Stencil Computation Auto-Tuning via Dataflow Graph Transformations

Erik Lagercrantz
Abstract

Stencil Computation Auto-Tuning via Dataflow Graph Transformations

Erik Lagercrantz

Fine-tuning of optimizations such as loop tiling and array padding is typically required in order to achieve good performance in memory-intensive numerical applications on current computer architectures. Auto-tuning techniques can automate this traditionally laborious task and achieve good performance, but the techniques have so far been application-specific and difficult to adapt to new applications. This thesis investigates developing a framework and language supporting a more general purpose auto-tuner.

A high level language capable of expressing programs that deal with multi-dimensional arrays was created to serve as a program representation for use with the auto-tuner. The language is based on the dataflow model where a program is represented as a directed graph describing the flow of data between operations, and uses a novel iteration model based on a hierarchy of flows leading to simple representations of certain loop tiling and parallelization transformations. The iteration construct proved useful by allowing composition of tiling and parallelization transformations, but may not have been a good choice since it made it difficult to validate transformation opportunities.

The auto-tuner was evaluated by applying it to a simple stencil computation program. It was confirmed that memory access optimizations do matter for parallel stencil computations on mainstream multicore architectures, and that the impact of optimizations is architecture-dependent. Auto-tuning was shown to be a useful strategy for applying a range of parameterized optimizations. The presented auto-tuner lacks an algorithm for identifying valid optimization opportunities, and must be told what transformations to apply and where to apply them. Still, the use of a high-level dataflow language did simplify auto-tuning by removing the need for writing a program-specific code generator.
PREFACE

This is the Master of Science thesis for my degree in Information Technology Engineering from the Faculty of Science and Technology, Uppsala University. The work and the majority of the writing was carried out during the fall of 2010 at the Department of Information Technology, Uppsala University. The thesis is written in English, but there is a popular science summary in Swedish starting on page vi.

Erik Lagercrantz
Uppsala, 2015-04-01

FÖRORD


Erik Lagercrantz
Uppsala, 2015-04-01

Vilka optimeringar som resulterar i bäst prestanda beror på många faktorer som varierar från program till program och mellan olika datortyper. Kompilatörn identifierar mönster i programmet och förlitar sig sedan ofta på heuristisk information om vad som ger bra prestanda för en viss typ av dator. Detta fungerar utmärkt för många optimeringar, men det finns också fall där det är mycket svårt eller omöjligt att avgöra hur olika optimeringar bör anpassas eller kombineras för att nå ett bra resultat.


Autojustering har hittills använts i begränsad utsträckning för att till exempel snabba upp program som arbetar med stora mängder numeriska data. Dessa tillämpningar har gett goda resultat, vilket visar på teknikens potential, men ett problem är att lösningarna har varit programspecifika. Varje autojusteringslösning har utformats för att optimaera ett visst program, och anpassning för tillämpning på nya program kräver en stor arbetsinsats. Detta har hittills begränsat teknikens spridning.

Jag har därför undersökt hur man kan skapa en autojusteringslösning som är mer generell och som därmed enkelt kan tillämpas på nya program. En sådan lösning skulle kunna utgöra ett steg på
vägen mot att på ett bättre sätt utnyttja dagens och framtidens flerprocessordatorer.

En generell autojusterare måste kunna tolka det program som ska optimeras och utröna hur det kan modifieras. Vidare krävs att de optimeringar som ska justeras kan beskrivas med hjälp av någorlunda enkla transformationer som kan utföras på programmet. Dessa kriterier är svåra att uppfylla för traditionellt representerade program som består av en serie instruktioner, eftersom dessa beskriver hur något ska utföras snarare än vad som är målet. Detta är troligtvis en starkt bidragande orsak till att existerande autojusteringslösningar är programspecifika, och ett val av en mer lämplig programrepresentation kan därför underlätta utvecklingen av en generell autojusterare.

Den autojusteringslösning som beskrivs i denna rapport arbetar med en alternativ typ av programbeskrivning som bygger på dataflöden. Programmet byggs då upp som en enkelt riktad graf istället för en serie instruktioner. Grafen består av noder som utgör programmets operationer, och bågar som visar hur information flödar från en operation till en annan. Ett enkelt dataflödesspråk utformades för att användas i den nya autojusteringslösningen. Språket innehåller en konstruktion för iterativa beräkningar som skiljer sig från andra dataflödesspråk, och i rapporten beskrivs hur detta medför att ett antal optimeringar kan beskrivas som enkla transformationer av programmets dataflödesgraf.


Den beskrivna lösningen kräver fortfarande ett visst manuellt arbete för varje enskilt program som ska autojusteras, då den behöver hjälp med att identifiera optimeringsmöjligheter i programmet. Detta visade sig svårt att lösa med den valda programrepresentationen, eftersom den i vissa fall saknar tillräcklig information för att verifiera om en viss transformation är säker att applicera. En möjlighet för vidareutveckling är därmed att förbättra dataflödes språket i detta avseende och utveckla en algoritm för att identifiera och verifiera optimeringsmöjligheter i godtyckliga program.
LIST OF FIGURES

Figure 1  Simple 2-dimensional stencil computation  4
Figure 2  Data reuse in stencil computation  5
Figure 3  Tiled 2-dimensional stencil computation  6
Figure 4  Auto-tuning results by Datta et al.  10
Figure 5  Dataflow graph concept  12
Figure 6  SDF graph example  13
Figure 7  WSDF graph example  14
Figure 8  Shape of stencil in hardcoded auto-tuner  16
Figure 9  Performance results of hardcoded auto-tuner  20
Figure 10  Effect of loop tiling on performance results  22
Figure 11  Dataflow language elements  24
Figure 12  Dataflow type hierarchy  26
Figure 13  Dataflow memory handling examples  26
Figure 14  Dataflow data transfer strategies  28
Figure 15  Simple dataflow iteration example  29
Figure 16  Nested dataflow iteration example  30
Figure 17  Class hierarchy of dataflow implementation  30
Figure 18  Minimal stencil computation dataflow graph  34
Figure 19  Dataflow tiling operations  38
Figure 20  Untuned stencil computation dataflow graph  42
Figure 21  Tuned stencil computation dataflow graph  43
Figure 22  Performance results of dataflow auto-tuner  45

LISTINGS

Listing 1  Basic stencil code from C code generator  17
Listing 2  Tiled stencil code from C code generator  17
Listing 3  C code for Apply Stencil operation  33
Listing 4  C code for Assemble Elements operation  34
Listing 5  C code for stencil computation dataflow graph  35
Listing 6  C code for Tile dataflow operation  37
## ACRONYMS

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In, First Out</td>
</tr>
<tr>
<td>NUMA</td>
<td>Non-Uniform Memory Access</td>
</tr>
<tr>
<td>SDF</td>
<td>Synchronous Data Flow</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>SSE</td>
<td>Streaming Single Instruction Multiple Data (SIMD) Extensions</td>
</tr>
<tr>
<td>WSDF</td>
<td>Windowed Synchronous Data Flow</td>
</tr>
</tbody>
</table>
Part I

MAIN DISCOURSE
INTRODUCTION

Programmers of performance-critical software typically need to spend a significant amount of time manually optimizing the performance of their code. Because of the diversity and complexity of modern parallel computer architectures, many optimizations have an architecture-dependent impact, meaning that achieving satisfactory performance requires tuning of the optimizations for specific architectures; a laborious and traditionally manual task.

Auto-tuning is a method that assists tuning of program optimizations using a “generate and test” principle and automated search of alternative optimization combinations. Recently developed auto-tuning solutions have shown great promise by delivering significant performance improvements when tuning numerical computations for modern multicore architectures [1, 2]. However, most of these techniques have been designed for a specific program and contain detailed hardcoded instructions for tuning a particular algorithm. Adapting such a special purpose auto-tuning solution for other applications requires significant effort and knowledge. The availability of a more general purpose auto-tuner is therefore crucial for enabling wider use of auto-tuning techniques.

The purpose of this work is exploring how to develop a general purpose auto-tuner that reduces the efforts required to tune a program with a range of optimizations. Naturally, a general purpose auto-tuner must be able to reason about the program and the data manipulated by it in order to find safe optimization opportunities. The basic premise of this thesis is that such reasoning becomes feasible if the program is described using a suitable high level representation. The representation should allow the wanted optimizations to be implemented as simple reusable transformations, and it must expose the information required to determine where the transformations can be safely applied. A dataflow language may be a particularly suitable representation since control flow is abstracted while data movement and parallelism is directly exposed, as explained in section 1.3. The goals of this work are therefore to create a simple data flow language that lends itself well to auto-tuning, and to implement an auto-tuner that can apply a few different effective optimizations.

The auto-tuner is evaluated by applying it to a program that performs stencil computations; a type of nearest neighbor array computations described in section 1.1. To aid the evaluation, a traditional special purpose auto-tuner is also created and hardcoded for optimizing the same type of program. The results from the spe-
cial purpose solution are used as a baseline that the more general purpose dataflow solution is compared to.

1.1Stencil Computations

This work investigates auto-tuning applied to a widely used class of algorithms called stencil computations. A stencil computation is a form of nearest-neighbor algorithm in which a compute kernel is applied in one or more sweeps over a numerical array [3].

The following items are typically involved in a stencil computation: a) an n-dimensional input array where n is usually 2 or 3; b) a stencil of some shape which works as a moving mask or window into the input array; c) a compute kernel which is a function that is applied to data selected by the stencil; and in most cases d) a separate output array of the same shape as the input array. As illustrated in Figure 1, a stencil computation sweep consists of letting the stencil move over the elements of the input array in some predefined regular order. At each position, the elements which are overlapped by the stencil are used as input to the compute kernel, and the kernel output is usually written to the corresponding position in the output array (which may be used as the input array in the next sweep). In other words, each output element is calculated by applying the compute kernel to the values overlapped by the stencil when it is centered on the corresponding input element.

The following pseudo-code, which corresponds to the example in Figure 1, will perform a 2-dimensional stencil computation sweep through a M-by-N array with a 3-by-3 stencil:

```plaintext
for i:=2 to M-1 do
    for j:=2 to N-1 do
        out[i,j] := kernel([in[i-1,j-1], ..., in[i+1,j+1]]);
```

Since the stencil requires an input of 3x3 to produce a single output the stencil cannot be applied at the edge of the array. The iteration index limits are therefore inset by 1 in this case, meaning that a 1-element border of the output array will remain unset. Alternatively a modified calculation can be used at the border, or the output array can be made slightly smaller than the input array so as to completely remove the border. Note that the iteration is performed in row-major order, meaning that the outer loop goes over the rows of the array, and the inner loop goes over the columns within each row. The row-major iteration order is used in all computations performed in this work, and array elements are always stored in the corresponding order in memory. The dimension iterated in the inner loop can thus be called the unit stride dimension, owing to the fact that advancing one array element in this dimension corresponds to advancing a single memory location.

Stencil computations are widely used in scientific computing, image analysis and other fields and have become an important and
Introduction

(a) First iteration, index (2, 2).

(b) Second iteration, index (2, 3).

(c) Iteration N − 2 (end of row), index (2, N − 1).

(d) Iteration N − 1, index (3, 2).

(e) Iteration N, index (3, 3).

(f) Iteration (N − 2) × (M − 2) (end of sweep), index (M − 1, N − 1).

Figure 1: Simple 2-dimensional stencil computation with a 3-by-3 stencil. The illustration shows input data usage and output data production during selected iterations of a single sweep. The iteration is performed in row-major order. Note that nothing is written to the border of the output array in this example.
performance-critical part of many applications. In particular, stencil computations are the core of many algorithms for differential equation solving and image filtering [4].

Apart from widespread usage, the main reason for using stencil computations as a benchmark is that, while the algorithms are very simple, the performance characteristics are still complex. Changes that affect memory traffic can have a big impact on performance since stencil computations are in many cases memory bound [5]. For example, as illustrated Figure 2, the use of neighbors around each input element leads to data reuse which can be exploited by the cache hierarchy to reduce memory traffic. For any given cache size, the degree to which data reuse can be taken advantage of mostly depends on the reuse distance, i.e., the number of unrelated memory accesses between use and reuse of an element. Because of this, iteration order changes and other changes that affect memory reuse patterns may significantly affect performance. Compilers are typically not very good at exploiting opportunities for this kind of high level memory optimizations [1], making stencil computations a good candidate for auto-tuning.

Figure 2: Reuse of elements demonstrated for a stencil computation with the same state as in Figure 1(e). Each previously used cell contains its age as the number of iterations since it was last used. The cells in the rightmost column of the stencil have an age tied to the width of the array (except for the very last cell, which is used for the first time). To take advantage of this reuse, all cells with age N-2 or less (i.e. several full array rows) need to fit in the cache, since cells will otherwise be expelled from the cache before reaching age N-2, assuming the cache eliminates the least recently used element when full. The rest of the cells have an age of one, meaning that they were just used in the previous iteration. To keep these cached it is sufficient to have a small cache corresponding to the stencil size.

Extensive research has been conducted into high level optimization of stencil computations. The following list describes some of the optimization types that have been described [5, 6].

**Loop Tiling** is a transformation that changes the order of iteration within a sweep, without changing the result. Conceptually, tiling can be seen as dividing the input and output array into smaller blocks, and sweeping each block completely before moving on to the next one, as shown in Figure 3.
(a) First iteration, index (2, 2) in the first tile.

(b) Second iteration, index (2, 3) in the first tile.

(c) Iteration N/2 – 1 (end of row in first tile), index (2, N/2) in the first tile.

(d) Iteration N/2, index (3, 2) in the first tile.

(e) Iteration (N/2 – 1) * (M – 2) + 1, index (2, 1) in the second tile.

(f) Iteration (N – 2) * (M – 2) (end of sweep), index (M – 1, N/2 – 1) in the second tile.

Figure 3: The same computation as in Figure 1, but here the arrays are tiled in one dimension, which affects the iteration order. Note that input elements from neighboring tiles are used when computing output elements that lie on the edge of a tile.
The ways in which iteration order and thus tiling affect performance are connected to the cache hierarchy and reuse distance. Changing the iteration order in a way that reduces the number of iterations between use and reuse of an element may increase the chance that the element remains cached when reused, thus improving performance.

Imagine, for instance, a 2-dimensional array that is iterated in row-major order during a stencil computation sweep. If this array is tiled in the horizontal dimension, so that the width of a tile is less than the total array width, the number of elements per row is effectively reduced. This is illustrated in Figure 3. By adjusting the tile size we can change the number of previous rows (if any) that remain in the cache during the iteration, which affects the number of requests that need to be sent to the main memory to load the elements overlapped by the stencil. If the tiles are made too small, however, the reduced length of sequential memory accesses may reduce performance by undermining the ability of the memory system to exploit spatial reuse of data.

The following pseudo-code, which corresponds to the example in Figure 3 when $T = 2$, will perform a tiled stencil computation sweep through an $M$-by-$N$ array with a 3-by-3 stencil. The array is split into $T$ evenly sized tiles along the horizontal dimension:

```
for t:=1 to T do
  for i:=2 to M-1 do
    for j:=tilestart(t) to tileend(t) do
      out[i,j] := kernel(in[i-1,j-1],...,in[i+1,j+1]);
```

To avoid applying the stencil on the border of the input array, the function `tilestart` needs to be defined to return 2 for $t = 1$, and $(t - 1) \cdot N/T$ for $1 < t \leq T$. The `tileend` function similarly needs to be defined to return $N - 1$ for $t = T$ and $t \cdot N/T$ for $1 \leq t < T$.

By comparing the tiled example with the untiled example shown previously, it can be seen that in practice, a loop tiling transformation can be applied by wrapping the original loops in a new outer loop for iterating through the tiles, and manipulating the index limits of the inner loops.

By applying multiple levels of tiling (i.e., further subdividing each block into smaller blocks) we can adapt for multiple levels of caches, where the tile size at each level is adjusted to improve reuse in the corresponding cache. Still, the optimal tile sizes cannot be exactly predicted since hardware prefetchers and other code and data also interact with the caches in non-obvious ways.

Parallelization of stencil computations is straightforward in theory since each element in the output array can be calculated independently of any other (though some input elements will be shared). However, on today’s multi-core architectures parallelism is often limited by contention of resources such as bandwidth, cache space and hardware prefetchers. This means that the optimal level
of parallelism may be unpredictable, and the level of parallelism will at least affect the optimal combination of other optimizations. For example, the optimal tile size may be affected by parallelization if two processors share cache space.

Note that a parallelization scheme for stencil computations can be based on tiling as described above, by distributing tiles among the available processors. Parallelization can also be achieved on a smaller scale using instructions that take advantage of SIMD hardware. These two approaches may be combined.

**NUMA-AWARE** memory allocation implies making sure that each piece of data is placed at a memory location that can be accessed quickly from the processor that is most likely to work on the data. This is applicable on Non-Uniform Memory Access (NUMA) computers, where a memory bank may be logically closer to certain processors than to others. For stencil computations, NUMA-aware memory allocation can be achieved by distributing the initialization of the input array, so that each processor initializes the tiles that it is going to work with.

**Padding** of the input array may have an impact on the performance of stencil computations by affecting the memory access pattern, and hence also the cache access pattern, of the program. For example, with certain combinations of array sizes and tile sizes, the distance in memory between nearby array elements will be such that they are mapped to the same location in the cache, which may cause conflict misses. Padding the input array to artificially increase its size in some dimension may resolve the conflict misses by increasing the distance in memory between the array elements in question.

**Software Prefetching** can potentially be used to reduce load delays in cases where changing the iteration order prevents the hardware prefetcher from finding patterns in the memory accesses. However, in simple stencil computations the iteration order is highly regular even after loop tiling, meaning that the hardware prefetcher can do a good job of predicting memory accesses, so that software prefetching is unlikely to add any benefit.

**Bypassing the Cache** when writing to the output array leaves more cache space for other data, such as the input array. This is likely to result in a performance benefit because there is reuse of input array elements during a stencil computation sweep, but no reuse of output array elements.

As we have seen there are several conceptually simple optimizations that can be applied to stencil computations. Among the optimizations described above, tiling and parallelization are perhaps the more interesting ones. They perform high-level algorithmic changes with results that are difficult to predict, mainly because
they are affected by computer architecture and memory hierarchy
details that can be hard to model. These difficulties often pre-
vent us from analytically determining optimal parameter values for
each optimization, and further each added optimization is likely to
change the effect of other optimizations. Finding the optimal com-
bination of optimizations is therefore non-trivial, which leads us
to consider auto-tuning as discussed in the next section.

1.2 AUTO-TUNING

Auto-tuning is a name used for methods that assist computer pro-
gram optimization by automated empirical evaluation and selec-
tion of various optimization alternatives. Auto-tuning is a brute-
force approach that can be used to find optimal parameters for
parameterized optimizations, and finding optimal combinations of
mutually compatible optimizations, even in cases when theory and
heuristics cannot be used to correctly predict the optimal selection.

When auto-tuning a program, the set of possible combinations
of optimizations and optimization parameters is given structure
and treated as a traversable parameter space. A search algorithm
walks this parameter space and picks candidate solutions that are
then evaluated by applying the corresponding optimizations and
measuring the resulting run time. In the end, the evaluated can-
didate with the shortest run time is selected. The search algorithm
may be a naïve brute force search, which guarantees the selection
of a globally optimal result by trying every possible candidate so-
lution, or it may be a more sophisticated algorithm that applies
a heuristic using partial results to guide the progression of the
search.

Since performance is evaluated by measuring the actual run time
of a program, the auto-tuning result will necessarily be optimized
for any specific input data and parameters used during evaluation.
If this is not desired, it may be necessary to test each candidate
several times with different input data.

Similarly, the result will also necessarily be tuned to the environ-
ment — and specifically the computer architecture — in which the
evaluation is performed. In many cases this aspect may be seen as
a major advantage of auto-tuning, mainly because it allows pro-
grams to be optimized for computer architectures that are not yet,
or cannot be, well understood. It also allows tuning to multiple en-
vvironments or architectures without performing additional analysis
or extra manual work.

Stencil computations are a good example of the potential bene-
fits of auto-tuning for multiple architectures. It was argued in the
previous section that several interesting optimizations are applica-
ble to stencil computations, but that it is difficult to analytically
determine the best way to apply and combine some of those opti-
mizations since the result of each depends in subtle ways on the hardware and on other optimizations. Previous work by Datta et al. [1] has shown that auto-tuning optimization techniques can avoid these difficulties and achieve good performance on diverse modern multicore architectures.

Datta et al. built an auto-tuning environment for stencil computations supporting parallelization and optimizations such as multilevel tiling, array padding, NUMA-aware memory allocation, software prefetching, SIMD hardware utilization and cache bypass for store instructions. The auto-tuner was evaluated on various recent multi-core architectures, showing significant speedup over naïve implementations in every case. Performance improvements for some of the architectures are summarized in Figure 4.

Figure 4: Selected parts of a figure by Datta et al. [1], showing results of auto-tuning stencil computations on three different CPU architectures: Intel’s Clovertown, AMD’s Barcelona and Sun’s Victoria Falls. Optimizations included multiple levels of tiling (called blocking in the figure) corresponding to different memory hierarchy levels, as well as NUMA-aware memory allocation, array padding, software prefetching and cache bypass. The results are presented by the number of CPU cores used, and show that tuning was needed to successfully utilize hardware parallelism.

Apart from stencil computations, auto-tuning has also been used successfully to optimize real world linear algebra software such as ATLAS [7] and OSKI [8], signal processing software such as FFTW [9] and SPIRAL [10], and various other application-specific computations. These examples have demonstrated the usefulness of auto-tuning for achieving good performance on diverse computer architectures, especially on modern cache-based multicore architectures, where tuning was essential for achieving close to optimal performance. It appears that auto-tuning is becoming more important as the complexity of computer architectures increases.

All of the aforementioned auto-tuning examples were performed using application-specific auto-tuners, each incorporating a custom
parameterized code generator that outputs the code of the program with transformations applied according to given optimization parameters. Developing such a code generator is non-trivial and requires the programmer to investigate potential transformations and analyze their effect on the program code. With the approach taken by Datta et al., the code generator is a script that produces stencil computation C code with, e.g., variations in loop structures depending on parameters that control the details of tiling and tile sizes. A separate auto-tuner component searches the optimization parameter space and calls the code generator for each candidate combination of parameter values. To adapt this type of solution for auto-tuning a different program would entail rewriting the code generator to emit code corresponding to that program instead, which requires a significant amount of work.

The difficulty of reusing existing auto-tuning solutions seems to have largely limited the use of auto-tuning to certain high-profile numerical software, most notably cross-platform numerical computation libraries. The availability of a reusable, general purpose auto-tuning solution would facilitate auto-tuning of many applications.

The need for custom code generators can be eliminated by creating an auto-tuner that works with general purpose program descriptions. The auto-tuner needs to be able to read an input program and apply transformations that must not change the correctness of the program. Doing this is difficult if programs are described using a low level imperative programming language. Applying tiling or parallelization transformations, for instance, requires analyzing loops to determine number of iterations, and more significantly, data dependencies between loops and between iterations of a single loop. This is possible to do in simple cases, but extremely difficult or impossible to do in other cases. Stencil computations are regular enough that a straightforward implementation can be analyzed by existing loop analysis tools. However, these tools are fragile, and require programs to follow expected patterns. They are likely to fail for programs that are not written with this in mind. The difficulty of loop analysis is demonstrated by the fact that it is currently not included by default in mainstream compilers [11].

Application of important transformations that affect data movements, such as tiling and parallelization, can be greatly simplified by using a high-level language that removes the need for loop dependence analysis.

1.3 DATAFLOW LANGUAGES

In a dataflow programming language, a program is represented as a directed graph describing the flow of data between operations, as in Figure 5. Operations correspond to nodes in the graph. Edges
between nodes dictate how data moves or flows between operations or, in other words, how the output from one operation is used as input for another operation. Whereas in imperative programming the focus lies on describing control flow and state modifications, in dataflow programming it lies instead on describing what happens to the data that the program is operating on.

![Figure 5: A simple dataflow graph. When operations are executed, data flows over the edges that connect them.](image)

In most dataflow models the operations are seen as black boxes, and their construction is left undefined. They are often seen as automatons, but may be implemented using a traditional imperative programming language.

To enable execution of a dataflow program, we need to define when and how operations are run. The basic principle is that an operation can run (or fire) when data is available on all of its input edges. When an operation is run it will provide data on each of its output edges. When data is available for more than one operation they may in principle be run in parallel, and if an operation has no input edges it can be run at any time. Execution is said to be data-driven because the order in which operations are run is only constrained by availability of data (or equivalently, by the data dependencies that are implied by edges in the dataflow graph) [12].

Based only on the above description, a dataflow language would be quite restricted. It would not, for example, have any iteration construct. The basic model thus needs to be extended to be useful for general purpose programming. This is solved to varying extents and in different ways in actual dataflow languages.

One common way of attaining basic iteration is to let each edge carry a sequence of data tokens instead of just a single item of data, thus allowing operations to run repeatedly. If this is combined with letting operations consume and produce multiple data tokens when run, it allows operations to be run at different rates. These extensions roughly form the basis of streaming dataflow models, where the inputs and outputs of a program are conceptually infinite streams of data tokens. Operations, also called computation kernels, are connected together and to the inputs and outputs by channels which act as First In, First Out (FIFO) queues for the stream data. Streaming models map well to application domains where the input is generated and processed continuously, and where reuse of the data is largely local in both time and space.
Of particular note is Synchronous Data Flow (SDF), a basic streaming dataflow model on which many later dataflow languages are based. SDF was described and implemented in the 1980s and is most suitable for signal processing applications [12]. In this model it has to be statically declared, for each input and output of an operation, how many tokens of data will be consumed or produced, respectively, when the operation fires. This implies that programs can only describe loops with a predetermined number of iterations, which greatly limits the ability of SDF programs to handle dynamically sized data, but allows for sophisticated optimization and scheduling of computation. Figure 6 contains an SDF program dataflow graph.

Figure 6: An SDF graph example. When one data token is available on the input edge, Operation A can fire, producing 8 data tokens on the outgoing edge for operation B to consume. Operation B only consumes 2 tokens of data a time, so it needs to be fired at four times the rate of operation A. Operation C, on the other hand, will fire at the same rate as operation A.

SDF has been used to describe traditional signal processing applications such as data modems [12] and digital audio sample rate conversion [13]. However, because streams are one-dimensional, it cannot easily be used to process higher dimensional data, as required for stencil computations.

There have been multiple attempts at extending the SDF model to support multidimensional data. A recent example is the Windowed Synchronous Data Flow (WSDF) language by Keinert et al. [14], which incorporates streams of multidimensional data tokens, and also supports iteration over these using windows that can be made to move in a way that corresponds to stencil computations. However, the language was developed with a strong focus on minimizing buffer space between operations at the expense of other concerns, which led to the following problems: a) the introduction of concepts such as “effective tokens”, “virtual tokens” and “virtual token unions”, resulting in a complex application description; b) limited iteration that cannot easily be extended since it is formalized as a fundamental part of the language; and c) as in SDF, no support for dynamically sized data. Figure 7 contains an example that demonstrates the complexity of describing iteration in WSDF.

In summary, dataflow models do limit the expression of programmers in some ways. For example, data usage is limited to that which can be expressed as edges between operations, and control flow is limited to that which is implicitly defined by data dependencies. As we have seen, this has led to awkward ways of achieving iter-
Figure 7: Example WSDF graph presented by Keinert et al. [14], representing an image processing application. The edge annotations describe sizes of various types of tokens, windows, iteration steps and borders.

...ation. However, these limitations come with the advantage that it greatly simplifies program analysis, and thus optimization, by directly exposing data movements and parallelism. This should simplify optimizations that modify data movement, such as tiling, and makes parallelization trivial in many cases.

In this work, the focus lies on finding a simple high level program representation that is well suited for describing stencil computations and applicable optimizations. By relaxing the typical streaming model goals of optimal scheduling and minimal buffer space, it should be possible to develop a dataflow language with a simpler iteration concept, that also supports dynamically sized data.

1.4 HYPOTHESIS

With this work I seek to demonstrate auto-tuning of traditional imperative optimizations in the context of a dataflow graph. The work is primarily limited to n-dimensional stencil computations to make it possible to focus on the optimization steps without having to develop a complex application description infrastructure. The insights gained from this work should hopefully provide a piece in the puzzle of how to get programs automatically optimized for multicore execution.

Specifically, my hypothesis is that traditional imperative optimizations such as loop tiling can be easily expressed and applied as transformations of a dataflow graph, and that doing so will simplify auto-tuning of such optimizations.
This chapter describes a problem-specific solution for auto-tuning n-dimensional stencil computations, which I refer to as the hardcoded auto-tuner to distinguish it from the more general purpose dataflow auto-tuner described in chapter 4. The hardcoded auto-tuner was created to serve as a comparison baseline for the dataflow auto-tuner. The methods used to create the hardcoded auto-tuner were chosen to be similar to those used in the study by Datta et al. [1] that was described in section 1.2, with the intent of partially replicating the results produced by that study.

The hardcoded auto-tuner is a program that takes options specifying the size of a certain stencil computation problem, as well as allowed values for various optimization parameters, and finds the combination of parameters that results in the fastest run time when used to generate an optimized stencil computation program. The output of the auto-tuner is the best parameter values that were found, along with the corresponding optimized stencil computation C code. It also optionally outputs a table of parameter value combinations and corresponding run times for all tested configurations, which can be used for evaluation purposes.

The main steps performed by the auto-tuner are:

- Creating a representation of the parameter space, the set of allowed combinations of optimization parameters, controlled by arguments provided by the user.
- Traversing the parameter space, picking combinations of parameter values to evaluate.
- For each configuration, generating stencil computation C code transformed according to the corresponding optimization parameter values.
- Compiling and running the generated C code and measuring its run time.

These steps are described in the following sections, after an initial section describing the program that is being generated and tuned.

2.1 STENCIL PROGRAM

The generated C code is a complete program that initializes an input array with pseudo-random values, performs a stencil com-
vation repeatedly while measuring run time and, finally, sends the measurement results to the auto-tuner.

For simplicity, the stencil computation works on arrays with 32-bit integer elements. The stencil used contains the points at a Manhattan distance of 1 from the center point, as illustrated in Figure 8. The kernel that is applied calculates the bitwise exclusive OR of all the input values. This simple operation is chosen to minimize the computation complexity of the kernel and thus maximize the bandwidth dependency of the program, so as to put the focus on memory-related optimizations.

![Stencil Diagram](image)

Figure 8: Shape of the stencil used to perform a 2-dimensional stencil computation in program generated by the hardcoded auto-tuner. The stencil includes points at a Manhattan distance of 1 from the center point.

Listing 1 shows the interesting part of the code generated for a 2-dimensional stencil computation with no optimizations turned on (except for moving the array index calculations out of the inner loop, which is done regardless of optimization parameters).

2.2 OPTIMIZATIONS

Optimizations that can be added and tuned by the auto-tuner are:

**TILING**: Multiple levels of tiling can be added for each array dimension. A tiling optimization is tuned by a tile size parameter. Listing 2 shows an example of code generated with tiling.

**PARALLELIZATION**: The outermost tiling loop of any array dimension can be parallelized, with the tuning parameter specifying in which array dimension to tile for parallelization, and which CPU cores to use.

**PADDING**: Each row of the input and output array is padded with the same amount, tuned by a pad size parameter.

**CACHE BYPASS**: Cache can be bypassed when storing values to the output array. The tuning parameter for cache bypass is simply on or off.

NUMA-aware memory allocation is always enabled, and loop index calculations are always moved out of the inner loop, regardless of tuning parameters.
Listing 1: Main part of the C code generated for a basic 2-dimensional stencil computation. In this example the input array has a size of 500 by 300 elements. The code formatting has been edited to improve readability.

Listing 2: Main part of the C code generated for a 2-dimensional stencil computation, with one level of tiling in the unit stride dimension. In this example the input array has a size of 500 by 300 elements, divided into 4 tiles of size 125 by 300.
The code generation is implemented by means of a simple method. A complete syntax tree is not constructed. Instead the code generator module consists of functions that each returns one snippet of C code, and many also take a number of smaller snippets of C code as arguments. Most of these functions are trivial code builders corresponding to C syntax elements, such as `add` which takes two strings containing C expressions and returns a C expression for adding the two, or `for_loop` which takes initializer, loop-test, counter and body statements as strings, and returns a corresponding for loop statement. Functions for constructing more intricate code are built around calls to the trivial functions.

The most complicated code generation function is `stencil_iter`, which produces stencil computation code as shown in listings 1 and 2 after being provided with: a) stencil shape description; b) names of the input and output arrays; c) size of the arrays by dimension, including padding; d) border size for limiting the sweep area; e) a list of the levels of iteration, each item specifying a dimension and step size; f) a list of dimensions for which the outer loop should be parallelized; g) a mapping from threads to cpu cores; h) data type names; i) and finally, statements for the body of the inner loop. The function will return the provided body statements wrapped in nested for loops corresponding to the specified iteration levels. Loop index updates and other array index calculations are generated according to the step size of each iteration. Loops that should be parallelized are preceded with appropriate OpenMP pragma directives.

There is also a top-level code generator function which creates a complete program that can be compiled and executed. The main function of the generated program comprises array allocation and initialization code, stencil computation code surrounded by calls for measuring run time, and code for communicating results to the auto-tuner. Computation and measurement is repeated 9 times and the shortest run time is used to reduce noise. The shortest run time ought to be less susceptible to noise than the average since there is a lower bound that can be approached when there is no interference, but there is no upper bound.

The generated C code is passed to gcc version 4.4 for compilation. The following compiler flags are used: `-O2 -m64 -march=native -mtune=native -mmmx -msse -msse2 -msse3 -mssse3 -fopenmp`. The Streaming SIMD Extensions (SSE) compiler flags are needed for supporting the cache bypass optimization, which is implemented using an SSE streaming store instruction. The output executables have been manually inspected to verify that the compiler does not override the loop transformations that were made by the auto-tuner.
To evaluate a candidate program, the auto-tuner runs the compiled executable and records the presented run time. Only one candidate program is executed at a time to avoid interference. For the same reason, although multiple candidate programs are compiled in parallel, compilation and evaluation are not performed simultaneously.

2.4 SEARCHING THE PARAMETER SPACE

The auto-tuner employs a simple brute force search algorithm that tries all possible combinations of parameter values, ensuring that the results are based on complete information. This is inefficient but has proven to be sufficient for the particular program and parameter spaces that were used in this case, with auto-tuning typically taking a few hours to complete.

2.5 RESULTS

The auto-tuner was used to optimize a 2-dimensional stencil computation with an array size of $2^{10}$ rows by $2^{19}$ columns, stored and iterated in row-major order. Given that the elements were 32-bit integers, the combined storage requirements for the input and output array were 4 GiB in the base case, and slightly more when array padding optimizations were applied.

The optimization parameter space allowed the following optimizations: a) parallelization by tiling the outer dimension by 1, 2, 4 or 8 CPUs; b) tiling the outer dimension with a tile height of $2^n$ for $1 \leq n \leq 7$; c) tiling the inner (unit-stride) dimension with a tile width of $2^n$ for $6 \leq n \leq 18$; d) padding the inner dimension by 0 or 7 elements; and e) cache bypass for stores enabled or disabled. NUMA-aware memory allocation was always enabled. Padding was limited to 0 or 7 elements in this experiment to limit the size of the search space, and because a pad size of 7 had been identified in earlier experiments to give good results.

The experiments were run on two computer systems. The first machine had two quad-core 2.26 GHz Intel Xeon Nehalem 5520 CPUs of the Nehalem architecture, with 8 MiB L3 cache per CPU, 256 KiB L2 cache per core and 32 KiB of L1 data cache per core. It was equipped with 24 GiB of RAM, accessed with a maximum theoretical bandwidth of 25.6 GB/s per CPU.

The second machine had two dual-core 2.8 GHz AMD Opteron 2220SE CPUs of the Santa Rosa architecture, with 1 MiB of L2 cache per core and 64 KiB of L1 data cache per core. It was equipped with 8 GiB of RAM, accessed with a maximum theoretical bandwidth of 10.6 GB/s per CPU.
Performance improvement by auto-tuned optimizations on two computers based on different CPU architectures. The stacked bars separate the contribution to speedup from parallelization, tiling, array padding and cache bypass optimizations, compared to unoptimized serial version.

The main parts of the C code of the untuned program as well as the program tuned for the AMD Santa Rosa architecture are included in Appendix A.

The performance improvements attained by auto-tuning on the two architectures are shown in Figure 9. The tiling and cache bypass optimizations had a large impact on both architectures in the highly parallel case when running on all cores. When using only a single core per CPU, the impact was still large on the AMD Santa Rosa CPU, but non-existent or negligible on Nehalem. This indicates that the significantly larger memory bandwidth of the Nehalem CPU was large enough to satisfy the needs of the stencil computation on a single core. Array padding improved performance in all cases on the Opteron machine, but provided no improvement at all on the Nehalem machine.

An example of the impact of the tiling optimization as a function of tile size can be seen in Figure 10.

2.6 CONCLUSIONS AND DISCUSSION

It was confirmed that memory access optimizations do matter for parallel stencil computations on mainstream multicore architectures, and that the impact of optimizations varies from one architecture to another. This is consistent with the conclusions reached by Datta et al. [1]. Auto-tuning was shown to be a useful strategy for applying parameterized optimizations such as loop tiling.

The auto-tuner implementation was fairly straightforward, except for code generation. The code generator component consisted
of over 600 lines of Python code, much of it quite convoluted, which
was used to generate less than 200 lines of simple C code. While
some parts of the code generator could be reused to generate parts
of other programs, the main part of it was problem-specific.

With the auto-tuning model used here, the manual steps for auto-
tuning a program include a) writing code for solving the problem;
b) finding optimization opportunities; c) determining how the code
should change (systematically) according to optimization parameters;
d) implementing a code generator that generates equivalent
code, modified according to optimization parameters; and e) mod-
ifying or configuring the auto-tuner to search an appropriate pa-
rameter space and to correctly pass on the parameters to the code
generator.

While this process is workable, repeating it for each program is
impractical. The following chapters describe how a more general
purpose auto-tuner can be created which removes some of the steps
from the process.
Figure 10: Comparison of the effect of loop tiling on the Nehalem-based computer for minimum and maximum number of cores, showing speedup compared to untiled version as a function of tile width and height. The baseline for calculating speedup in (b) includes parallelization; this means that the plotted speedup is from tiling only.
We saw how creating an auto-tuner based on an application-specific code generator is a difficult and time consuming manual process. In order to simplify the process, a high level language that can represent programs for use with a general purpose auto-tuner is needed. I have created a dataflow language to serve this purpose. This chapter describes the design of the dataflow language and the implementation of a framework for creating and manipulating dataflow programs and generating runnable code. Also shown are a few examples of operations and how they can be used to build a program.

The main requirements for the language were that it should be
a) capable of expressing programs that deal with multi-dimensional arrays and iteration in general, and stencil computations in particular; b) sufficiently high level to allow simple and compact representation of said concepts; c) exposing the details that need to be examined or modified in order to apply transformations corresponding to loop tiling and other optimizations; and finally, for practical reasons d) simple to implement.

While each individual requirement is most likely satisfied by some existing dataflow language, I have found no language that satisfies all of them. Dataflow languages that support multi-dimensional arrays tend to be complex, and thus not simple to implement or extend. Existing languages are also likely to be missing details needed for certain optimizations, while at the same time containing features that are not relevant for this study, adding unnecessary complexity. As a consequence it was decided to start with a clean slate and design a simple language for investigating the needs for auto-tuning of stencil computations and similar algorithms.

The language and dataflow framework is designed for translating dataflow programs into C code, allowing the generation of efficient machine code to be delegated to a C compiler.

### 3.1 Language Basics

The basic building block of a program is an operation. An operation may contain (or more correctly, generate) C code that is to be executed when the operation runs (i.e., each time it is fired). Each operation can have any number of input ports and output ports. An input port is a named receiver of data that will be used by the operation, and an output port is a named provider of data
produced by the operation. Each port declares the type of the data that it will send or receive.

Pieces of data have to arrive at the same rate to all input ports that belong to the same operation, but the various input ports may be receiving data items of different types and sizes. An operation can run only when a piece of data is available on each input port. When the operation is run, it has to make a piece of data available on every one of its output ports. Therefore, when an operation is said to consume one item, it means that it is consuming one item from each of its input ports, and when it is said to produce one item it means that it is producing one item for each output port.

The C code within an operation should perform some computation that uses the available input data to produce some output data. Input data can be accessed by reading from variables that correspond to input ports, and output data is provided by writing to variables that correspond to output ports. The computation is not allowed to have any side effects that are visible to other operations.

Operations are organized by connecting ports to each other using directed edges, forming a dataflow graph. The direction of an edge is always from an output port of the source operation, to an input port of the target operation. Several edges may originate from the same output port, but only one a single edge may arrive to a given input port. Cycles are not permitted.

Figure 11 shows a graphical representation of the basic elements defined in this section.

![Figure 11: Illustration of the basic elements of the dataflow language. Operation A has two input ports and one output port, and correspondingly generates C code that reads two input variables and writes to a single output variable. Operation B has one input port and one output port. An edge connects the output port of Operation A to the input port of Operation B, ensuring that the output variable written by the former operation is used as an input variable in the latter.](image-url)
3.2 DATA TYPES

Each input port specifies the type of data it can receive, and each output port specifies the type of data that it provides. The supported data types are defined in a type hierarchy, formed as a tree with the most general type at the root and more specialized types further out on the branches. A type inherits from and is considered to conform to its parent type, so that every type conforms to the general root type.

The implemented data type hierarchy is shown in Figure 12. Data types include the base type to which all other types conform, the void type which is used for inputs or outputs that transport no actual data, and the primitive types which correspond to C types such as int and float. There are also array types with varying levels of specificity. A basic array type requires that the element type is known, but the size and shape of the array can vary. A more specific type can be used when the number of dimensions is known, and an even more specific type can be used when the exact shape of the array is known. This allows operations to generate efficient code (with, e.g., pre-calculated constant loop bounds) when the shape of an array is known in advance.

When using a fixed shape array type, the tokens passed between operations consist of a pointer to the start of the memory area allocated for the array. When using dynamically sized arrays, tokens comprise size and memory stride information along with a pointer to the array memory.

Corresponding to each array type there is also a range of view types, which are used for referring to regions of larger arrays. A view type is considered to conform to its corresponding array type, meaning that a view into a larger array can be used transparently in place of an array of the same size as the view. The view types can be used by some operations to provide parts of the input array as output without having to copy data from the input array to a separate output array; instead they can simply provide a pointer to the start of the wanted area within the input array. This is useful for creating array tiling operations, as we will see in section 4.1.

3.3 MEMORY HANDLING

To support use of the view types described in the previous section as a means to avoid unnecessary copying of data passed between operations, the base language has been extended with annotations for specifying where data storage should be allocated. This is achieved by adding a flag to each input and output port, indicating whether or not the port provides storage for the data.

The usefulness of this can be seen by studying the two examples in Figure 13. In example (a), one operation produces an array. The
second operation takes this as input and outputs a slightly larger array, which is equal to the input except for an added outer border. In this case, we have an operation that incorporates its input data unchanged as part of its output data.

In example (b), on the other hand, the first operation takes an array as input and outputs a smaller array equal to the input but with a border removed. Here, we have an operation that extracts and outputs an unmodified part of its input data. The smaller array is then passed to another operation that may use it in some way.

Depending on what storage allocation strategy is used, the two examples are affected in different ways. Imagine first that each operation is responsible for allocating or in some other way preparing storage space for its own output data. The Generate Array operation in example (a) would then allocate just enough space for its
output array and write the array data to this space. The Add Border operation would be given a pointer to the array, and would not be able to wrap the existing array in a border. This strategy thus requires copying of data in example (a). In example (b), however, copying can be avoided in the Remove Border operation by using a view type for the output port. The operation could then skip allocation of a separate output array, and simply provide a pointer to the first element inside the border of the input array, along with the stride information of the original input array. The Use Array operation could safely treat this as a pointer to a normal array.

Now imagine instead that each operation is responsible for allocating or preparing storage space for its input data. The Generate Array operation in example (a) would then write its output array to storage space provided by the Add Border operation. The Add Border operation could allocate storage large enough for storing the final output array including the added border, and provide the Generate Array operation with a pointer to the first element inside of the border in this array. The Generate Array operation would then write its array data directly to the final output array, and the Add Border operation would simply have to fill in the values at the border. With this strategy, copying of data is thus no longer needed in example (a). In example (b), though, the Remove Border operation can no longer simply provide a pointer to existing data, because it has to write the data into storage provided by the Use Array operation.

To summarize, copying can be eliminated in example (a) if storage is provided by the data consumer, but this leads to copying in example (b). Copying can be eliminated in example (b) if storage is provided by the data producer, but this instead leads to copying in example (a). By allowing both storage strategies copying can be eliminated in both cases, and this is achieved using the aforementioned flags added to each input and output port.

When an output port of a source operation is connected to an input port of a target operation, there are four possible combinations of storage flags, which result in different data transfer strategies.

- If only the output port provides data storage, the target operation will access data directly from there. In this case, the edge connecting the two ports is considered to be a read edge.

- If only the input port provides data storage, the source operation should contain code that writes the output data directly to the provided storage. The connection is considered to be a write edge.

- If both ports provide storage, indicating that the source operation will write to the storage allocated by its output port and the target operation will read from the storage allocated
by its input port, a copy operation is implied between the
operations. In this case, the connecting edge is considered
to be a copy edge. It may be possible to eliminate the copy
operation in some cases but this has not been investigated.

• Finally, if none of the ports provide storage, a shared storage
— to which the source operation should write and from which
the target operation should read — will automatically be
allocated by the dataflow framework. The edge that forms
the connection is therefore called a write-read edge.

The four possible situations and the resulting data transfer strate-
gies are illustrated in item 14. The input and output ports in Fig-
ure 13 are colored to show the storage allocation flags required to
avoid copying of data in the example programs.

![Figure 14: Graphical representation of data transfer strategies. A black-
filled shape represents a port with data storage provided, and
a white-filled shape represents a port that does not provide
storage. For an output port and an input port connected by an
edge this gives four possible combinations (from left to right),
resulting in the edge types a) write-read; b) write; c) read;
and d) copy.]

Deallocation of allocated memory is not handled by the current
implementation. This was not a major concern since the simple
programs studied in this work have static data regions; they have
a single input array and a single output array, and no memory is
dynamically allocated.

3.4 FLOWS AND ITERATION

In dataflow languages that derive from the SDF model, iteration is
based on the static data consumption and production rates of op-
erations. As discussed in section 1.3, this allows for sophisticated
scheduling optimizations, but it also limits the possible types of
iteration constructs: only loops with static number of iterations
can be constructed, and only highly regular iteration orders are
supported since iteration is based on consuming a stream of ele-
ments. Multidimensional iteration is also problematic; as we saw,
it results in complicated application descriptions.

For these reasons, I have used a different iteration model that
is not based on a stream concept, and which has no consumption
and production rate annotations. Instead, iteration is controlled by
the code inside a certain kind of operations. This allows any iteration order to be implemented, and also allows loops with dynamic number of iterations. As we will see in chapter 4, it also leads to simple and composable representations of tiling and parallelization transformations.

Operations are classified by kind. The following three kinds are supported: a) normal operations that consume one item and produce one item when run; b) generators that consume one item and produce multiple items, thus controlling iteration; and c) accumulators that consume multiple items and produce a single item. Other variations are not allowed. The number of items produced by a generator or consumed by an accumulator does not have to be static; it can be decided at runtime by the operation implementation.

Generators and accumulators are used in pairs, with each pair forming the basis of a so called flow. A flow is a logical grouping of operations, which consists of a subgraph that starts with a generator and ends with an accumulator, with any number of other operations in between. For each element produced by the generator, each operation within the flow will be fired exactly once, and the accumulator will receive exactly one element. In other words, all operations belonging to the same flow will be fired at the same rate, which is determined by the generator. Also, the number of items consumed by an accumulator will always be equal to the number of items produced by the corresponding generator. Figure 18 illustrates how a generator and accumulator form a flow that describes a simple iteration construct.

![Diagram](image.png)

**Figure 15:** Dataflow program for smoothing a signal using a simple moving average. The program consists of three operations: a generator shown in red, a normal operation shown in blue, and an accumulator shown in green. The generator takes a one-dimensional array and outputs a sequence of partly overlapping subarrays, as if by sliding a window over the input array. The middle operation takes an array and outputs the average value of its elements. Finally, the accumulator takes a sequence of values and puts them back together in an array.
By putting one flow within another, a hierarchy of flows can be created, corresponding to multiple levels of iteration. The program in Figure 18 demonstrates the use of nested iterations to calculate a matrix norm.

![Diagram of dataflow program with nested flows](image)

Figure 16: Dataflow program with nested flows, calculating the \( \infty \)-norm of a matrix as the maximum absolute row sum.

Every operation is part of a flow; either explicitly or by being part of an implicit top level main flow. Connections between different flows are only permitted through generators and accumulators. Each generator and accumulator has connections to two flows, called the outer flow and the inner flow, but is considered to be part of the one that controls its execution, i.e., the flow in which the input ports are located. Thus, a generator is part of its outer flow, and an accumulator is part of its inner flow.

### 3.5 Implementation

There is a base class for each type of entity (operations, ports, edges, flows and data types) which implements the relevant Application Programming Interface (API), and subclasses for specialization. The major parts of this class hierarchy are shown in Figure 17.

![Diagram of class hierarchy](image)

Figure 17: Hierarchy of key classes in the dataflow language framework.

Ignoring generators and accumulators for the moment, an operation is basically an object that contains or generates a fragment
of C code that should be executed to read a number of input variables, perform a computation, and write to a number of output variables. The code for a complete dataflow program then consists of such code fragments joined together in an appropriate order, intermingled with code for preparing the input and output variables.

Operations in dataflow graphs are instances of operation classes, and every type of operation is implemented in its own class. A normal operation class contains, at the minimum, a declaration of the names and types of input and output ports, and a code generation method that takes the names of input and output variables and returns a code fragment for performing the operation. If any of the input or output ports has a flag set indicating that it provides storage, the operation class should also contain a method that takes the name of a port and of the corresponding input or output variable, and returns code for initializing the variable (e.g., making it point to the correct place in memory).

The code generation method of a generator operation additionally takes an existing code fragment corresponding to its inner flow, and may include this code fragment by, for example, wrapping it in a loop.

Accumulator operations may have an extra code generation method that returns a code fragment for any preparation needed before starting the accumulation, since the main code fragment will be executed within an iteration. For example, the prepare fragment of a sum accumulator might initialize the sum to zero, while the main fragment would add a single value to the sum each time it was executed.

A few predefined operation classes are described in section 3.6.

Code generation starts with the top level main flow. To generate code for a flow, the constituent operations are traversed in a topological order, i.e., an order where for every edge uv, the operation u comes before v in the ordering. This ensures that the data dependencies of the operations are satisfied. During the traversal, code is generated for each operation, and the generated code fragments are joined in order. The following steps are performed for each operation:

- Incoming edges are followed backwards to the output ports of preceding operations, and the names of the variables used to store the output of those ports are retrieved to be used for accessing the input data in the current operation.

- New unique variable names are generated for the output ports of the current operation.

- Code fragments for declaring and initializing the output variables are generated. Initialization code is generated by the current operation if the outgoing edge is a read or copy edge, or by a succeeding operation if the outgoing edge is a write
edge, as discussed in section 3.3. If the outgoing edge is a write-read edge, generic initialization code is generated that is appropriate for the data type of the output port. In this case, for primitive types, no initialization is performed. For array types, the initialization code generates a memory allocation call corresponding to the size of the array.

- If the current operation is a normal operation or an accumulator, the main fragment of the operation code is generated. The code may read the input variables, perform some computation, and write to the output variables.

- If the current operation is a generator, multiple code fragments are generated and pieced together. Code for declaring and initializing the output variables of the corresponding accumulator is generated together with an optional fragment of preparation code needed by the accumulator. Code for the inner flow is generated recursively. Finally, the main code generation method of the generator operation is called and given the names of input and output variables, as well as the inner flow code to be embedded.

3.5.1 Array implementation

In the generated code, an array is described by four pieces of information: a) a typed pointer to the start of a contiguous chunk of memory; b) a shape description, which consists of a list of integers specifying the number of elements in each array dimension; c) an integer offset in case there is initial padding; and d) stride specification, which is a list of integers specifying the distance in memory between consecutive array elements in each dimension. The same pieces of information are used for all array types. However, for fixed-shape arrays, only the pointer needs to be stored in a variable. The other information is implicit in the data type and can be used as constants in the generated code, leading to code with, e.g., static increments in loops.

Array views are described similarly. The differences are that the offset is always zero, and that the pointer does not necessarily refer to the start of a chunk of memory; it points to the first element of the view, which may be anywhere within a chunk of memory that belongs to an array. A view will have a shape that is smaller than that of the target array, but the stride specification is always identical to that of the target array. The use of stride specifications when iterating arrays allows operations to work on arrays and array views indiscriminately.
Listing 3: Example showing the structure of code generated by the Apply Stencil operation. Array-specific values have been replaced with names in capital letters for increased readability. The values $S_1$ and $S_2$ refer to the stride lengths of the array.

```c
for (int idx1 = 0; idx1 < HEIGHT - 2; idx1 += 1)
    for (int idx2 = 0; idx2 < WIDTH - 2; idx2 += 1) {
        int a = input[OFFSET + S1 * idx1 + S2 * (idx2 + 1)];
        int b = input[OFFSET + S1 * (idx1 + 1) + S2 * idx2];
        int c = input[OFFSET + S1 * (idx1 + 1) + S2 * (idx2 + 2)];
        int d = input[OFFSET + S1 * (idx1 + 2) + S2 * (idx2 + 1)];
    // INNER FLOW CODE INSERTED HERE
    }
```

3.6 OPERATIONS

A few operations that are useful for constructing stencil computation programs have been implemented.

**APPLY STENCIL** is a generator operation that takes a 2-dimensional array as input and performs a stencil computation sweep using a 4-point stencil, in the same way as was done in the hard-coded auto-tuner in chapter 2. This produces code similar to that in Listing 3. There are 4 normal output ports, one for each element in the 4-point stencil. There are also two hidden output ports that may be accessed by the accumulator; one provides the shape of the iteration, i.e., the number of iteration steps in each dimension, and the other provides the current iteration indices.

**INDEX ITERATOR** is a generator that iterates over the coordinates of a given array shape, instead of the contents of an actual array. It can be used to initialize array elements based on the coordinates of their position in the array. There is one normal output port for each dimension, providing the current iteration index. There are hidden output ports that may be accessed by the accumulator, providing the shape of the iteration.

**ASSEMBLE ELEMENTS** is an accumulator that takes a sequence of values, and assembles them into an array. It can only be used if the generator of the inner flow provides information about the shape of the iteration, which is used to determine the shape of the output array, as well as the current iteration indices, which is used to determine where in the output array to place an element.
Listing 4: Example of code generated by the Assemble Elements operation. Array-specific values have been replaced with names in capital letters for increased readability. The idx1 and idx2 variables are provided by the generator of the flow (e.g., Apply Stencil or Index Iterator).

// PREVIOUS OPERATION: element = ... 
output[OFFSET + STRIDE1 * idx1 + STRIDE2 * idx2] = element;

SUM, MIN AND MAX are reduce accumulators that all have a single input port taking a sequence of values, which is reduced to a single value provided on a single output port.

KERNEL is a normal operation provided as an example of what might be used in a stencil computation program. It has four input ports that may correspond to four stencil points, and it calculates a Xor-based checksum of the input values.

Figure 18 illustrates how some of the operations described above can be used together to create a minimal stencil computation program, and Listing 5 contains C code generated from this example.

Figure 18: Stencil computation dataflow program, consisting of an Apply Stencil generator paired with an Assemble Elements accumulator. The inner flow contains a Kernel operation that in each iteration operates on the four values provided by the generator and provides a single value to the accumulator.

3.7 LIMITATIONS

While the language and type system are designed to handle dynamically sized data, some of the functionality needed for using dynamically sized arrays is currently unimplemented.

The memory management model currently has some features for avoiding excessive copying and multiple memory allocations
for each piece of data, which is enough for minimizing memory use in simple programs. However, allocated data is never deallocated. Thus, for more complex programs, memory leaks are unavoidable. To solve this problem a memory deallocation strategy has to be implemented.
In this chapter I present an auto-tuning solution that can be applied to programs expressed in the dataflow language described in chapter 3. The solution is intended to be general purpose in the sense that it requires no significant re-implementation for each program. However, the optimizations supported so far are adapted for stencil computations since they closely mimic those used in the problem-specific auto-tuner described in chapter 2. This enables a fair comparison with the results of the problem-specific auto-tuner.

The dataflow auto-tuner is a program that takes a dataflow program and a description of optimization opportunities as input and, given arguments specifying allowed values for various optimization parameters, finds the combination of optimization parameters that results in the fastest run time. The output of the auto-tuner is the best parameter values that were found, along with the corresponding optimized dataflow program. For evaluation purposes it can also optionally output a full table of the tested configurations with parameter value combinations and corresponding run times.

The construction of the dataflow auto-tuner has much in common with that of the hardcoded auto-tuner. They share the same basic steps of a) building a parameter space corresponding to user-given arguments; b) traversing the parameter space and picking combinations to evaluate; c) generating C code with transformations applied according to the parameters; d) compiling and running candidates and measuring run time.

Most steps are virtually unchanged from the hardcoded auto-tuner. The significant differences lie in the generation of optimized C code, which in the dataflow auto-tuner has been split into separate transformation and code generation steps. The optimizations supported by the hardcoded auto-tuner were recreated as dataflow graph transformations, which are described in the following sections. C code is generated from the dataflow graph as was described in section 3.5.

4.1 TILING

4.1.1 Implementation

Two dataflow operations, *Tile* and *Join Tiles*, are defined to support a transformation that corresponds to loop tiling of array operations.
The first operation, Tile, is a generator that takes an array and splits it along one of its dimensions into equally sized tiles that may optionally be overlapping. The Tile operation constructor takes three parameters: 1. the array dimension to tile, 2. the size of a tile in the tiled dimension, and 3. the amount of overlap between tiles. There is one input port which takes an array of any size and dimensionality, and a single output port which provides the tiles in order as a sequence of views into the input array. The tiles are of the same shape as the input array, except that the size in the tiled dimension is equal to the specified tile size.

The Tile operation implementation generates a loop that iterates over the tiles and wraps execution of the inner flow. In each iteration, an offset variable is incremented and used to calculate the memory address of the first element of the current tile. The address is stored in a variable that represents a view into the input array, and is then provided as input to succeeding operations which are part of the inner flow and thus executed within the loop body. Listing 6 shows the type of code that is generated for a Tile operation.

```c
for (int offset = 0; offset < DIM_SIZE; offset += TILE_STEP) {
    int * tile = &input_array[DIM_STRIDE * offset];
    // Code for inner flow inserted here
}
```

Listing 6: Example of code generated by the Tile operation for a fixed-shape input array. DIM_SIZE represents the size of the input array in the tiled dimension, TILE_STEP represents the size of a tile in the tiled dimension, minus the size of the overlap between tiles, and DIM_STRIDE represents the input array stride length, i.e., the distance in memory between successive elements, in the tiled dimension.

The second operation, Join Tiles, is an accumulator that can be seen as taking a series of equally sized arrays and stitching them together, in order, into a larger array. A single parameter specifies the dimension along which the tiles are joined. An array type can be given as the data type of the input port. The data type of the single output port will be the same as that of the input port, except that the size in the joined dimension is multiplied by the number of input arrays received.

In practice, by setting the data storage flag of its input port, the Join Tiles operation does not actually need to perform the work of copying data from separate pre-existing tiles into the output array. Instead, it tiles the output array similar to how the Tile operation tiled its input, and in each iteration calculates a view into the pre-allocated output array corresponding to one of the tiles. The view is provided to the preceding operation as storage for the data it produces. Therefore, the data type of the input port is an array
view type, rather than a regular array type, and the input port has the data storage flag set so that it gets to provide storage.

If a Tile and Join Tiles operation are used together as generator and accumulator, respectively, of the same flow, the effect is that the operations in the inner flow will map each tile of the input array into the corresponding tile of the output array. A tiling transformation is thus performed by wrapping an existing array operation in a new flow that has a Tile generator and a Join Tiles accumulator, as shown in Figure 19. The subgraph that is wrapped should have a single input and a single output, each with an array type annotation. Fixed-size array type annotations are modified when wrapping to reflect the sizes of the input and output tiles.

![Tiling by wrapping a subgraph in a flow with a Tile generator and a Join Tiles accumulator.](image)

Figure 19: Tiling by wrapping a subgraph in a flow with a Tile generator and a Join Tiles accumulator.

Multiple levels of tiling are achieved by wrapping the subgraph several times, creating multiple levels of flows with Tile generators and Join Tiles accumulators. The outermost flow works with the largest tiles, which are split further into smaller tiles in the second level flow and so on.

4.1.2 Limitations

The Tile and Join Tiles operations are implemented with support only for statically sized arrays. Recall that static array sizes are achieved using type annotations. A consequence is that a type annotation will specify the size of the tiles produced by a Tile operation, and that size applies for all invocations of the inner flow. In other words, all tiles need to have the same size, which means that tiling can only be applied to arrays with a size that can be divided into (overlapping) tiles of equal size. This limitation is acceptable for demonstration purposes, but it would not be acceptable for general use.

In order to support tiling of arbitrary arrays the tile generator must be able to produce differently sized arrays in different
iterations. This could be achieved by implementing support for dynamically sized arrays in the tiling operations, though that could potentially sacrifice performance. To support tiling of arbitrary arrays while keeping the use of static array sizes, the dataflow language would need to be extended. One alternative is to allow a generator to instantiate its inner flow with different types in different iterations, though that would make the programming model more complicated.

4.1.3 Differences from hardcoded auto-tuner

The approach to tiling used in the dataflow solution follows from the nature of the paradigm and differs somewhat from the approach used in the hardcoded auto-tuner.

In the hardcoded auto-tuner the array is divided into equally sized, non-overlapping tiles. There is no need for the tiles to overlap because it is possible to access elements outside of the tile being processed, meaning that the stencil can be applied at the border of a tile. However, the original array borders need to be avoided since they lack neighboring elements, which means that the tiles adjacent to the original array borders (henceforth called edge tiles) need special handling.

In the dataflow language there is no way to access elements of neighboring tiles, because a tile is seen as self-contained data rather than a pointer to a location in a bigger array. To cater for this limitation all neighboring tiles need to overlap by an amount related to the stencil size. With this approach to tiling, elements that are close to a border belong to more than one tile and the stencil never needs to be applied on the border of any tile. All tile borders are avoided, whether they were original array borders or not, so there is no longer a need for special handling of the edge tiles.

The introduction of overlap in tiling leads to differences in the naturally supported array sizes. As an example, in a program produced by the hardcoded auto-tuner, an array of width 50 might be split into 5 tiles of width 10. When iterating through such a tile with a stencil of width 3, an extra column of elements will be accessed on each side so that 12 columns of data will be used, except in the edge tiles, where only 11 columns of data will be used. The closest possible corresponding example in the dataflow solution consists of 5 overlapping tiles of with 12, with an overlap of 2. However, this adds up to a total array width of 52, not 50. In general, the array sizes supported for tiling into equally sized tiles by the dataflow auto-tuner are larger than those supported by the hardcoded auto-tuner by the size of the stencil minus one. The real underlying difference is in the handling of the edge tiles, which are
special-cased in programs from the hardcoded auto-tuner but not so in the dataflow programs.

The hardcoded auto-tuner could have been implemented with a tiling approach similar to the one used by the dataflow auto-tuner to avoid these differences. However, changing the hardcoded auto-tuner to suit the dataflow auto-tuner is against the spirit of this work.

4.2 OTHER OPTIMIZATIONS

4.2.1 Parallelization

To support parallelization of array operations, a Parallel Tile operation is created. It functions like the regular Tile operation, but the array tiles are processed in parallel, i.e., the inner flow executions are concurrent. This is implemented by attaching an OpenMP pragma to the loop that iterates over tiles. Instantiation of a Parallel Tile operation requires three parameters: 1. the array dimension to tile, 2. the number of tiles, i.e., the number of threads to launch, and 3. the amount of overlap between tiles.

4.2.2 NUMA-aware memory allocation

NUMA-aware memory allocation can be achieved by parallelizing the initialization of an array in the same way that the main computation is parallelized.

4.2.3 Array padding

An array, or its data type annotation in the case of fixed-shape arrays, contains memory stride lengths for advancing an element in each dimension. Array padding is implemented by manipulating these stride lengths, as well as the initial offset and the total array length.

4.2.4 Cache bypass

Cache bypass for storing data is represented as a flag on the port that provides the data. If the flag is set, assignments to the output data are replaced with a call to a compiler intrinsic function that may result in a store instruction that bypasses the cache.

4.3 Locating Optimization Opportunities

An algorithm for automatically finding safe optimization opportunities was removed from the scope of the project. Instead, the
auto-tuner must be told where to try to apply the various transformations. This is done by supplying references to the parts of the dataflow graph that should be wrapped in tiling and/or parallelization flows, the output ports that may have cache bypass enabled, and the data type annotations that array padding may be applied to.

4.4 TIME MEASUREMENT

To allow tuning of a certain part of the program, dataflow operations that measure run time are defined. The auto-tuner can be told what part of the input program to wrap with these time measurement operations. The operation \textit{Start Timer} outputs the current value of the system clock, and the operation \textit{Read Timer} takes a clock value as input and outputs the time interval difference between the input value and the current clock value. The operation \textit{AutowTune} takes a numerical value as input and sends it to the auto-tuner as the score (i.e., run time or any other measurement that should be minimized) of the program.

4.5 AUTO-TUNING_STENCIL COMPUTATIONS

The stencil computation program used to evaluate the dataflow auto-tuner is based on the dataflow graph shown in Figure 18. The tiling, parallelization and cache bypass transformations described above can easily be applied to that dataflow graph. Array padding and NUMA-aware memory allocation, however, cannot be applied to the program as shown; this is because they both apply to operations that allocate and initialize an array, and the shown dataflow graph contained no such operations since it was assumed that an already initialized input array would be provided. Thus, in order to apply all the optimizations, the allocation and initialization of the input array needs to be part of the program.

The complete stencil computation dataflow program including initialization (and implicitly allocation) of the input array is shown in Figure 20. Note that type annotations are not visible in the graph visualization, but all the arrays in this example have fixed sizes. An Index Iterator operation is used to iterate over the coordinates of the array to be created. Each array element is initialized to a value based on the coordinates of its position in the array. The actual array creation is performed by the Assemble Elements accumulator that was described in section 3.6.

To auto-tune the program, it was first annotated to specify the optimizations that may be applied, as indicated by the annotation labels in Figure 20. Via these annotations, the auto-tuner is informed that a) the subgraph consisting of the Apply Stencil operation and its inner flow can be wrapped in a tiling flow; b) the same
Figure 20: Stencil computation dataflow program including input array initialization. This program was used as input to the auto-tuner. Tags are attached to the dataflow graph to show how the auto-tuner was told about optimization opportunities.
Figure 21: Tuned stencil computation program. This shows the same program as in Figure 20 after transformations have been applied by the auto-tuner. Repeated run time measurement operations used by the auto-tuner are also visible, but would normally be removed after tuning.
subgraph can be wrapped in a parallel tiling flow as well, while applying the same transformation to the subgraph consisting of the Index Iterator operation and its inner flow; c) the data type annotations of the input and output array may be manipulated to add padding; d) the output port of the Assemble Elements operation may have the cache bypass flag enabled; and finally e) the run time of the stencil computation part of the program should be measured and minimized when auto-tuning.

The auto-tuner then performs the following steps: 1. traverse the allowed parameter space, picking combinations of parameter values to evaluate; 2. for each configuration, apply transformations according to the parameters to generate a modified dataflow graph; 3. generate C code from the modified dataflow graph; 4. compile and run the generated C code and measure its run time.

4.6 RESULTS

The dataflow auto-tuner was evaluated by applying it to the stencil computation program shown in Figure 20 using an array size of $2^{10} + 2$ rows by $2^{19} + 2$ columns. While this is slightly larger than the array used in the hardcoded auto-tuner, it is the closest possible array size that is supported by the tiling approach used in the dataflow auto-tuner, as explained in section 4.1. The experiment was performed on the same two machines that were used to evaluate the hardcoded auto-tuner in section 2.5, and with an equivalent selection of allowed values for the transformation parameters.

The auto-tuner successfully applied the supported optimization transformations with all permitted parameter combinations, and evaluated the resulting candidate programs. The best candidate program found for the Opteron machine is displayed in Figure 21, showing the outcome of all the transformations that were applied by the auto-tuner. The C code generated from the unoptimized program in Figure 20 as well as the fully optimized program in Figure 21 is contained in Appendix A.

The performance improvements attained on each machine is presented in Figure 22, which shows results with a similar pattern to those obtained with the hardcoded auto-tuner in section 2.5, as expected. The speedup values are not identical to those of the hardcoded auto-tuner, however, with discernible differences between the two solutions on the AMD Santa Rosa architecture in particular. Likely contributing factors are the aforementioned difference in array size and tiling approach compared to the hardcoded auto-tuner.
Figure 22: Performance improvements achieved by dataflow auto-tuner compared to untuned dataflow program. The stacked bars separate the contribution to speedup from parallelization, tiling, array padding and cache bypass optimizations.
CONCLUSIONS

In this study I showed that auto-tuning is useful but currently difficult to utilize, and that the process can be simplified by expressing programs and optimizations using a high level dataflow representation.

At first I confirmed the usefulness of auto-tuning by demonstrating its use in applying architecture-dependent program optimizations to stencil computations. At the same time, I demonstrated that auto-tuning a program by building a program-specific code generator is a formidable task. This made it clear that there is a need for a general purpose auto-tuning framework.

I then created a high level dataflow model that allows straightforward representation of stencil computations. The language provides a simple and flexible iteration construct based on a hierarchy of flows, which proved useful by allowing composition of tiling and parallelization transformations and thus simple application of relevant optimizations.

By implementing an auto-tuner that manipulates dataflow programs, I also showed that using such a representation does simplify auto-tuning by removing the need for writing a program-specific code generator. The dataflow auto-tuning solution applied the same optimizations as the hardcoded auto-tuning solution, and provided similar performance improvements on multiple computer architectures, with less implementation work.

The presented dataflow auto-tuner did not fully eliminate manual work, as it requires information about optimization opportunities to be provided as input. This limitation prevents the solution from being generally useful at this point. The chosen iteration construct, while simple and flexible and allowing composition of optimizations, did not make it particularly easy to analyze programs in order to identify and validate optimization opportunities.

5.1 FUTURE WORK

To further reduce the effort required to auto-tune a program and allow scaling up to larger programs, an algorithm is needed for identifying where transformations can be applied, and verifying that transformations do not change the meaning of the program. Doing the latter would require, among other things, an analysis method to determine whether or not a dataflow operation that works on arrays is data parallel. It should be investigated if the iteration construct of the presented dataflow language allows such analysis.
It may be valuable to explore alternative iteration models, or using an existing data flow language with a strict iteration model.

The presented dataflow language needs to be developed further in order to be generally useful. Among other things, it needs support for automatic deallocation of allocated memory. Support for dynamically sized data should be improved, and the possibility of supporting generators that produce elements of different but statically known sizes in different iterations (as needed for supporting arbitrary array and tile sizes) could be explored.

The brute force search algorithm used by the auto-tuning solution could be replaced by a more sophisticated search algorithm to reduce auto-tuning time.

5.2 DISCUSSION

In trying to make the language simple but powerful I made it so that iteration, rather than being a strictly defined part of the language, was a basic operation that could be implemented by the program developer. This allowed optimizations such as parallelization, array tiling and array padding to be expressed as operations, avoiding the need for extra language constructs and keeping the language small and elegant. It also made it easy to apply the optimizations as transformations to an existing program, which is an important aspect when creating a general purpose auto-tuner. However, another important aspect is being able to automatically identify and validate optimization opportunities by determining where the various transformations can be safely applied. The language cannot be said to be the right representation for the problem until this second aspect has also been solved, and the chosen iteration construct may have made it more difficult to do so.

While the idea of completely controlling iteration inside generator operations made the optimization transformations simple to represent and added much flexibility to the language, it also made use of iteration somewhat opaque, thereby making it difficult to determine where the transformations can be safely applied. The program would need to be annotated with this information, and this puts the responsibility on the program developer.

In retrospect, it would have been better to start with one of the existing, stricter data flow languages, where iteration is completely explicit in the data flow graph. There it would be possible to reason about how a transformation such as tiling would affect the following operations.

The reasons for not using existing data flow languages was that they were considered limited and difficult to work with. However, the limitations stem from the strictness of those languages, and the strictness would likely have been helpful in achieving the immediate goal of this work. Implementing auto-tuning in a strict and
well-understood language with a working, efficient implementation would have been easier than simultaneously trying to develop a new and more flexible language, and might have been immediately useable and thus valuable within the problem domain served by the existing language. Language development to improve auto-tuning support further or to increase iteration flexibility could then have proceeded in a separate project.

I was trying to avoid influence from existing data flow languages because of the idea that I should find the minimal information needed in a language to support auto-tuning, and that starting with an existing language would bring existing irrelevant baggage. The goal was not to create a real-world useable data flow language, and basing the work on one of those was therefore considered a distraction, while starting from scratch could allow me to focus on what is needed for auto-tuning.

However, I think that in reality the opposite is true. Starting from scratch without being an expert in existing data flow languages meant that I had to spend most of the time designing the language, trying out different constructs, trying to make it simple yet general. The number of open questions when designing a language from scratch without relevant experience is enormous, and without clear requirements on the language (or actual real world users) it is difficult to know when it is good enough, or if the basic ideas are even going in the right direction. I therefore kept coming back to the basic design, without a fixed ground to stand on, taking up time that could have been better spent focusing on the problem of finding and validating optimization opportunities.
Part II

APPENDIX
A.1 HARDCODED AUTO-TUNER

A.1.1 Before auto-tuning

The following code was produced by the hardcoded auto-tuner with no transformations enabled. It performs a four-point stencil computation on a \(2^{10}\) row by \(2^{10}\) column array of 32-bit integers, writing to an equally sized output array.

C code generated by the hardcoded auto-tuner before adding any optimizations.

```c
// Allocation
int *const restrict input = alloc_array(4096, 536870912 * sizeof(int));
int *const restrict output = alloc_array(4096, 536870912 * sizeof(int));

// Initialization
const int y1_start = 0;
const int y1_stop = 1024;
const int x1_start = 0;
const int x1_stop = 524288;
for (int y = y1_start; y < y1_stop; y += 1) {
    for (int x = x1_start; x < x1_stop; x += 1) {
        input[1 * x + 524288 * y] = x ^ y;
        output[1 * x + 524288 * y] = 0;
    }
}

// Repeat experiment
double min_real_time = INFINITY;
for (int i = 0; i < 9; i++) {
    // Check start time
    real_time_t real_time_start, real_time_end;
    get_real_time(&real_time_start);

    // Stencil computation
    const int y1_start = 1;
    const int y1_stop = 1023;
    const int x1_start = 1;
    const int x1_stop = 524287;
    const int base = 1 * x1_start + 524288 + y1_start;
    int *o0 = &output[base + 0];
    const int *i0 = &input[base - 1];
    const int *i1 = &input[base - 524288];
    const int *i2 = &input[base + 524288];
    const int *i3 = &input[base + 1];
    const int dx = 1;
```

const int dy = 524288 - (x1_stop - x1_start);

for (int y = y1_start; y < y1_stop; y += 1) {
    for (int x = x1_start; x < x1_stop; x += 1) {
        *o0 = *i0 ^ *i1 ^ *i2 ^ *i3;
        o0 += dx; i0 += dx; i1 += dx; i2 += dx; i3 += dx;
    }
    o0 += dy; i0 += dy; i1 += dy; i2 += dy; i3 += dy;
}

// Check end time, calculate run time
get_real_time(real_time_end);
double real_time = real_time_diff_to_seconds(real_time_end,
                                           real_time_start);
min_real_time = real_time < min_real_time ? real_time :
                                           min_real_time;
}

// Output
printf("Real time: %f\n", min_real_time);

A.1.2 After auto-tuning

The following code was produced by the hardcoded auto-tuner when tuning for the AMD Santa Rosa machine. It performs the same stencil computation as the previous example, but with transformations applied to enable parallelization on 4 threads, tiling into $2^6$ row by $2^4$ column blocks, array padding by 7 columns, and cache bypass using streaming store instructions.

C code generated by the hardcoded auto-tuner after tuning for the Opteron machine.

// Allocation
int *const restrict input = alloc_array(4096, 536878080 * sizeof(int));
int *const restrict output = alloc_array(4096, 536878080 * sizeof(int));

// Initialization
const int y2_start = 0;
const int y2_stop = 1024;
const int x2_start = 0;
const int x2_stop = 524288;
uint8_t cpus[4] = {0, 1, 2, 3};
#pragma omp parallel num_threads(4) default(shared)
{
    const int thread_num_y = omp_get_thread_num();
    pin_current_thread_to_cpu(cpus[thread_num_y]);
    #pragma omp for schedule(static)
    for (int y1_start = y2_start; y1_start < y2_stop; y1_start += 64) {
        const int y1_stop = y1_start + 64 < y2_stop ? y1_start + 64 : y2_stop;
for (int x1.start = x2.start; x1.start < x2.stop; x1.start += 2048) {
    const int x1.stop = x1.start + 2048 < x2.stop ?
        x1.start + 2048 : x2.stop;
    for (int y = y1.start; y < y1.stop; y += 1) {
        for (int x = x1.start; x < x1.stop; x += 1) {
            _mm_stream_si32(&input[1 * x + 524295 * y], x ^ y);
            _mm_stream_si32(&output[1 * x + 524295 * y], 0);
        }
    }
}

// Repeat experiment
double min_real_time = INFINITY;
for (int i = 0; i < 9; i++) {
    // Check start time
    real_time_t real_time_start, real_time_end;
    get_real_time(real_time_start);

    // Stencil computation
    const int y2.start = 1;
    const int y2.stop = 1023;
    const int x2.start = 1;
    const int x2.stop = 524287;

    uint8_t cpus[4] = {0, 1, 2, 3};
    #pragma omp parallel num_threads(4) default(shared)
    {
        const int thread_num_y = omp_get_thread_num();
        pin_current_thread_to_cpu(cpus[thread_num_y]);
        #pragma omp for schedule(static)
        for (int y1.start = y2.start; y1.start < y2.stop; y1.start += 64) {
            const int y1.stop = y1.start + 64 < y2.stop ?
                y1.start + 64 : y2.stop;
            for (int x1.start = x2.start; x1.start < x2.stop;
                x1.start += 2048) {
                const int x1.stop = x1.start + 2048 < x2.stop ?
                    x1.start + 2048 : x2.stop;

                const int base = 1 * x1.start + 524295 * y1.start;
                int * o0 = &output[base + 0];
                const int * i0 = &input[base - 1];
                const int * i1 = &input[base - 524295];
                const int * i2 = &input[base + 524295];
                const int * i3 = &input[base + 1];
                const int dx = 1;
                const int dy = 524295 - (x1.stop - x1.start);

                for (int y = y1.start; y < y1.stop; y += 1) {
                    for (int x = x1.start; x < x1.stop; x += 1) {
                        _mm_stream_si32(o0, *i0 ^ *i1 ^ *i2 ^ *i3);
                        o0 += dx; i0 += dx; i1 += dx; i2 += dx; i3 += dx;
                    }
                }
            }
        }
    }
}
A.2 DATAFLOW AUTO-TUNER

A.2.1 Before auto-tuning

The following code was generated from the dataflow stencil computation program with no tuning enabled. It performs a four-point stencil computation on a \(2 \times 0 + 2\) row by \(2 \times 9 + 2\) column array of 32-bit integers, writing to an equally sized output array.

C code generated for the stencil computation dataflow program in Figure 20 before auto-tuning.

```c
// Storage preparation for Constant operation
int value_1 = 0;

// Constant (operation)

// Storage preparation for Assemble Elements output, Apply Stencil input
int * assemblage_1 = calloc(524290 * 1026, sizeof(int));

// Index Iterator (generator)
for (int array_idx_1 = 0; array_idx_1 < 1026; array_idx_1 += 1) {
    for (int array_idx_2 = 0; array_idx_2 < 524290; array_idx_2 += 1) {
        int y_1 = value_1 + array_idx_2;
        int x_1 = value_1 + array_idx_1;

        // Assemble Elements (accumulator, input storage preparation)
        int result_1;

        // XOR (operation)
        result_1 = x_1 ^ y_1;

        // Assemble Elements (accumulator, main step)
    }
}
```
assemblage_1[0 + 524290 * array_idx_1 + 1 + array_idx_2] = 
result_1;

} // Storage preparation for Assemble Elements output
int * assemblage_2 = calloc(524290 * 1026, sizeof(int));

// Apply Stencil (generator)
for (int stencil_idx_1 = 0; stencil_idx_1 < 1024; stencil_idx_1 += 1)
  for (int stencil_idx_2 = 0; stencil_idx_2 < 524288; 
    stencil_idx_2 += 1) {
    int bottom_1 = assemblage_1[0 + 524290 * (stencil_idx_1 + 2) + 1 * (stencil_idx_2 + 1)];
    int right_1 = assemblage_1[0 + 524290 * (stencil_idx_1 + 1) 
      + 1 * (stencil_idx_2 + 2)];
    int top_1 = assemblage_1[0 + 524290 * stencil_idx_1 + 1 + 
      1 * stencil_idx_2];
    int left_1 = assemblage_1[0 + 524290 * (stencil_idx_1 + 1) 
      + 1 * stencil_idx_2];

    // Assemble Elements (accumulator, input storage preparation)
    int result_2;

    // Kernel (operation)
    result_2 = top_1 ^ left_1 ^ right_1 ^ bottom_1;

    // Assemble Elements (accumulator, main step)
    assemblage_2[524290 + 1 + 524290 * stencil_idx_1 + 1 *
      stencil_idx_2] = result_2;
  }

A.2.2 After auto-tuning

The following code was generated from the dataflow stencil computation program after tuning by the dataflow auto-tuner for the Intel Nehalem architecture. It performs the same stencil computation as the previous example, but with transformations applied to enable parallelization over 8 threads, tiling into $2^7$ row by $2^{14}$ column blocks, array padding by 7 columns, and cache bypass using streaming store instructions.

C code generated for the stencil computation dataflow program after auto-tuning for Opteron. This corresponds to the dataflow graph in Figure 21.

// Storage preparation for Constant output, Index Iterator input
int value_1;

// Constant (operation)
value_1 = 0;

// Storage preparation for Join Tiles output, Repeat input
int * assemblage_1 = calloc(524297 * 1026, sizeof(int));
// Parallel Index Tile (generator)
#pragma omp parallel for num_threads(8) schedule(static) default(shared)
for (int tile_offset_1 = 0; tile_offset_1 < 1024; tile_offset_1 += 128) {
    int y_1 = value_1;
    int x_1 = value_1 + tile_offset_1;

    // Join Tiles (accumulator, input storage preparation)
    int * tile_1 = &assemblage_1[0 + 524297 * tile_offset_1 + 1 * 0];

    // Index Iterator (generator)
    for (int array_idx_1 = 0; array_idx_1 < 130; array_idx_1 += 1)
        for (int array_idx_2 = 0; array_idx_2 < 524290; array_idx_2 += 1) {
            int y_2 = y_1 + array_idx_2;
            int x_2 = x_1 + array_idx_1;

            // Assemble Elements (accumulator, input storage preparation)
            int result_1;

            // XOR (operation)
            result_1 = x_2 ^ y_2;

            // Assemble Elements (accumulator, main step)
            tile_1[524297 * array_idx_1 + 1 * array_idx_2] = result_1;
        }
}

// Storage preparation for Last output
int * item_1 = calloc(524297 * 1026, sizeof(int));

// Storage preparation for Min output
double min_1;

// Min (accumulator, preparation step)
min_1 = INFINITY;

// Repeat (generator)
for (int i_1 = 0; i_1 < 9; i_1 += 1) {
    int * item_2 = assemblage_1;

    // Storage preparation for Start Timer output
    real_time_t timer_1;

    // Start Timer (operation)
    get_real_time(timer_1);

    // Last (accumulator, input storage preparation)
    int * item_3 = item_1;

    // Parallel Tile (generator)
    #pragma omp parallel for num_threads(8) schedule(static)
    default(shared)
```c
for (int tile_offset_2 = 0; tile_offset_2 < 1024; tile_offset_2 += 128) {
    int * tile_2 = &item_2[0 + 524297 * tile_offset_2 + 1 * 0];

    // Join Tiles (accumulator, input storage preparation)
    int * tile_3 = &item_3[524297 + 1 + 524297 * tile_offset_2 + 1 * 0];

    // Tile (generator)
    for (int tile_offset_3 = 0; tile_offset_3 < 128;
         tile_offset_3 += 128) {
        int * tile_4 = &tile_2[524297 * tile_offset_3 + 1 * 0];

        // Join Tiles (accumulator, input storage preparation)
        int * tile_5 = &tile_3[524297 * tile_offset_3 + 1 * 0];

        // Tile (generator)
        for (int tile_offset_4 = 0; tile_offset_4 < 524288;
             tile_offset_4 += 16384) {
            int * tile_6 = &tile_4[524297 + 0 + 1 *
                                   tile_offset_4];

            // Join Tiles (accumulator, input storage preparation)
            int * tile_7 = &tile_5[524297 + 0 + 1 *
                                   tile_offset_4];

            // Apply Stencil (generator)
            for (int stencil_idx_1 = 0; stencil_idx_1 < 128;
                 stencil_idx_1 += 1) {
                for (int stencil_idx_2 = 0; stencil_idx_2 <
                     16384; stencil_idx_2 += 1) {
                    int bottom_1 = tile_6[524297 * {
                                                   stencil_idx_1 + 2) + 1 * (stencil_idx_2 + 1)];
                    int right_1 = tile_6[524297 * {
                                                   stencil_idx_1 + 1) + 1 * (stencil_idx_2 + 2)];
                    int top_1 = tile_6[524297 + stencil_idx_1 +
                                        1 * (stencil_idx_2 + 1)];
                    int left_1 = tile_6[524297 + (stencil_idx_1 +
                                               1) + 1 * stencil_idx_2];

                    // Assemble Elements (accumulator, input
                      storage preparation)
                    int result_2;

                    // Kernel (operation)
                    result_2 = top_1 ^ left_1 ^ right_1 ^
                                      bottom_1;

                    // Assemble Elements (accumulator, main
                      step)
                    _mm_stream_si32(&tile_7[524297 +
                                      stencil_idx_1 + 1 * stencil_idx_2],
                                      result_2);
                }
            }
        }
    }
}
```
```c
}

// Min (accumulator, input storage preparation)
double time_1;

// Read Timer (operation)
real_time_t timer_3;
get_real_time(timer_3);
time_1 = real_time_diff_to_seconds(timer_3, timer_1);

// Last (accumulator, main step)
item_1 = item_3;

// Min (accumulator, main step)
if (time_1 < min_1)
    min_1 = time_1;

// Auto-Tune (operation)
printf("Real time: %f\n", min_1);
```
BIBLIOGRAPHY


58


