4, 12");
asm volatile("fxcxma 13, 29, 5, 13");
asm volatile("fxcxma 12, 28, 5, 12");
asm volatile("fxcxma 13, 28, 6, 13");
asm volatile("fxcxma 12, 29, 6, 12");
asm volatile("fxcxma 13, 29, 7, 13");
asm volatile("fxcxma 12, 31, 8, 12");
asm volatile("fxcxma 13, 31, 9, 13");
asm volatile("fxcxma 12, 30, 9, 12");
asm volatile("fxcxma 13, 30, 10, 13");
asm volatile("fxcxma 12, 31, 10, 12");
asm volatile("fxcxma 13, 31, 11, 13");
asm volatile("lfdux 0, %0, %1":"+b" (a_ptr):"b" (next_frame));
asm volatile("lfdux 1, %0, %1":"+b" (a_ptr):"b" (next_jam));
asm volatile("lfdux 2, %0, %1":"+b" (a_ptr):"b" (next_jam));
asm volatile("lfdux 3, %0, %1":"+b" (a_ptr):"b" (next_jam));
asm volatile("lfdux 4, %0, %1":"+b" (a_ptr):"b" (next_jam));
asm volatile("lfdux 5, %0, %1":"+b" (a_ptr):"b" (next_jam));
asm volatile("lfdux 6, %0, %1":"+b" (a_ptr):"b" (next_jam));
asm volatile("lfdux 7, %0, %1":"+b" (a_ptr):"b" (next_jam));
asm volatile("lfdux 8, %0, %1":"+b" (a_ptr):"b" (next_jam));
asm volatile("lfdux 9, %0, %1":"+b" (a_ptr):"b" (next_jam));
asm volatile("lfdux 10, %0, %1":"+b" (a_ptr):"b" (next_jam));
asm volatile("lfdux 11, %0, %1":"+b" (a_ptr):"b" (next_jam));
asm volatile("fxcpmadd 12, 31, 0, 12");
asm volatile("fxcpmadd 13, 31, 1, 13");
asm volatile("fxcpmadd 12, 31, 10, 12");
asm volatile("fxcpmadd 13, 31, 11, 13");
asm volatile("fxcpmadd 13, 30, 2, 13");
asm volatile("fxcpmadd 12, 31, 2, 12");
asm volatile("fxcpmadd 13, 31, 3, 13");
asm volatile("fxcpmadd 12, 29, 4, 12");
asm volatile("fxcpmadd 13, 29, 5, 13");
asm volatile("fxcpmadd 12, 28, 5, 12");
asm volatile("fxcpmadd 13, 28, 6, 13");
asm volatile("fxcpmadd 12, 29, 6, 12");
asm volatile("fxcpmadd 13, 29, 7, 13");
asm volatile("fxcpmadd 12, 31, 8, 12");
asm volatile("fxcpmadd 13, 31, 9, 13");
asm volatile("fxcpmadd 12, 30, 9, 12");
asm volatile("fxcpmadd 13, 30, 10, 13");
asm volatile("fxcpmadd 12, 31, 10, 12");
asm volatile("fxcpmadd 13, 31, 11, 13");
asm volatile("stfxd ux 12, %0, %1":"+b" (r00):"b" (sixteen));
asm volatile("stfxd ux 13, %0, %1":"+b" (r01):"b" (sixteen));
}

Listing A.8: Generated 27-point stencil for a 1x2 jam kernel

asm volatile("fxpmul 12, 31, 0");
asm volatile("lfsdux 0, %0, %1":"+b" (a_ptr):"b" (next_frame));
asm volatile("fxcpmadd 12, 30, 1, 12");
asm volatile("lfsdux 1, %0, %1":"+b" (a_ptr):"b" (next_j_jam));
asm volatile("fxcpmadd 13, 30, 2, 13");
asm volatile("fxcpmadd 12, 31, 2, 12");
asm volatile("lfsdux 2, %0, %1":"+b" (a_ptr):"b" (next_j_jam));
asm volatile("fxcpmadd 13, 31, 3, 13");
asm volatile("lfsdux 3, %0, %1":"+b" (a_ptr):"b" (next_j_jam));
asm volatile("fxcpmadd 12, 29, 4, 12");
asm volatile("lfsdux 4, %0, %1":"+b" (a_ptr):"b" (next_i_jam));
asm volatile("fxcpmadd 13, 29, 5, 13");
asm volatile("fxcpmadd 12, 28, 5, 12");
asm volatile("lfsdux 5, %0, %1":"+b" (a_ptr):"b" (next_j_jam));
asm volatile("fxcpmadd 13, 28, 6, 13");
asm volatile("fxcpmadd 12, 29, 6, 12");
asm volatile("lfsdux 6, %0, %1":"+b" (a_ptr):"b" (next_j_jam));
asm volatile("fxcpmadd 13, 29, 7, 13");
asm volatile("lfsdux 7, %0, %1":"+b" (a_ptr):"b" (next_j_jam));
asm volatile("fxcpmadd 12, 31, 8, 12");
asm volatile("lfsdux 8, %0, %1":"+b" (a_ptr):"b" (next_i_jam));
asm volatile("fxcpmadd 13, 31, 9, 13");
asm volatile("fxcpmadd 12, 30, 9, 12");
asm volatile("lfsdux 9, %0, %1":"+b" (a_ptr):"b" (next_j_jam));
asm volatile("fxcpmadd 13, 30, 10, 13");
asm volatile("fxcpmadd 12, 31, 10, 12");
asm volatile("lfsdux 10, %0, %1":"+b" (a_ptr):"b" (next_j_jam));
asm volatile("fxcpmadd 13, 31, 11, 13");
asm volatile("lfsdux 11, %0, %1":"+b" (a_ptr):"b" (next_j_jam));
asm volatile("fxcxma 12, 31, 0, 12");
asm volatile("lfdux 0, %0, %1":"+b" (a_ptr):"b" (next_frame));
Listing A.9: Generated 27-point stencil loop body with instruction interleaving for a 1x2 jam kernel

asm volatile("fxcxma 13, 31, 1, 13");
asm volatile("fxcxma 12, 30, 1, 12");
asm volatile("lfxdux 1, %0, %1: "+b" (a_ptr):"b" (next_jam));
asm volatile("fxcxma 13, 30, 2, 13");
asm volatile("fxcxma 12, 31, 2, 12");
asm volatile("lfxdux 2, %0, %1: "+b" (a_ptr):"b" (next_jam));
asm volatile("fxcxma 13, 31, 3, 13");
asm volatile("lfxdux 3, %0, %1: "+b" (a_ptr):"b" (next_jam));
asm volatile("fxcxma 12, 29, 4, 12");
asm volatile("lfxdux 4, %0, %1: "+b" (a_ptr):"b" (next_i_jam));
asm volatile("fxcxma 13, 29, 5, 13");
asm volatile("fxcxma 12, 28, 5, 12");
asm volatile("lfxdux 5, %0, %1: "+b" (a_ptr):"b" (next_jam));
asm volatile("fxcxma 13, 28, 6, 13");
asm volatile("fxcxma 12, 29, 6, 12");
asm volatile("lfxdux 6, %0, %1: "+b" (a_ptr):"b" (next_jam));
asm volatile("fxcxma 13, 29, 7, 13");
asm volatile("lfxdux 7, %0, %1: "+b" (a_ptr):"b" (next_jam));
asm volatile("fxcxma 12, 31, 8, 12");
asm volatile("lfxdux 8, %0, %1: "+b" (a_ptr):"b" (next_i_jam));
asm volatile("fxcxma 13, 31, 9, 13");
asm volatile("fxcxma 12, 30, 9, 12");
asm volatile("lfxdux 9, %0, %1: "+b" (a_ptr):"b" (next_jam));
asm volatile("fxcxma 13, 30, 10, 13");
asm volatile("fxcxma 12, 31, 10, 12");
asm volatile("lfxdux 10, %0, %1: "+b" (a_ptr):"b" (next_jam));
asm volatile("fxcxma 13, 31, 11, 13");
asm volatile("lfxdux 11, %0, %1: "+b" (a_ptr):"b" (next_jam));
asm volatile("fxcmpadd 12, 31, 0, 12");
asm volatile("fxcmpadd 13, 31, 1, 13");
asm volatile("fxcmpadd 12, 30, 1, 12");
asm volatile("fxcmpadd 13, 30, 2, 13");
asm volatile("fxcmpadd 12, 31, 2, 12");
asm volatile("fxcmpadd 13, 31, 3, 13");
asm volatile("fxcmpadd 12, 29, 4, 12");
asm volatile("fxcmpadd 13, 29, 5, 13");
asm volatile("fxcmpadd 12, 28, 5, 12");
asm volatile("fxcmpadd 13, 28, 6, 13");
asm volatile("fxcmpadd 12, 29, 6, 12");
asm volatile("fxcmpadd 13, 29, 7, 13");
asm volatile("fxcmpadd 12, 31, 8, 12");
asm volatile("fxcmpadd 13, 31, 9, 13");
asm volatile("fxcmpadd 12, 30, 9, 12");
asm volatile("fxcmpadd 13, 30, 10, 13");
asm volatile("fxcmpadd 12, 31, 10, 12");
asm volatile("fxcmpadd 13, 31, 11, 13");
asm volatile("lfxdux 12, %0, %1: "+b" (r00):"b" (sixteen));
asm volatile("lfxdux 13, %0, %1: "+b" (r01):"b" (sixteen));
REFERENCES


