# MacroSS: Macro-SIMDization of Streaming Applications

Amir H. Hormati[1], Yoonseo Choi[1], Mark Woh[1], Manjunath Kudlur[2], Rodric Rabbah[3], Trevor Mudge[1], and Scott Mahlke[1]

[1]Advanced Computer Architecture Laboratory
University of Michigan - Ann Arbor, MI
{hormati, yoonseo, mwoh, tnm, mahlke}@umich.edu

[2]NVIDIA Corp.
Santa Clara, CA
mkudlur@nvidia.com

[3] IBM T.J. Watson Research Center
Hawthorne, NY
rabbah@us.ibm.com

## Abstract

*SIMD (Single Instruction, Multiple Data) engines are an essential part of the processors in various computing markets, from servers to the embedded domain. Although SIMD-enabled architectures have the capability of boosting the performance of many application domains by exploiting data-level parallelism, it is very challenging for compilers and also programmers to identify and transform parts of a program that will benefit from a particular SIMD engine. The focus of this paper is on the problem of SIMDization for the growing application domain of streaming. Streaming applications are an ideal solution for targeting multi-core architectures, such as shared/distributed memory systems, tiled architectures, and single-core systems. Since these architectures, in most cases, provide SIMD acceleration units as well, it is highly beneficial to generate SIMD code from streaming programs. Specifically, we introduce MacroSS, which is capable of performing macro-SIMDization on high-level streaming graphs. Macro-SIMDization uses high-level information such as execution rates of actors and communication patterns between them to transform the graph structure, vectorize actors of a streaming program, and generate intermediate code. We also propose low-overhead architectural modifications that accelerate shuffling of data elements between the scalar and vectorized parts of a streaming program. Our experiments show that MacroSS is capable of generating code that, on average, outperforms scalar code compiled with the current state-of-art auto-vectorizing compilers by 54%. Using the low-overhead data shuffling hardware, performance is improved by an additional 8% with less than 1% area overhead.*

***Categories and Subject Descriptors*** D.3.4 [*Programming Languages*]: Processors—Compilers ; C.1.2 [*Processors Architectures*]: Multiple Data Stream Architectures—Single-instruction-stream, multiple-data-stream processors (SIMD)

***General Terms*** Design, Languages, Performance

***Keywords*** Streaming, Compiler, SIMD Architecture, Optimization

## 1. Introduction

Support for parallelism in hardware has greatly evolved in the past decade as a response to the ever-increasing demand for higher performance and better power efficiency in different application domains. Various companies have introduced vastly different solutions to bridge the performance and power gap that many applications are facing. These solutions include shared-memory multicore systems (Intel Core i7 [13]), distributed-memory multicore processors (IBM Cell [11]), tiled architectures (Tilera [30]) and in some cases a combination of these (Intel Larrabee [27]). These architectures not only achieve higher performance and efficiency by combining multiple cores into one die, but they are also equipped with one or more single-instruction-multiple-data (SIMD) engines to enable more efficient data-level parallelism support for several important application domains such as multimedia, graphics, and encryption. SIMD engines are not suitable for all applications, but if an application can be tailored to efficiently exploit them, the performance and power benefits can often be superior to the gains from other architecture solutions. Therefore, SIMD engines like Altivec [28], Neon [4], SSE4 [12] are now an essential part of most architectures on the market. With SIMD width expanding in future architectures, such as Intel's Larrabee, under-utilization of the SIMD units would translate into a loss in performance and also power consumption.

Traditional sequential programming languages are ill-suited to exploit parallel architectures because they have a single instruction stream and a monolithic memory. Extracting task/pipeline/data-level parallelism from these languages needs extensive and often intractable compiler analysis. Using techniques targeted for a certain class of architectures is also undesirable because it limits the flexibility and *retargetability* of the program in that *each program needs to be rewritten and optimized for a specific architecture*. Data-parallel programming languages like OpenCL [19] and CUDA [24] that target data-parallel architectures like GPUs expose parallelism to the compiler, but in their current form fail to provide retargetable code. The main problem with these languages is that explicitly-programmed parallelism in each application has to be tuned for different targets based on the memory size and number of processing elements. To deal with these problems, the streaming programming paradigm provides an extensive set of compiler optimizations for mapping and scheduling applications to various parallel architectures ([9, 10]). The retargetability of streaming languages, such as StreamIt [29], has made them a good choice for parallel system programmers.

Streaming language retargetability and performance benefits on multi-core systems are mainly a result of having well-encapsulated constructs that expose the parallelism and communication without depending on the topology or granularity of the underlying architecture. Currently, streaming compilers map the abundance of task, pipeline, and data-level parallelism that exist within an application into task-level parallelism across multiple cores but not into data-level parallelism on SIMD engines. Mapping parallelism onto multi-core provides reasonable speedup for streaming applications but can also experience slowdown due to inter-core communication overhead and high memory/cache traffic. Utilizing SIMD engines is preferred, even for applications where multi-core speedup is close to the theoretical maximum, because SIMD engines can improve performance without increasing communication overhead and memory/cache traffic. Exploiting SIMD engines, in some cases, can achieve greater performance than multi-core while using less area and power.

Extending the retargetability of streaming languages for multi-core systems by adding effective SIMD support to their compilers is desirable because of the variation in characteristics of SIMD accelerators between different standards, such as number of lanes, memory interface, and scalar/vector transfers. Implementing and porting applications between different architectures can be difficult and error-prone. Therefore, efficiently vectorizing stream programs is essential to expand their applicability as a universal programming paradigm for current and future single/multicore architectures with various wide or narrow SIMD units.

*ASPLOS'10,* March 13–17, 2010, Pittsburgh, Pennsylvania, USA.
Copyright © 2010 ACM 978-1-60558-839-1/10/03. . . $10.00

To exploit SIMD engines, current streaming compilers translate the streaming languages down to an intermediate language, such C++ or Java, and then apply vectorization[1] techniques to generate SIMD-enabled code. The most popular techniques are hand-optimizing the code and traditional auto-SIMDization [1–3, 16, 23]. Both of these solutions have proven difficult to apply in real world scenarios. Hand-optimizing the binary or sequential code using architecture-specific instructions or intrinsic functions is a time-consuming and error-prone task which results in an inflexible and unportable binary. Auto-vectorization is, at this stage, still impractical and far from being able to universally utilize the various kinds of available SIMD facilities. Also, performing SIMDization on streaming applications after intermediate-level code generation may result in an inefficient schedule and mapping of the stream graph since the schedule is already fixed and information that is available in the high-level stream graph is lost. Extracting this information from the generated code is predicated on performing complex compiler analysis and transformations which are impossible in some cases. In summary, *lack of global knowledge about the program*, *inability to adjust the schedule*, and also *loss of data flow information* are the main reasons behind inefficiency of traditional auto-vectorization techniques in dealing with streaming applications.

In this work, we introduce *MacroSS*; a streaming compiler for the StreamIt language that is capable of performing macro-SIMDization on stream graphs. Macro-SIMDization uses high-level information such as the valid set of schedules and communication patterns between actors to transform the graph structure, vectorize actors of a streaming program, and generate intermediate code (C++ in this work). Then, it uses the host compiler to compile the generated intermediate code to binary for a specific target processor. The information that is used by MacroSS is deduced from the high-level program structure and is not available to low-level traditional compilers that are used to compile the intermediate code. As a result, MacroSS has a broader understanding of the program structure and macro-level characteristics of the streaming application that allows the compiler to utilize SIMD engines more efficiently.

MacroSS is capable of performing single-actor, vertical, and horizontal SIMDization of actors. Single-actor SIMDization targets each SIMDizable actor separately and transforms consecutive sequential executions of a SIMDizable actor to data-parallel executions on the SIMD engine. Vertical SIMDization fuses a pipeline of vectorizable actors to build a larger vectorizable actor and reduces the scalar-to-vector (packing)/vector-to-scalar (unpacking) overhead that exists between actors. Our experiments show that vertical SIMDization is applicable in many cases and can significantly improve performance by eliminating the need for translating back and forth between scalar and vector. Finally, horizontal SIMDization takes a set of isomorphic task parallel actors and replaces them with one or more data parallel actors. The choice of which vectorization technique to apply to a stream graph is based on the internal target-specific cost model and the structure of the graph. After SIMDization, MacroSS is able to generate architecture-specific intermediate code with SIMD intrinsics. This intermediate code uses vector types and intrinsics specific to the target architecture and can be compiled using the host compiler.

Packing of scalar values to a vector or unpacking a vector to scalar values typically takes between a couple of cycles to tens of cycles depending on the architecture. Since communicating data between vectorized and scalar actors or vice versa needs several packing/unpacking operations, MacroSS is equipped with two techniques to optimize this costly communication overhead. The first technique tries to replace the packing/unpacking operations with permutation instructions in actors that, during each execution, read or write $2^n$ elements. In the second technique, we introduce a low-overhead dynamic shuffler called the streaming address generation unit (SAGU). This unit eliminates the need to perform complicated address translations, data alignment, and packing/unpacking of data as data crosses vector-scalar boundaries of the graph.

[1] In this paper, we use SIMD(ize) and Vector(ize) interchangeably.

To summarize, this paper makes the following contributions:

- Introduction of macro-level SIMDization techniques for streaming languages: single actor, vertical and horizontal SIMDization. Based on these techniques, MacroSS compiler for the StreamIt language is implemented.
- Hardware and permutation-based tape optimizations for reducing the overhead of scalar-to-vector and vector-to-scalar data conversions.
- Evaluation of MacroSS on various streaming workloads from the StreamIt benchmark suite [29] on the Intel Core i7.

The rest of the paper is organized as follows. In Section 2, the stream programming model and the input language (StreamIt) are discussed. Macro-SIMDization and the related optimizations in MacroSS are explained in Section 3. Section 4 includes a brief discussion about the differences between traditional auto-vectorization and macro-SIMDization. Experiments are shown in Section 5. Finally, in Section 6, we discuss related works.

## 2. Stream Programming Model

With the ubiquity of multi-core systems, the stream programming paradigm has become increasingly important. Exposed communication and abundance of parallelism are the key features making streaming a flexible and architecture-independent solution for parallel programming. In this paper, we focus on stream programming models that are based on synchronous data flow (SDF) models [17]. In SDF, computation is performed by actors, which are autonomous and isolated computational units. Actors communicate through data-flow channels (i.e. tapes), often realized as FIFOs. SDF and its many variations expose the input and output processing rates of actors, and in turn this affords many optimization opportunities that can lead to efficient schedules (e.g., allocation of actors to cores, and tapes to local memories).

For our purpose, we assume all computation that is performed in an actor is largely embodied in a *work* method. The *work* method runs repeatedly as long as the actor has data to consume on its input port. The amount of data that the work method consumes is called the *pop* rate. Similarly, the amount of data each work invocation produces is called the *push* rate. Some streaming languages (e.g., StreamIt [29]) provide a non-destructive read which does not alter the state of the input channel. The amount of data that is read in this manner is specified by the *peek* rate. An actor can also have an *init* method that is executed only once for the purpose of initializing the actor before the execution of program starts.

We distinguish between stateful and stateless actors. A stateful actor modifies its local state and maintains a persistent history of its execution. Unlike a stateful actor, which restricts opportunities for parallelism, a stateless actor is data-parallel in that every invocation of the work method does not depend on or mutate the actor's state. The semantics of stateless actors thus allow us to replicate a stateless actor. This opportunity is quite fruitful in scaling the amount of parallelism that an application can exploit, as shown in [9, 10].

We use the StreamIt programming language to implement streaming programs. StreamIt is an architecture-independent streaming language based on SDF. The language allows a programmer to algorithmically describe the computational graph. In StreamIt, actors are known as filters. Filters can be organized hierarchically into *pipelines* (i.e., sequential composition), *split-joins* (i.e., parallel composition), and *feedback loops* (i.e., cyclic composition). StreamIt is a convenient language for describing streaming algorithms, and its accompanying static compilation technology makes it suitable for our work.

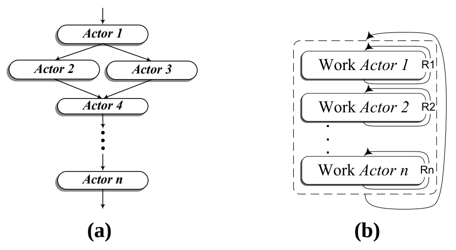


**Figure 1:** *This figure shows an example stream graph and also the intermediate code template for executing steady state schedule. $R_i$ is the repetition number for actor i.*

A crucial consideration in StreamIt programs is to create a steady state schedule which involves rate-matching of the stream graph. Rate-matching guarantees that, in the steady state, the number of data elements that is produced by an actor is equal to the number of data elements its successors will consume. Rate-matching assigns a static repetition number to each actor. In the implementation of a StreamIt schedule, an actor is enclosed by a *for-loop* that iterates as many times as its repetition number. The steady state schedule is a sequence of appearances of these *for-loops* enclosed in an outer-loop whose main job is to repeat the steady schedule. The template code in Figure 1b shows the intermediate code for the steady state schedule of the streaming graph shown in Figure 1a.

## 3. Macro-SIMDization

The SIMDization path in MacroSS consists of several steps to make the streaming graph more amenable to vectorization, tune the steady state schedule, vectorize actors, and perform target-specific code generation.

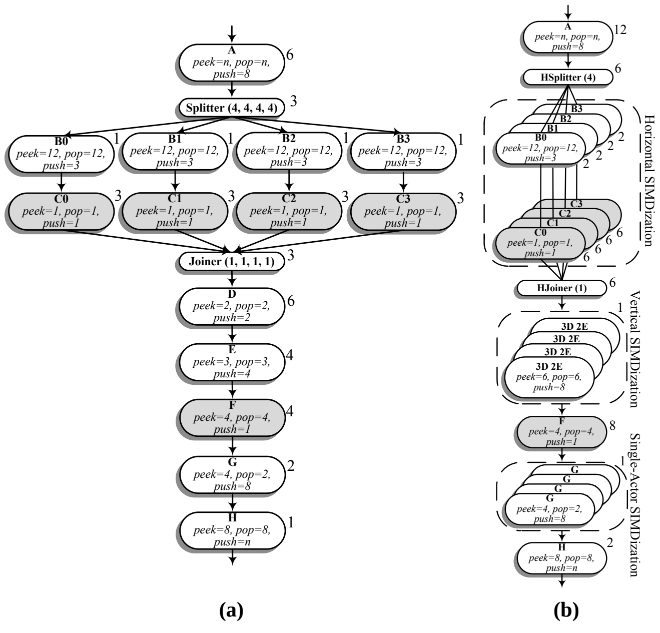


**Figure 2:** *Part (a) of this figure shows the stream graph used as a running example in this paper. Part (b) shows the same stream graph after MacroSS has SIMDized it.*

MacroSS is equipped with three main techniques: *Single-Actor*, *Vertical*, and *Horizontal SIMDization*. Single actor SIMDization targets each stateless actor separately. The goal is to convert multiple (equal to SIMD-Width) consecutive executions of a SIMDizable actor into one data-parallel execution on the target SIMD engine. Single-actor SIMDization leaves the input and output tapes of a vectorized actor as scalar and does not convert the tape accesses to vector since complicated shuffle operations must be introduced in the code in case vector tape accesses are used. The scalar tapes introduce packing/unpacking overheads in each SIMDized actor. Vertical SIMDization, which is a more optimized way of performing single-actor SIMDization on a pipeline of vectorizable actors, reduces this overhead. It enables MacroSS to implement vector communication between the actors of a SIMDizable pipeline. Both single-actor and vertical SIMDization try to convert sequential execution of a single actor or a pipeline of actors to data-parallel execution. The third technique, horizontal SIMDization, converts task parallelism into data parallelism for a group of isomorphic actors (stateful or stateless) in a stream graph. Horizontal SIMDization is mainly beneficial in cases where a group of several isomorphic actors are placed between a splitter and joiner and it is not possible to fuse these actors into one coarse actor to perform vertical SIMDization. MacroSS finds the parts of a graph that are suitable for this kind of SIMDization and converts the eligible task-parallel actors into one or more SIMD actors.

The stream graph illustrated in Figure 2a is used as a running example to explain different actions that the compiler takes to perform macro-SIMDization. This graph shows the structure of a streaming application with 10 unique actors. Each box shows one actor in the program. Each edge in this graph indicates a tape implemented using FIFO queues. The text written inside each box shows how each actor interacts with its input and output tapes. Each shaded box represents a stateful actor. On the right side of each node, the repetition number of that node in the steady state is shown. Even though MacroSS is able to target processors equipped with SIMD engines with any SIMD width, for the sake of presentation, the target hardware platform to which MacroSS compiles is set to a core with SIMD width of four 32-bit data types, and main memory line width of 128-bit. Figure 2b shows how MacroSS vectorizes the streaming graph. The *split-join* structure is horizontally vectorized. The task-parallel actors between the splitter and joiner are converted to SIMD actors and the splitter and joiner are replaced with horizontal versions. Actors $D$ and $E$ are vertically fused and SIMDized. Single-actor SIMDization is applied to actor $G$.

The details of how MacroSS performs SIMDization on a streaming graph are explained in the following three subsections. Next, in Section 3.4, the way MacroSS deals with SIMDization of tapes in the presence of architectural support is explained. Finally, Section 3.5 explains the overall structure of the macro-SIMDization technique in MacroSS.

### 3.1 Single-Actor SIMDization

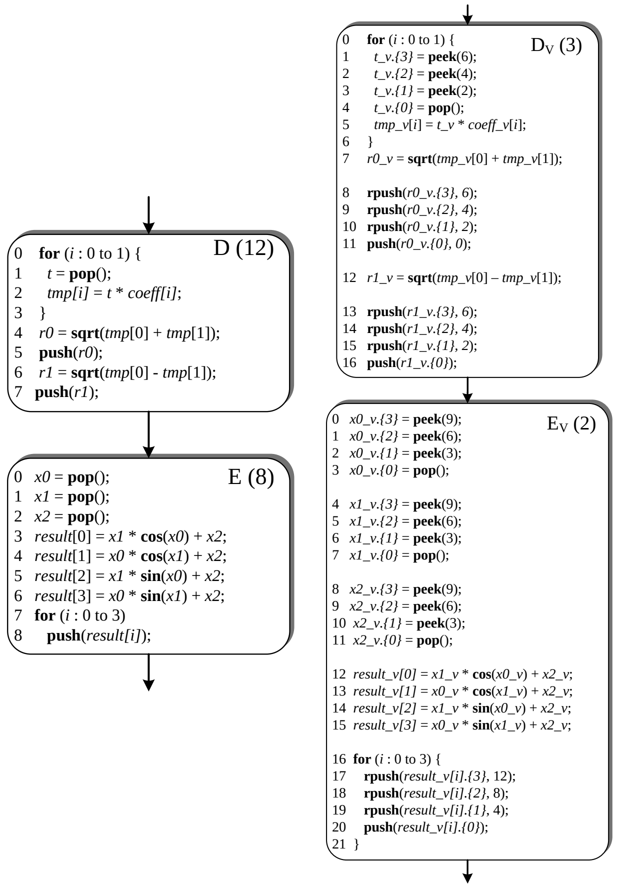
Let $SW$ denote the SIMD width of the target machine. The goal of single-actor SIMDization is to run $SW$ consecutive executions of an actor in data-parallel fashion using the target SIMD engine. As mentioned before, actors in a StreamIt program execute based on a steady state schedule in which each actor is enclosed by a *for-loop* that iterates as many times as its repetition number (see Figure 1b). Conceptually, single-actor SIMDization is similar to vectorizing the actor's enclosing *for-loop* whose trip count is the repetition number of the actor. Therefore, MacroSS adjusts the repetition numbers of all actors to make them multiples of $SW$ before single-actor SIMDization.

MacroSS finds the smallest factor that the repetition number of each vectorizable actor should be multiplied by based on the following equation:

$$M = Max\{\frac{LCM(SW, R_i)}{R_i}, \quad \forall \text{ SIMDizable actor } A_i\} \quad (1)$$

Each term of the $Max$ function finds the smallest factor that each repetition number ($R_i$) should be multiplied by to make it a multiple of $SW$. After finding the minimum for each SIMDizable actor, the largest factor is chosen and all of the repetition numbers are scaled based on that. According to Equation (1), the repetition numbers of the graph in Figure 2a must be scaled by 2 (= $M$) before SIMDization.

Suppose that, after this adjustment of the repetition numbers, the resulting repetition number of an actor $A$ is $m \times SW$. Then, MacroSS transforms the $m \times SW$ sequential executions of $A$ into $m$ sequential executions of $SW$ data-parallel $A$'s. Since several executions of the SIMDized actor will be running at the same time, only stateless actors are eligible for single-actor SIMDization. This kind of SIMDization can be applied to actors $D$, $E$, and $G$ in the example shown in Figure 2a. The code in Figure 3 illustrates how single-actor SIMDization is performed for actors $D$ and $E$. Ignoring the tape accesses, it can be seen that the variables in the original actors are packed into vector variables and computation functions are calculated on vector variables instead of scalar. Vector variables are depicted by _v suffix as in `tmp_v[]`, `t_v` and `coeff_v[]`. Actors $D$ and $E$ originally had repetition number of 12 and 8 and after SIMDization are executed 3 times and 2 times since each execution of the vectorized actors is in fact 4 data-parallel executions of the original actors.



```
0   for (i : 0 to 1) {
1     t = pop();
2     tmp[i] = t * coeff[i];
3   }
4   r0 = sqrt(tmp[0] + tmp[1]);
5   push(r0);
6   r1 = sqrt(tmp[0] - tmp[1]);
7   push(r1);
```

```
0  x0 = pop();
1  x1 = pop();
2  x2 = pop();
3  result[0] = x1 * cos(x0) + x2;
4  result[1] = x0 * cos(x1) + x2;
5  result[2] = x1 * sin(x0) + x2;
6  result[3] = x0 * sin(x1) + x2;
7  for (i : 0 to 3)
8    push(result[i]);
```

```
0   for (i : 0 to 1) {
1     t_v.{3} = peek(6);
2     t_v.{2} = peek(4);
3     t_v.{1} = peek(2);
4     t_v.{0} = pop();
5     tmp_v[i] = t_v * coeff_v[i];
6   }
7   r0_v = sqrt(tmp_v[0] + tmp_v[1]);

8   rpush(r0_v.{3}, 6);
9   rpush(r0_v.{2}, 4);
10  rpush(r0_v.{1}, 2);
11  push(r0_v.{0}, 0);

12  r1_v = sqrt(tmp_v[0] – tmp_v[1]);

13  rpush(r1_v.{3}, 6);
14  rpush(r1_v.{2}, 4);
15  rpush(r1_v.{1}, 2);
16  push(r1_v.{0});
```

```
0  x0_v.{3} = peek(9);
1  x0_v.{2} = peek(6);
2  x0_v.{1} = peek(3);
3  x0_v.{0} = pop();

4  x1_v.{3} = peek(9);
5  x1_v.{2} = peek(6);
6  x1_v.{1} = peek(3);
7  x1_v.{0} = pop();

8  x2_v.{3} = peek(9);
9  x2_v.{2} = peek(6);
10 x2_v.{1} = peek(3);
11 x2_v.{0} = pop();

12 result_v[0] = x1_v * cos(x0_v) + x2_v;
13 result_v[1] = x0_v * cos(x1_v) + x2_v;
14 result_v[2] = x1_v * sin(x0_v) + x2_v;
15 result_v[3] = x0_v * sin(x1_v) + x2_v;

16 for (i : 0 to 3) {
17   rpush(result_v[i].{3}, 12);
18   rpush(result_v[i].{2}, 8);
19   rpush(result_v[i].{1}, 4);
20   push(result_v[i].{0});
21 }
```

**Figure 3:** *This figure shows how single-actor SIMDization transforms actors $D$ and $E$ into $D_V$ and $D_E$. All the vector variables are concatenated with _v at the end. Part (a) of this figure shows the code for actors $D$ and $E$ in scalar mode. Part (b) illustrates the vectorized version of actors $D$ and $E$.*

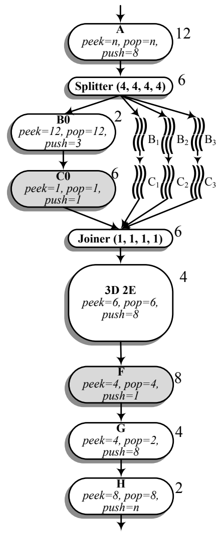
In the single-actor vectorization, the input and output tapes of a vectorized actor are left as scalar in two cases. First, the producer actor that fills the input tape of the vectorized actor is not SIMDizable. Second, the producer actor is vectorizable but its push rate is different from the pop rate of the consumer actor. For similar reasons, applying vectorization to the tape between the vectorized actors and its consumer is not possible in some cases. Therefore, the input and output tapes of a vectorized actor using single-actor SIMDization are not vectorized and remain as scalar. In order to read or write data elements in the correct order from the scalar input or output tapes in the vectorized actor, the pops/peeks for reading from the input tape and pushes for writing to the output tape must be done in a scalar fashion.

Lines 1-4 of $D_V$ in Figure 3b show the scalar tape read accesses. After single actor vectorization, the three `peek()`s and one `pop()` in lines 1-4 are induced from one `pop()` in the original code, line 1 of $D$ in Figure 3a. The `peek()`s and `pop()`s are reading the scalar input tape for 4 (=$SW$) consecutive executions of the original actor and packing those four read elements into a vector by writing each element to a lane of a vector variable. The accesses to the $i^{th}$ lane of a vector variable are indicated by `_v.{i}`. After a vector is formed from the scalar input tape in this way, the vector will be used for the computation in the rest of the actor's code. When the actor wants to write data to the output tape, it unpacks the data to scalar variables and pushes them to the scalar output tape (lines 8-11 of $D_V$ in Figure 3b). In other words, after each read and before each write to tapes, a SIMDized actor should perform packing and unpacking operations.

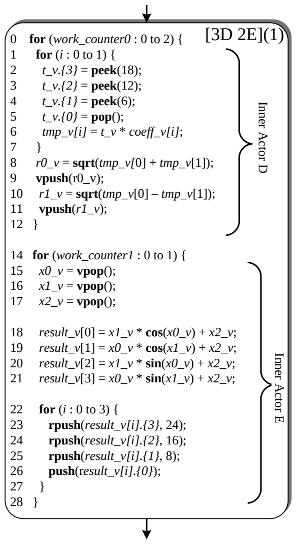
Since the tapes are left as scalar and each tape read is replaced by $SW$ tape reads after single-actor SIMDization, it is necessary to perform strided reads to receive the right data element for each of the $SW$ pops. The stride for each set of $SW$ reads in a SIMDized actor is equal to the pop rate in the original actor. For example in Figure 3a, since the pop rate of actor $D$ is 2, the `pop()` in line 1 is converted into 4 stride-two input tape reads as shown in lines 1-4 of Figure 3b. To read the scalar input tape in a non-destructive way, `peek()` is used instead of `pop()` for the first 3 reads, and the `pop()` is used only for the last read which also adjusts the read pointer of the input tape. For the same set of reasons, the scalar output tape is written with a stride equal to the push rate of the original actor. In Figure 3b, lines 8-11 unpack vector variable `r0_v` and write each element to the scalar output tape with a stride of 2, since the push rate of the original actor, $D$, is 2. The first 3 writes are done using *random access push* operations that do not move the write pointer of the tape (lines 8-10 and 13-15). Random access push operation are indicated by `rpush(data, offset)` in the code. The first argument of `rpush()` is the data to write and the second argument is the offset from the write pointer of the output tape to which the data will be written. The last write of each set of writes is performed using a normal push operation which updates the write pointer of the tape.

In Figure 3, only the code for the *work* functions of $D$ and $E$ is shown and the *init* functions are omitted. Actual vectorization of an actor's *work* and *init* method comprises of two parts: identifying variables and constants to be vectorized in an actor and rewriting the actor by replacing the vectorized variables with vector accesses and fixing the tape accesses. Identifying variables and constants to be vectorized can exploit the fact that the tape reads are the source of data for the variables used in the computation assignments inside an actor. A variable definition (i.e. *def*) originating for a pop/peek is marked to be vectorized. For other assignment statements, the *def* is identified as vector if its right hand side contains all variable *use*s marked as a vector. Also, a variable *use* that is used with other vector variable *use*s on the right hand side of a statement is marked as a vector. Similarly, constants used with other vector variable *use*s are marked to be vectorized as well. For example, in line 2 of actor $D$ in Figure 3a, `tmp[]` is identified as a vector because the right hand side variable, `t`, is written to by `pop` in line 1. After that, `coeff[]` is also detected as vector because of `t` on the right hand side. After identifying the variables, the statements are rewritten using the vector constructs. Also, the tape accesses are replaced with strided accesses at this point.

Single-actor SIMDization is not applicable to all the actors in a stream graph. Actors with mutable state (i.e. stateful) are excluded from single-actor SIMDization because it is not possible to run multiple executions of them in parallel. Splitters and joiners at this point are also excluded since they consist of only tape access operations without any substantial computation. Actors with function calls that are not supported by the SIMD engine are not SIMDized either. Input-tape-dependent control flow (i.e. *if* statements with pop-dependent conditions) or memory accesses (i.e. pop-dependent array subscripts) can also prevent MacroSS from performing single-actor SIMDization. The way MacroSS handles the input-tape-dependent control-flow structures or memory accesses is by switching to scalar mode (unpacking) before the input-tape-dependent structure and switching back to vector mode after the pop-dependent structure is finished (packing). MacroSS uses an internal cost model to decide if SIMDizing an actor with input-tape-dependent *if* or *for-loop* structures is beneficial or not.



**(a)**


[3D 2E](1)
Inner Actor D
Inner Actor E

```
0   for (work_counter0 : 0 to 2) {
1     for (i : 0 to 1) {
2       t_v.{3} = peek(18);
3       t_v.{2} = peek(12);
4       t_v.{1} = peek(6);
5       t_v.{0} = pop();
6       tmp_v[i] = t_v * coeff_v[i];
7     }
8     r0_v = sqrt(tmp_v[0] + tmp_v[1]);
9     vpush(r0_v);
10    r1_v = sqrt(tmp_v[0] – tmp_v[1]);
11    vpush(r1_v);
12  }

14  for (work_counter1 : 0 to 1) {
15    x0_v = vpop();
16    x1_v = vpop();
17    x2_v = vpop();

18    result_v[0] = x1_v * cos(x0_v) + x2_v;
19    result_v[1] = x0_v * cos(x1_v) + x2_v;
20    result_v[2] = x1_v * sin(x0_v) + x2_v;
21    result_v[3] = x0_v * sin(x1_v) + x2_v;

22    for (i : 0 to 3) {
23      rpush(result_v[i].{3}, 24);
24      rpush(result_v[i].{2}, 16);
25      rpush(result_v[i].{1}, 8);
26      push(result_v[i].{0});
27    }
28  }
```

**(b)**

**Figure 4:** *Part (a) of this figure shows the stream graph in Figure 2a after vertical fusion of $D$ and $E$. Part (b) illustrates the vectorized code for the fused actor, $3D\_2E$.*

## 3.2 Vertical SIMDization

Each actor vectorized by single-actor SIMDization performs packing and unpacking at points where tape reads or writes are performed for communicating with producer and consumer actors. The overhead introduced by the packing and unpacking operations can negatively affect the performance gains, even resulting in slowdowns in some cases. Vertical SIMDization is introduced in MacroSS to overcome this problem by merging vertically aligned vectorizable actors and reducing the number of packing and unpacking operations. In vertical SIMDization, pipelines of vectorizable actors are detected and transformed into a single actor. As long as the original actors in a pipeline are vectorizable, and no actor performs peek operations except the first and the last actor in the pipeline, the resulting coarse actor is guaranteed to be SIMDizable since the transformation does not introduce state or any other construct that may prevent SIMDization. The original actors, which are encapsulated in the new coarse node, are called *inner actors*. Figure 4 shows the stream graph after applying vertical fusion to nodes $D$ and $E$ and the resulting coarse actor $3D\_2E$.

After vertical fusion, MacroSS adjusts the repetition numbers of all actors to guarantee that they are all the smallest possible multiples of SIMD width, $SW$. This adjustment is done in two steps. First, the repetition numbers of inner actors and the coarse actor are changed. The repetition number of each inner actor will be its original repetition number multiplied by $\frac{M'}{SW}$. $M'$ is found by plugging the repetition numbers of the inner actors into Equation (1). The repetition number of the coarse actors is set to $\frac{SW}{M'}$. This guarantees that the repetition number of the coarse actor is set to the largest possible multiple or divisor of $SW$. After doing this step for each vertically fused SIMDizable actor, MacroSS applies Equation (1) to the entire graph to ensure that repetition number of all SIMDizable actors, including the coarse actor, is multiple of $SW$. In general, applying this method guarantees that the repetition vector of the graph is scaled by the smallest possible number. Using this method, the inner actors for $D$ and $E$ in $3D\_2E$ have repetition numbers of 3 and 2, while the new node $3D\_2E$ has a repetition number of 4. The pop rate of $3D\_2E$ is set to 6, which equals the original pop rate of the first inner actor ($D$) multiplied by the repetition number of that inner actor. Similarly, the push rate of $3D\_2E$ is set to 8. Note that the total number of times that $D$ and $E$ run after the fusion is exactly equal to the number of times before applying fusion.

The graph resulting after vertical fusion will have coarser nodes. The communication between the inner actors of a coarse actor is done through internal buffers (i.e. arrays) instead of global tapes. Transferring data between the inner nodes can be completely done using vectors since packing and unpacking are needed only during tape reads (pops) and writes (pushes) of the new coarse node at the boundaries. The main reason behind this is due to the change in the relative execution order of $D$ and $E$. This will be illustrated shortly using an example. At this point, single-actor SIMDization can be applied to the vertically fused actor. The code in Figure 4b shows how the SIMDization is applied to the new actor. Since actor $3D\_2E$ has 6 pops and 8 pushes, the strides for accessing input and output tapes of $3D\_2E$ are set to 6 and 8. These reads and writes from input and to output tapes are performed, as described in Section 3.1, using `peek`, `pop` and `rpush` operations at the beginning and end of $3D\_2E$ (lines 2-5 and 23-26).

The reads and writes between inner actors are handled differently. The previous scalar tape writes of $D$ in lines 8-11 and 13-16 of $D_V$ in Figure 3b are now written using vector writes as shown in line 9 and 11 of Figure 4b. Vector variable `r0_v` is written to the internal vector buffer between *inner* $D$ and *inner* $E$ using `vpush(r0_v)`. Also, the scalar tape reads of $E$ in lines 0-11 of $E_v$ of Figure 3b are replaced with reads from the internal vector buffer as in lines 15-17 in $3D\_2E$. Compared to the code generated after SIMDizing $D$ and $E$ separately, the vertical SIMDization technique in MacroSS eliminates 24 unpacking ([$D$'s repetition number] * [ $D$' push rate] * [SIMD width]) and 24 packing ([$E$'s repetition number] * [ $E$'s pop rate] * [SIMD width]) operations.

Figure 5 shows the details of how vertical SIMDization changes the execution of a stream graph and eliminates the packing/unpacking operations between the fused inner nodes. Part (a) of this figure shows how actors $D$ and $E$ interact with each other in scalar mode. Since $D$ has a push rate of 2 and $E$ has a pop rate of 3, 12 invocations of actor $D$ feeds 8 invocations of actor $E$ ($D_i$ and $E_i$ denote $i^{th}$ executions of $D$ and $E$, respectively). In other words, every 3 consecutive executions of $D$ produce enough data for $E$ to consecutively execute 2 times. The 24 elements produced by $D$ are written to the tape in order and read by $E$ in the same order. After performing single-actor SIMDization, every 4 consecutive invocations of $D$ is merged in actor $D_V$. The first execution of this new actor is similar to executing $D0$, $D1$, $D2$, and $D3$ in parallel as shown in Figure 5b. Since every 3 consecutive $D$s feeds 2 $E$s, MacroSS needs to convert the vectors to scalars before each set of scalar strided writes to the output tape of $D$ and then form vectors after each set of scalar strided reads in $E$ to guarantee that $E$ is receiving its data elements in the correct order. Parts (c) and (d) of Figure 5 show the order that the pushes in $D_V$ write and pops in $E_V$ read the data elements. If the pushes in $D$ were replaced by a vector push, then elements 0, 2, 4, and 6 would be written to the first row in memory. In that case $E$ will receive its input in the wrong order.

Vertical SIMDization applied to $D$ and $E$ replaces these 2 actors with actor $3D\_2E$. After vectorizing this new actor, every 4 consecutive executions of $3D\_2E$ will be merged together as

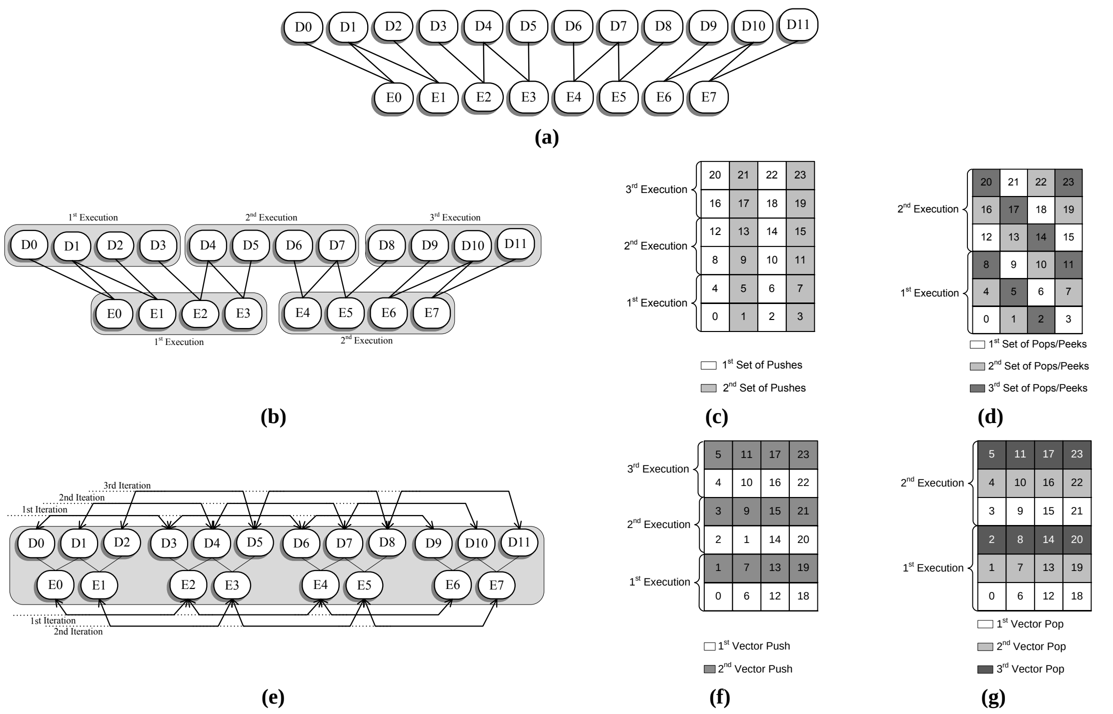


**Figure 5:** *Part (a) shows scalar execution of actors $D$ and $E$. Part (b) shows the execution of $D$ and $E$ after performing single-actor SIMDization. Part (c) illustrates the order that data elements are written to the tape in the main memory from $D$. The elements with the same colors are written in one set of push operations. Part (d) is similar to (c) but for the reads in actor $E$. Part (e) shows how vertical SIMDization changes the execution order of actors $D$ and $E$. Parts (f) and (g) illustrate the order that the elements are written to and read from the internal buffer between the inner actors $D$ and $E$.*

shown in Figure 5e. Since each invocation of this actor executes three $D$s first (for-loop in line 0 of Figure 4b) and then two $E$s (for-loop in line 14 of Figure 4b), running 4 of them in parallel will result in first running $\{D0, D3, D6, D9\}$, $\{D1, D4, D7, D10\}$, $\{D2, D5, D8, D11\}$ and then $\{E0, E2, E4, E6\}$ and $\{E1, E3, E5, E7\}$. Therefore, because the $D$s are generating their outputs in the same order as the $E$s need them, the scalar tape between original $D$ and $E$ can be changed to vector buffers and extra packing/unpacking operations can be deleted. Figures 5f and 5g show how the reads and writes are done between internal $D$s and $E$s. As shown, the vertical SIMDization has eliminated the need to perform packing and unpacking between $D$ and $E$. In summary, vertical fusion of vectorizable actors into a new coarse actor always results in less packing and unpacking operations because of the execution reordering of the inner actors.

## 3.3 Horizontal SIMDization

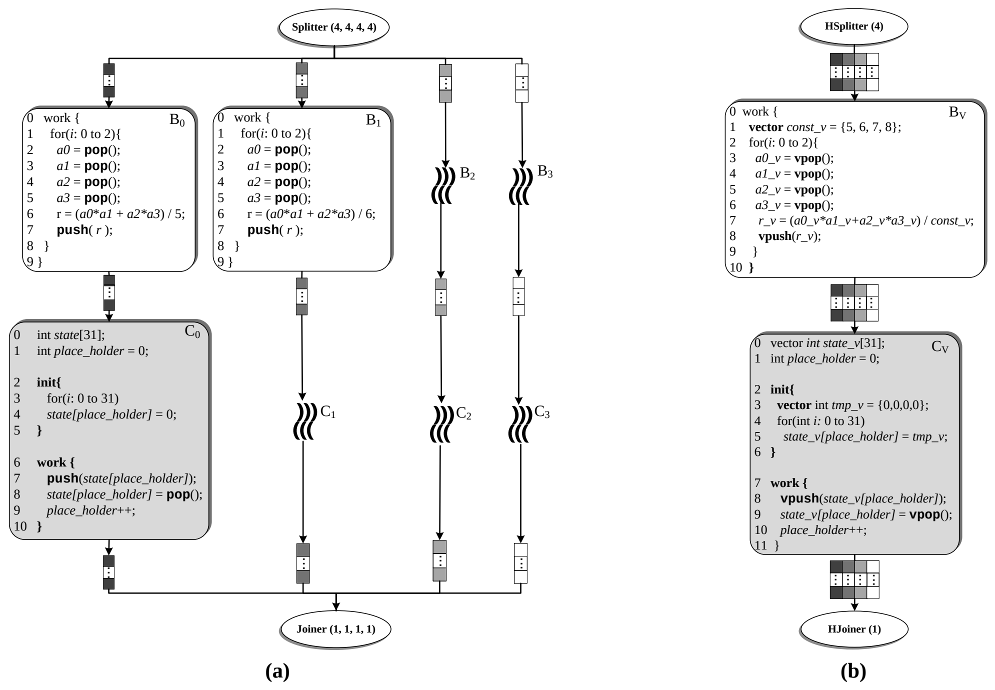
As mentioned earlier, only actors without mutable state can be SIMDized over using single-actor and vertical SIMDization. Since an invocation of a stateful actor depends on the previous invocation of the actor, different invocations cannot be parallelized. Due to the same reason, the existence of a stateful actor within a pipeline of actors or an actor whose peek rate is greater than pop rate prevents MacroSS from performing vertical SIMDization because the actor resulting after vertical fusion will be a stateful actor.

Horizontal SIMDization is an alternative approach taken by MacroSS to vectorize a set of task-parallel isomorphic actors when vertical and single-actor SIMDization are not applicable or result in inefficient SIMD code. First, Horizontal SIMDization finds task-parallel isomorphic actors by investigating each *split-join* (i.e. a subgraph containing a splitter and a joiner and all task-parallel actors between them). After finding the candidates, MacroSS horizontally SIMDizes $SW$ (SIMD Width) isomorphic actors by, conceptually, executing them together side by side. Input (output) tapes of $SW$ actors in a SIMDized set are also SIMDized, making each scalar tape a lane of a $SW$-wide SIMDized tape. Each actor in a SIMDized set still works on its own tape by accessing each lane of the SIMDized tape. Horizontal SIMDization is able to vectorize stateful actors as well as stateless actors because the state variables are kept in different vector lanes and updated separately similar to the non-vectorized case. The repetition number of the actors involved in this kind of SIMDization, unlike vertical and single-actor SIMDization, is not changed and can be numbers that are not multiples of $SW$.

Horizontal SIMDization mainly targets task-parallel *isomorphic* actors in *split-joins*. Two actors are called isomorphic if they have identical *work* and *init* functions with similar or different constant literals. A set of $SW$ isomorphic actors can be horizontally SIMDized as long as the following conditions are true: (1) all of them have the same repetition numbers, (2) all of them have the same *push* and *pop* rates, and (3) all of them are at the same level in a set of pipelines that are children of a *split-join*. Actors $B_0$ to $B_3$ and also $C_0$ to $C_3$ are considered isomorphic in Figure 2a.

Figure 6a shows a *split-join* subgraph of the stream graph in Figure 2a in more detail. Waves are used for depicting isomorphic actors due to the lack of space. Shaded actors $C_0$ to $C_3$ are stateful and can not be vectorized using any of the previously mentioned techniques. Although actors $B_0$ to $B_3$ are stateless, fusing each of them with the $C_i$ right after them prevents MacroSS from performing vertical SIMDization on the fused actor. Horizontal SIMDization can overcome this problem by forming one SIMDized actor out of actors $B_0$ to $B_3$ and another SIMDized actor out of actors $C_0$ to $C_3$ as shown in Figure 6b. Note that although the constants in line 6 of $B_i$s are different in each actor, the $B_i$s are still considered isomorphic because the constants can be vectorized together as shown in line 1 of actor $B_V$ in Figure 6b.

Before horizontal vectorization, each pipeline of $B_i$ and $C_i$ actors works on a separate set of scalar tapes highlighted by different shades in Figure 6a. Horizontal vectorization SIMDizes this set of four scalar tapes into one vector tape (See Figure 6b). `vpop()` in



```
0  work {
1    for(i: 0 to 2){
2      a0 = pop();
3      a1 = pop();
4      a2 = pop();
5      a3 = pop();
6      r = (a0*a1 + a2*a3) / 5;
7      push( r );
8    }
9  }
```

```
0  work {
1    for(i: 0 to 2){
2      a0 = pop();
3      a1 = pop();
4      a2 = pop();
5      a3 = pop();
6      r = (a0*a1 + a2*a3) / 6;
7      push( r );
8    }
9  }
```

```
0   int state[31];
1   int place_holder = 0;

2   init{
3     for(i: 0 to 31)
4     state[place_holder] = 0;
5   }

6   work {
7     push(state[place_holder]);
8     state[place_holder] = pop();
9     place_holder++;
10  }
```

```
0  work {
1    vector const_v = {5, 6, 7, 8};
2    for(i: 0 to 2){
3      a0_v = vpop();
4      a1_v = vpop();
5      a2_v = vpop();
6      a3_v = vpop();
7       r_v = (a0_v*a1_v+a2_v*a3_v) / const_v;
8       vpush(r_v);
9     }
10 }
```

```
0   vector int state_v[31];
1   int place_holder = 0;

2   init{
3     vector int tmp_v = {0,0,0,0};
4     for(int i: 0 to 31)
5       state_v[place_holder] = tmp_v;
6   }

7   work {
8     vpush(state_v[place_holder]);
9     state_v[place_holder] = vpop();
10    place_holder++;
11  }
```

**Figure 6:** *Part (a) and (b) show the graph before and after horizontal SIMDization, respectively.*

line 3 of $B_V$ reads 4 data items at once from the vectorized input tape. The lanes of this vector tape correspond to $B_0$, $B_1$, $B_2$ and $B_3$ respectively. Similarly, `vpush()` in line 8 pushes 4 data items at once to the vectorized output tape. Since tapes are also vectorized, no non-unit strided accesse using `peek()` or `rpush()` is needed. Horizontally vectorizing tapes can greatly improve the final performance by replacing the scalar tape accesses with vector accesses and, therefore, better utilizing the memory bandwidth. Actors $B_0$ to $B_3$, originally had 96 pops (= [pop rates: 12] $\times$ [repetition numbers: 2] $\times$ [SIMD with: 4]) which is reduced to 24 vector pops (= [vector pop rates: 12] $\times$ [repetition number: 2] ) after SIMDization. Similarly, the number of pushes in $B_i$s decreases to 6 vector pushes from 24 pushes, and $C_i$'s 24 pops (pushes) drops to 6 vector pops (pushes). In general, the number of tape accesses in the actors between a horizontally vectorized *split-join* structure is always reduced by factor of $SW$.

During horizontal SIMDization, MacroSS replaces the original splitter and joiner with *horizontal splitter (HSplitter)* and *horizontal joiner (HJoiner)*. In a horizontally vectorized structure, transitions between a scalar tape and vector tape occurs within the HSplitter and HJoiner. The HSplitter reads from a scalar tape and performs packing operations and writes them to its vectorized output tape. The HJoiner reads vector data types from its input and converts them to scalar before writing them to its scalar output tape. For example, in Figure 6, before SIMDization, the splitter executes 6 times and, during each execution, it conducts 16 pops from its scalar input tape and distributes the popped values between its scalar output tapes in a round-robin fashion using scalar push operations. After horizontal vectorization, the new HSplitter still executes 6 times and it performs 16 pops from its scalar input tape each time it executes. It forms 4 vectors out of the 16 data elements using packing operations and finally does a vector push to its vector output tape. The HJoiner is formed in a similar way, but instead of packing, it performs unpacking on the vector data it reads from its input tape.

Horizontal vectorization of an actor's *work* and *init* method comprises of two parts similar to single-actor SIMDization: identifying the vectors and rewriting the code for the actor. First, MacroSS needs to identify variables and constants for vectorization. The destination of pop and peek operations are marked as vector variables. Also, if the value of a constant in an actor is different from that of a matching constant in another isomorphic actor, the constant should be raised to a vector constant that contains the values of a matching constant of $SW$ actors. The vector variable `const_v` in line 1 of $B_V$ in Figure 6b is created from 4 different constants in $B_0$ to $B_3$. The identified vector variables and constants are used as the seeds for marking the other vector variables similar to single-actor SIMDization. After marking is done, MacroSS rewrites the horizontally SIMDizable actors using the marked vectors and changes their input and output tapes to vector tapes. Finally, the splitter and joiner in the horizontally SIMDizable *split-join* are replaced with horizontal splitter and joiner actors.

In summary, horizontal SIMDization is different from vertical and single-actor SIMDization in several ways. First, horizontal SIMDization can be applied only to isomorphic actors. Second, unlike other techniques used by MacroSS, it can handle stateful actors. Third, horizontal SIMDization does not affect the latency of the graph because there is no need to scale the repetition numbers of the actors. Finally, using horizontal vectorization, MacroSS can transform the existing task-level parallelism among the isomorphic actors to data-level parallelism.

## 3.4 Architecture Support for Tape SIMDization

In both single-actor and vertical SIMDization techniques, tape accesses are left as scalar. Converting these accesses to SIMD accesses results in reading or writing the data elements in an order which is different from the scalar execution. Vertical SIMDization reduces this overhead by replacing the scalar tape accesses between a pipeline of SIMDizable actors that are fuse-able with vector accesses to an internal buffer. In this section, two techniques that MacroSS uses to optimize the scalar tape accesses are discussed. The first technique uses a permutation based approach to target the overhead of performing packing/unpacking whenever data is communicated between scalar and vector parts of the stream graph. The second technique shows how MacroSS can simplify the read and write accesses of data that moves between scalar and vector actors in the presence of a unit called the streaming address generation unit (SAGU).

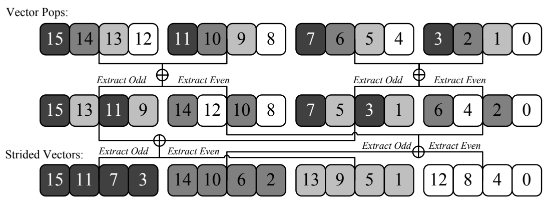


**Figure 7:** *This graph shows how 16 stride-4 tape reads in an actor are replaced with 4 vector pops and 8 permutation instructions*

**Permutation-based Tape Accesses:** The packing/unpacking overhead exists between scalar and vector actors, such as $F$ and $G$, in the SIMDized graph in Figure 2b. MacroSS optimizes these data conversions for actors whose push or pop counts are powers of 2 using two general architecture independent permutation operation:*extract_even(V1, V2, R), extract_odd(V1, V2, R)*. The *extract_even* ( *extract_odd*) operation takes two input vectors, $V1$ and $V2$, and constructs a third vector, $R$, using even (odd) positions of the inputs. This kind of permutation is supported by almost all SIMD standards (SSE, Altivec, Cell SPU, Neon).

Assume an actor($A$) has $X_r$ pop accesses without any peeks. Each pop access is a load operation followed by an add to adjust the position of the read pointer. After single-actor vectorization on $A$, the stride for scalar pop accesses will be $X_r$. For example, actor $D$ in Figure 2a originally had $X_r = 2$ pops and after SIMDization the stride is 2 as well. This stride guarantees that each set of scalar pops reads the right elements from the input tape. If a load instruction takes $C_r$ cycles, ignoring the add operations, popping the elements from the input tape in actor $A_v$, actor $A$ after SIMDization, takes $C_r \times X_r \times SW$ cycles. The other way that MacroSS can perform the same pop operations is to do $X_r$ vector loads, and then perform a set of permutations to form vectors identical to the case that the pops were in strided scalar format. MacroSS finds the minimum number of *extract_odd* and *extract_even* operations to shuffle the elements in the vectors after the vector pops. An example of this is shown in Figure 7. Assume that MacroSS is trying to SIMDize an actor with 4 pop operations. Instead of performing 16 strided pop/peek operations, MacroSS can generate 4 vector pops and then use 8 permutation operations (4 *extract_even* and 4 *extract_odd*) to form the strided pattern. This reduces the 16 scalar load operations to 4 vector load operations and 8 permutations. We ignore the savings due to removal of address generation operations.

In general, shuffling the elements of $X_r$ vectors to get to the same number of vectors each with elements strided at distance of $X_r$ from the original vector needs $X_r lg_2 X_r$ *extract_odd* and *extract_even* operations [22]. The same formula can be used to find the number of permutations that are needed to replace scalar push or peek operations with their vector equivalent. MacroSS compares the overhead of performing scalar tape accesses and vector tape accesses to identify the cheaper solution. After finding the cheaper solution, MacroSS transforms the tape accesses. The best solution can be different based on the SIMD width, tape access strides, permutation cost, and also read/write access latencies.

**Streaming Address Generation Unit:** Exploiting permutation-based tape accesses becomes harder when the push and pop rates are not powers of two or the underlying architecture does not support the needed permutation instructions. In these scenarios, replacing the strided scalar push or pop operations with vector versions in a vectorized actor forces subsequent scalar consumer or producer actors to perform complex address calculations to access the tape in the correct order. Although replacing the scalar accesses with vector accesses reduces the number of memory accesses and address generation operations in the vector actor, the overhead introduced due to additional address calculation operation in the direct consumer or producer is non-trivial. The code in Figure 8 shows how the address calculation should be performed in scalar actors that are connected to vectorized actors in which all the pushes are replaced with vector pushes. The *PushCnt* is set to the push rate of the vectorized actor. The overhead introduced by this code on the Intel Core i7 is at best 6 cycles on top of the memory access overhead assuming multiple back-to-back pop operations.

```
if (PushCnt - (BaseCntr-1) == 0 ) {
   BaseCntr = 0;

   if (StrideCntr - (SIMD_SIZE-1) == 0) {
      StrideCntr = 0;
      OffsetAddr = OffsetAddr + (PushCnt << LOG2_SIMD));
   } else { StrideCntr++; }

} else { BaseCntr++; }

OffsetValue = BaseCntr << LOG2_SIMD;
OffsetValue += StrideCntr;
OffsetValue += OffsetAddr;
ResultAddr = OffsetValue + BaseAddr;
```

**Figure 8:** *This code shows the address calculation in a scalar actor which is the consumer of a vectorized actor with vector pushes.*

To deal with this problem, we developed the Streaming Address Generation Unit (SAGU). The SAGU is able to reduce the overhead cost of address calculation in a scalar actor that is connected to a vectorized actor, in which all the scalar strided tape accesses are replaced with vector version, through a special functional unit that loads configuration data (push or pop count) and holds internal state allowing for quick generation of the required addresses. Figure 9 shows the hardware of the SAGU. Conceptually, when vector pushes (pops) occur the writes (reads) are row based but the reads (writes) have to access tape in a column-wise order to access the data elements in correct order. The *Stride_Counter* points to the column that needs to be accessed. The *Base_Counter* register points to the row location in the current column that contains the data element needed by the actor. The *Offset_Address* register offsets the *Base_Address* to the next set of vector data elements. Each scalar pop increments the *Base_Counter*. After the number of pops equals to the *Push_Count*, the *Stride_Counter* increments in order to access the next column and the *Base_Counter* is reset. When the *Stride_Counter* equals the *SW*, the *Stride_Counter* resets and the *Offset_Address* increments. The same operation occurs when scalar pushes are used. When designing the SAGU, we found that the largest push/pop count for SIMD to scalar conversion across all the kernels was 16K. With a SIMD width of 4, this allows us to use only 16-bit calculations throughout the unit except when we add the results to the base address register to generate the effective address. Most of the operations occur in parallel making the critical path two 16-bit operations and the 64-bit base address calculation. When optimized, we find that this unit will not be on the critical path allowing the address calculation to take the same amount of time as other address calculation instructions.

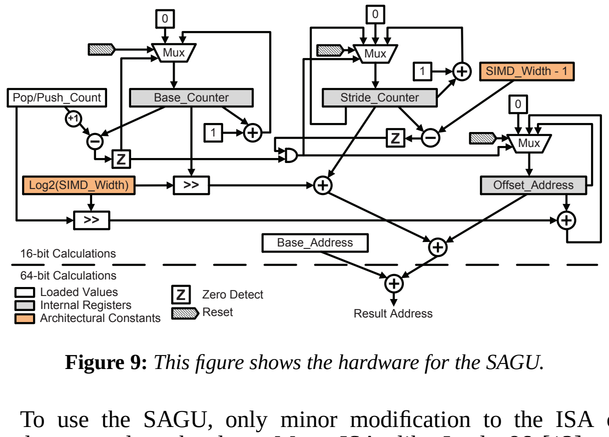


**Figure 9:** *This figure shows the hardware for the SAGU.*

To use the SAGU, only minor modification to the ISA or hardware needs to be done. Many ISAs like Intel x86 [13] and ARM [26] support multiple addressing modes which can perform operations on multiple address registers. There are available addressing mode configurations in these ISAs that we can modify to support the SAGU addressing mode. Effectively, this would be like performing a post-increment on an address register which would be transparent to the programmer and architecture. The alternative to this technique, if the ISA cannot support the addressing mode, would be to add another opcode to setup the SAGU and to increment it. Before starting each scalar actor, we would perform a SAGU setup and write the pop or push count. This would reset the internal counters to 0. After performing a pop or push operation, on the address register we would execute a SAGU increment to update the value to the next memory location. This would only require 2 additional instructions to the ISA and introduce 1 extra instruction for each memory operation in the program which would be far less than directly calculating the address. Because of the low cost[2] of the SAGU and the speed of the calculation, multiple units can be implemented if needed with little to no overhead.

### 3.5 Implementation

**Algorithm 1** Macro SIMDization Steps

```
Input: Stream Graph G, Architecture Description A
    {Apply prepass classic and streaming optimizations and also perform
    scheduling on the graph.}
 1: Prepass-Optimizations (G);
 2: Prepass-Scheduling (G);

    {Find the segments suitable for vertical/horizontal SIMDization.}
 3: ($G_V$, $G_H$) := Find-Vectorizable-Segments(G, A.CostModel);

    {Adjust the repetition numbers and perform vertical SIMDization on
    the specified segments.}
 4: Adjust-Repetition-Numbers (G);
 5: Vertically-SIMDize ($G_V$, A.CostModel);

    {Perform horizontal SIMDization after vertical is finished.}
 6: Horizontally-SIMDize ($G_H$, A.CostModel);

    {Apply Permutation-based optimizations and exploit SAGU.}
 7: Optimize-Tapes (G, A.CostModel);

    {Generate intermediate code for the specified target.}
 8: Emit-Intermediate-Code (G, A);
```

MacroSS's SIMDization algorithm can be divided into several distinct phases. In this section, a high-level overview of these steps are given. Algorithms 1 illustrates the overall ordering of the macro-SIMDization phases in MacroSS for vertical, horizontal SIMDization, and tape SIMDization. The remainder of this section explains each of the phases and their relationship to one another.

**Prepass Optimizations and Scheduling:** MacroSS applies a set of classic and streaming optimizations and also performs scheduling before starting the macro-SIMDization. The classic and streaming optimizations mainly improve the overall performance of the graph. The streaming optimization in some cases result in more efficient macro-SIMDization. For example, static parameter propagation, which propagates the values of the static read-only variables of an actor to all of its instances, helps detection of isomorphic actors. The steady state scheduling of the stream graph is also performed as a prepass.

**Identify Vectorizable Segments:** In this phase, MacroSS examines the stream graph and finds the segments of the graph that are suitable for vertical and horizontal SIMDization. For vertical SIMDization, MacroSS starts from a single vectorizable actor. This actor is added to an empty pipeline of vectorizable actors. Then MacroSS examines the consumer of that actor. If the consumer is also vectorizable and can be fused with the original actor without introducing state, it is added to the pipeline. This is repeated until the pipeline can not be extended anymore. At this point, all the actors in the pipeline are marked for vertical vectorization and added to $G_V$. Identifying horizontally vectorizable *split-join*s starts by testing the eligibility of a given *split-join* based on the definition given in Section 3.3. If a *split-join* passes the eligibility test it will be added o $G_H$.

One actor may be a member of both $G_V$ and $G_H$. Since MacroSS applies one form of SIMDization to any actor, it uses its cost model to choose what type of SIMDization (vertical or horizontal) is more effective for the actors that are in both $G_V$ and $G_H$. At the end, MacroSS guarantees that the intersection of the sets $G_V$ and $G_H$ is empty.

**Vertical SIMDization and Repetition Number Adjustment:** After finding the segments suitable for horizontal and vertical SIMDization, MacroSS adjusts the repetition numbers of the actors as described in Section 3.2. Then, the actual vertical vectorization is performed. This parts fuses the pipelines of vectorizable actors ($G_V$) found in the previous steps and changes them to vectorizable actors. Single-actor SIMDization is done as a special case of vertical SIMDization when a pipeline of vectorizable actor contains only one actor.

**Horizontal SIMDization:** After vertical SIMDization, the steady state repetition numbers are finalized. *Split-join*s eligible for horizontal SIMDization are passed to this phase and MacroSS changes the splitter and joiner actors to their horizontal versions. The statements in the task-parallel actors between the splitter and joiner are also merged to form vector instructions.

**Tape Optimization:** After vertical and horizontal vectorization, MacroSS searches for opportunities to perform tape optimization that are discussed in Section 3.4. This phase basically finds eligible set of reads or writes. Then, if it is cheaper, MacroSS replaces them with vector read or writes plus permutation instructions. If the target architecture is equipped with SAGU, MacroSS looks for cases where it can be exploited.

**Code Generation:** The final phase of macro-SIMDization deals with intermediate code generation. In this phase, MacroSS maps the internal stream representation to the target specific code (C++ in this case) and uses available architecture-dependent intrinsics to better utilize the target SIMD engines.

## 4. Comparison To Traditional SIMDization

Since MacroSS generates the intermediate code in a conventional imperative language, such as C or C++, traditional vectorization techniques can also be a viable approach to perform SIMDization on streaming applications. Traditional vectorization techniques mainly consist of inner-most loop, outer loop, and superword level parallelism extraction [1–3, 16, 23]. In this section, we try to compare MacroSS's graph-level SIMDization to traditional techniques and highlight the differences.

As streaming code gets converted to imperative intermediate code, it gets harder to extract the high-level information that is available at the graph-level. As a result, performing effective SIMDization becomes very difficult for some actors. Second, in some cases, traditional SIMDization is predicated on having complicated, carefully phase-ordered compiler analysis that needs the code in a certain templated form.

One of the points that makes MacroSS's SIMDization more powerful than any other vectorization technique on intermediate codes is the ability to identify isomorphic actors and perform horizontal SIMDization. At the graph level, MacroSS knows the relation between the actors and can detect the task-parallel isomorphic actors by doing a graph traversal. Performing the same task on the intermediate code is complicated. To find the isomorphic actors, the auto-vectorizer needs to extract the task graph and then compare the source code for the actors. Both of extracting the task graph and matching source code can be obfuscated by other optimizations.

The other issue that may disable auto-vectorization of the intermediate code is inability to adjust the schedule of the task graph. One of the main parts of the schedule is the repetition numbers.

[2] Area overhead is less than 1% of the area of the Core i7. This was measured by synthesizing the hardware model.

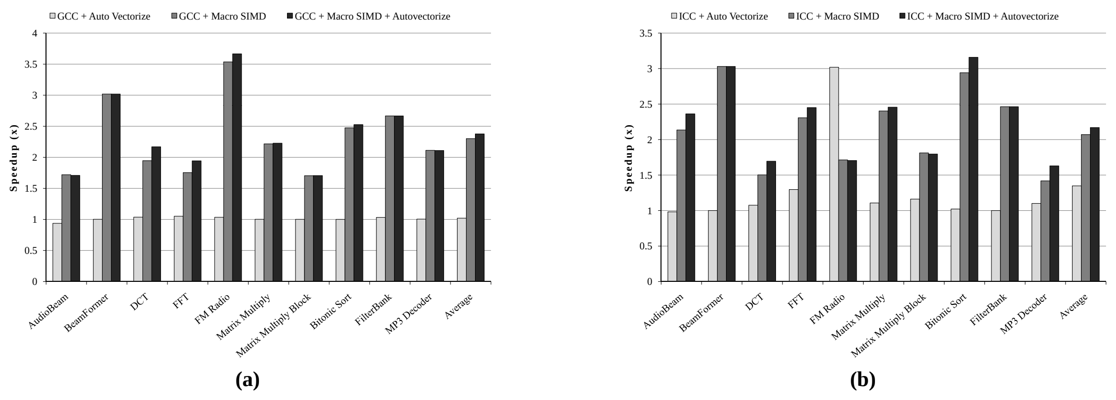


**Figure 10:** *In this graph the performance benefits of applying traditional auto-vectorization, macro-SIMDization, and both of them together are compared. Part (a) shows the speedups when GCC is used as the intermediate compiler. Applications in part (b) are compiled with Intel Compiler (ICC).*

MacroSS can intelligently scale the repetition numbers as needed by the SIMDization. Since the repetition numbers affect many parts of the generated code such as buffer (i.e. tape) allocation, and for-loop boundaries, they are not easily possible to adjust after generation of intermediate code.

Vertical SIMDization is another technique that MacroSS uses to perform vectorization. Even though performing vertical fusion on selected actors is in theory possible on intermediate code, it needs complex transformations and compiler analysis such as memory aliasing analysis, loop distribution, and loop relation analysis. MacroSS does not need these complex transformations and analyses since, at the graph-level, aliasing information and the relation between across is already embedded.

Although we are not proposing any universal partitioning approach that can handle both SIMDization and multi-core partitioning, performing vectorization on the high-level graph makes it possible for the partitioner and mapper parts of the streaming compiler to be able to make SIMD-aware decisions. This can lead to finding more efficient graph partitioning and mapping decisions. Since the intermediate code is already partitioned without considering possibility of SIMDization, it under-performs the macro-SIMDized code even after auto-vectorization.

In summary, MacroSS's SIMDization techniques are more efficient than auto-vectorization approaches because MacroSS has the ability to decide which actors are suitable for what kind of vectorization at the graph-level, transform the graph, adjust the schedule accordingly and generate permutation instructions based on actors read and write characteristics. Performing the same tasks during auto-vectorization after generation of intermediate code is difficult.

## 5. Methodology and Experiments

In this section, macro-SIMDization techniques in MacroSS are evaluated and compared against traditional techniques to perform auto-vectorization on languages. Also, the effectiveness of vertical and horizontal SIMDization is shown. The performance benefits of the streaming address generation unit is measured and presented in this section. Finally, the interaction between macro-SIMDization and multi-core scheduling is discussed.

**Methodology:** A set of benchmarks from the StreamIt benchmark suite [29] are used to evaluate MacroSS. The benchmarks are compiled and evaluated on a 3.26 GHz Intel Core i7 processor. The Intel Core i7 is used because it is equipped with the latest version of the SIMD engine from Intel, SSE 4.2.

MacroSS implementation is based on the StreamIt compiler. The macro-SIMDization steps are implemented as a separate compiler backend. The output of MacroSS is C++ code. To convert the generated C++ to x86 binary, GCC 4.3 [8] and Intel Compiler (ICC) 11.1 [14] are used. Both of these compilers are capable of performing aggressive optimizations and also auto-vectorization on C++ code. ICC is considered one of the best for its capabilities in performing inner-most, outer-most loop and superword-level parallelism vectorization. GCC also supports auto-vectorization for x86 processors and is widely used to compile C/C++ for Intel processors. In order to isolate the benefits of macro-SIMDization, all the experiments are performed using only one core of the processor except in the last experiment where we show performance benefits compared to multiple cores.

The original StreamIt backend in MacroSS is used to generate the baseline scalar intermediate C++ code. The baseline intermediate code is compiled to x86 binary using GCC or ICC with aggressive optimization flags enabled. The auto-vectorization pass in these compilers is used to perform traditional auto-vectorization on the generated C++ code. To macro-SIMDize streaming applications, the new backend in MacroSS is used to generate macro-SIMDized intermediate C++ code using target specific vector types and intrinsics. For measuring the performance of the generated binary the performance counters on the Intel Core i7 are exploited.

**Overall Performance:** The set of StreamIt benchmarks are compiled using macro-SIMDization and compared against ICC's and GCC's auto-vectorization. ICC and GCC are the leading auto-vectorizer compilers for Intel architectures capable of applying complex vectorization techniques proposed in the literature. Figure 10 illustrates how MacroSS's techniques perform compared to traditional auto-vectorization techniques. Figure 10a shows performance comparison between GCC's auto-vectorized, macro-SIMDized and auto-vectorized macro-SIMDized code. Figure 10b contains the same comparison for ICC. In both cases, macro-SIMDization achieves higher performance gains compared to auto-vectorization. On average, macro-SIMDization improves the final performance by an additional 54% and 26% compared to GCC and ICC auto-vectorizations. Applying both macro-SIMDization and auto-SIMDization can improve the performance by another 1.5% and 2.2% in benchmarks compiled using GCC and ICC. The only case that traditional auto-vectorization outperforms macro-SIMDization is *FMRadio* on ICC. In this special case, ICC performs inner-loop vectorization on the main for-loop in the code which results to aligned memory accesses but MacroSS's macro SIMDization results in unaligned memory accesses. It is possible to make MacroSS leave this for-loop for inner-loop vectorizer since, during macro-SIMDization, it knows inner loop vectorization will be more efficient in this special case. *BeamFormer* and *FilterBank* mainly consists of several pipelines of *split-join* structures with isomorphic task-parallel actors. It is not possible to collapse these pipelines into one pipeline because they have stateful actors. Therefore, the speedups in these two benchmarks are mainly due to horizontal vectorization. In summary, GCC shows unimpressive gains using auto-vectorization. Although, ICC shows fairly large gains (1.34x on average), MacroSS's techniques result in even larger gains (2.07x on average). Having access to global information enables MacroSS to achieve significant speedup.

**Effect of Vertical SIMDization:** Vertical SIMDization is one of the main techniques that MacroSS uses to perform vectorization on streaming graphs. Figure 11 illustrates, the effectiveness of this type of SIMDization. In this experiment, the baseline is a streaming graph macro-SIMDized with only single-actor SIMDization and compiled with GCC. As shown in the figure, vertical SIMDization, on average, improves the performance of the baseline by 40%. *Matrix Multiply Block* benefits the most because the vertical fusion of SIMDizable actors eliminates a large number of packing/unpacking operations. Without vertical fusion, macro-SIMDization in this benchmark would result in significantly less speedup then that shown in Figure 10a. The benefits in *FilterBank* and *BeamFormer* are very negligible because these benchmarks are vectorized mostly using horizontal vectorization. In *FMRadio* and *AudioBeam* the opportunity for performing vertical SIMDization is very small because most of the vectorizable actors in these benchmarks are isolated from each other and do not form a pipeline.

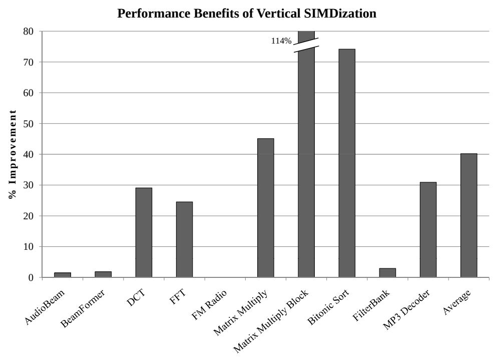


**Figure 11:** *This graph shows percent speedup due to vertical SIMDization compared to single-actor SIMDization.*

**Streaming Address Generation Unit:** MacroSS utilizes the SAGU to eliminate the packing/unpacking overhead and also improve memory bandwidth utilization when data is crossing scalar and vector boundaries in a stream graph. To evaluate the benefits of the SAGU, we use the performance counters on the Intel Core i7 to find the overheads introduced by packing and unpacking operations and also scalar memory accesses. Figure 12 illustrates the effect of utilizing SAGU. The baseline in this graph is macro-SIMDized code. On average, this unit can improve the final performance of the macro-SIMDized benchmarks by 8.1%. The performance of *Matrix Multiply* and *DCT* are improved 22% and 17% respectively because they perform a large number of packing/unpacking operations and scalar memory reads and writes. *BeamFormer* shows the least improvement because almost all the speedup in this benchmark is due to horizontal SIMDization. *MP3 Decoder* is also not affected by the SAGU because its computation to communication ratio is very high and the packing/unpacking operations do not cause a substantial performance overhead.

**Multicore and Macro-SIMDization:** Implementing a scheduler to decide how to partition a stream graph between multiple cores and also use the SIMD engines is a non-trivial task. Partitioning and mapping decisions taken by a naive multi-core scheduler may reduce the SIMD opportunities. In this section, we show conservatively estimated numbers on how a simple SIMD-aware multi-core scheduler/partitioner performs. The scheduler we use in this experiment first performs multi-core partitioning and then performs macro-SIMDization. This approach reduces the opportunities for performing vertical fusion and also horizontal SIMDization. If multi-core partitioning removes most of the benefits of the SIMDization and the scheduler has to choose between SIMDization and multi-core execution, it always chooses SIMDization because it reduces memory/cache traffic and communication overhead between the cores. Since the multi-core scheduler does not consider the possible benefits of vertical fusion and horizontal SIMDization in several benchmarks, the performance benefits of SIMDization

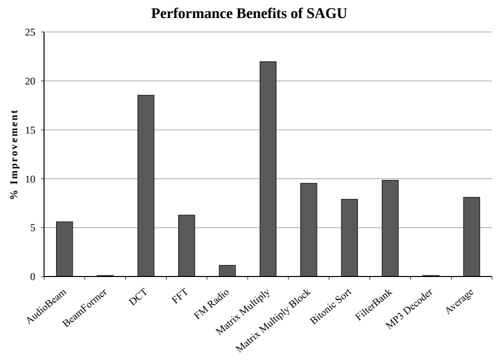


**Figure 12:** *This graph shows how SAGU can improve the performance of a macro-SIMDized graph.*

is reduced compared to Figure 10. Therefore, these numbers are conservative estimates of the performance of a SIMD-aware multi-core scheduler. As shown in Figure 13, the performance benefits of 4-core execution is within 5% of macro-SIMDized 2-core execution. Exploiting the SIMD engines increases the speedup from 1.28x to 2.03x in 2-core schedule and from 1.85x to 3.17x in 4-core schedule. For *Matrix Multiply* and *Matrix Multiply Block*, the scheduler prefers to only use the SIMD engines because multi-core partitioning, in this case, leads to high inter-core communication overhead.

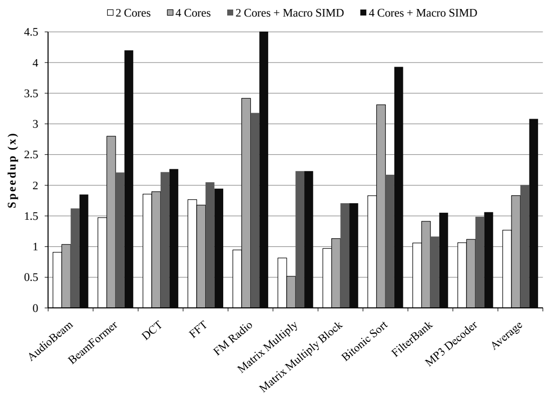


**Figure 13:** *The performance benefit of SIMDization in case a graph is scheduled for multi-core is shown in this graph.*

## 6. Related Work

There is a large body of literature that deals with exploiting parallelism in streaming languages for better performance [5, 6, 29]. The most relevant works include stream graph refinements to extract coarse-grain task-level, data-level and pipeline parallelism and map them onto multi-core architectures [9, 10]. Authors in [15] applied modulo scheduling to task graphs for maximizing pipeline parallelism also on multi-core architectures. Our work is distinctively different from and complementary to these previous works in its ability to exploit SIMD parallelism and generate SIMD enabled codes for various architectures. Vertical SIMDization focuses on fine-grain SIMD parallelism, while horizontal SIMDization transforms task-level parallelism to SIMD parallelism.

Auto-vectorization and SIMD code generation were studied extensively in the literature. The seminal work of Allen and Kennedy on the Parallel Fortran Converter [1, 2] set the grounds for most of the work on auto-vectorization that followed. For targeting a variety of SIMD architectures and solving severe problems that arise, specifically data alignments and permutations, a large number of studies has been conducted [7, 16, 21, 22, 25, 31]. All these techniques can be applied to the generated intermediate code of streaming applications. However, our work is unique in that vectorization is applied on a higher level of representation of the program, which enables us to utilize global information such as execution rates of actors and exposed data communications for generating better vectorized codes. In contrast to focusing on local structures like loop nests and basic blocks, our macro-SIMDization leverages the streaming applications' static characteristics, such as static schedules and pre-defined data access patterns.

There has been recent work [20] on generating efficient permutation instructions based on StreamIt, but for only one specific SIMD device (VIRAM). MacroSS provides efficient SIMDization for streaming applications which is flexible and portable enough to be applied to a variety of SIMD architectures.

Vectorizing computations that access non-unit stride data motivated the development of the SIMdD (Single Instructions on Multiple disjoint Data) model and SIMdD architectures, such as the IBM eLite DSP[18]. Such architectures better support non-consecutive data accesses via vector pointer hardware. Tuned for streaming applications in which non-unit strides are statically known and fixed for the entire execution of an actor, our architectural support, SAGU, is simpler and entails smaller overheads than what is available in general SIMdD architectures.

## 7. Conclusion

As SIMD-enabled multi-core systems become ubiquitous, it is critical for programming languages and compilers to be able to flexibly target both the SIMD and multi-core aspects of these architectures. Several retargetable streaming languages, such as StreamIt, have been proposed to exploit parallelism across the cores. These languages apply traditional auto-vectorization to the imperative intermediate code (e.g. C/C++) to target SIMD engines. In many cases, applying auto-vectorization to the generated intermediate code results in under-utilization of SIMD engines because much of the high-level information available in the streaming application, such as data-flow information and the set of valid schedules, is not used by the auto-vectorizer.

In this paper, we introduce macro-SIMDization: a technique for vectorizing stream graphs using the high-level information available in streaming programs. A new compilation system, MacroSS, is developed to show the benefits of macro-SIMDization compared to traditional SIMDization techniques. MacroSS utilizes three new techniques to achieve high utilization of the SIMD engines: single-actor, vertical, and horizontal SIMDization. Architectural support for tape optimizations, using general permutation operations and a streaming address generation unit (SAGU) is also discussed in the paper.

Our results show that MacroSS is capable of improving the performance of streaming applications by an average of 54% and 26% compared to auto-vectorizers in GCC and Intel compiler, respectively. In the experiments, we also evaluated how the SAGU can improve the performance on average by an additional 8.1% by eliminating packing/unpacking operations between scalar and vector actors. Finally, we show the performance benefits of macro-SIMDization in the presence of a naive multi-core scheduler for streaming applications. Even with a naive multi-core scheduler, we estimate that we can achieve better performance than a 4-core Intel Core i7 on only 2-cores using SIMD. The results indicate that performing macro-SIMDization can significantly improve the performance of streaming applications and extend their retargetabilitiy by making them more suitable for SIMD programming.

## Acknowledgement

Much gratitude goes to the anonymous referees who provided excellent feedback on this work. We also thank Chandra Krintz for shepherding this paper. This research was supported by ARM Limited, the National Science Foundation under grant CNS-0615261 and the Korea Research Foundation grant funded by the Korean Government (MOEHRD) (KRF-2007-356-D00200).

## References

[1] R. Allen and K. Kennedy. Pfc: A program to convert fortran to parallel form. Technical Report 82-6, Dept. of Math. Sciences., Rice University, Mar. 1982.

[2] R. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. *ACM TOPLAS*, 9(4):491–542, 1987.

[3] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures: A dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[4] ARM Ltd. *ARM Neon*, 2009. http://www.arm.com/miscPDFs/6629.pdf.

[5] I. Buck et al. Brook for GPUs: Stream computing on graphics hardware. *ACM Trans. Gr.*, 23(3):777–786, Aug. 2004.

[6] M. Chen, X. Li, R. Lian, J. Lin, L. Liu, T. Liu, and R. Ju. Shangri-la: Achieving high performance from compiled network applications while enabling ease of programming. In *Proc. '05 PLDI*, pages 224–236, June 2005.

[7] A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for simd architectures with alignment constraints. In *Proc. '04 PLDI*, pages 82–93, 2004.

[8] GNU Compiler Collection. Gcc 4.3.2, 2008. http://gcc.gnu.org/gcc-4.3/.

[9] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. Meli, A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *10th ASPLOS*, pages 291–303, Oct. 2002.

[10] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *12th ASPLOS*, pages 151–162, 2006.

[11] IBM. *Cell Broadband Engine Architecture*, Mar. 2006.

[12] Intel. Intel sse4, 2006. http://download.intel.com/technology/architecture/new-instructions-paper.pdf.

[13] Intel. Intel Core i7, 2008. http://www.intel.com/products/processor/corei7/index.htm.

[14] Intel. Intel compiler, 2009. software.intel.com/en-us/intel-compilers/.

[15] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proc. '08 PLDI*, pages 114–124, June 2008.

[16] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proc. '00 PLDI*, pages 145–156, June 2000.

[17] E. Lee and D. Messerschmitt. Synchronous data flow. *Proc. IEEE*, 75(9):1235–1245, 1987.

[18] J. H. Moreno, V. Zyuban, U. Shvadron, F. D. Neeser, J. H. Derby, M. S. Ware, K. Kailas, A. Zaks, A. Geva, S. Ben-David, S. W. Asaad, T. W. Fox, D. Littrell, M. Biberstein, D. Naishlos, and H. Hunter. An innovative low-power high-performance programmable signal processor for digital communications. *IBM Jrn. of Research and Development*, 47(2-3):299–326, 2003.

[19] A. Munshi. Opencl parallel computing on the gpu and cpu., 2008.

[20] M. Narayanan and K. A. Yelick. Generating permutation instructions from a high-level description. In *In Proc. MSP'04*, 2004.

[21] D. Nuzman and R. Henderson. Multi-platform auto-vectorization. In *Proc. 2006 CGO*, pages 281–294, 2006.

[22] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for simd. In *Proc. '06 PLDI*, pages 132–142, 2006.

[23] D. Nuzman and A. Zaks. Outer-loop vectorization - revisited for short simd architectures. pages 2–11, 2008.

[24] Nvidia. *CUDA Programming Guide*, June 2007. http://developer.download.nvidia.com/compute/cuda.

[25] G. Ren, P. Wu, and D. Padua. Optimizing data permutations for simd devices. In *Proc. '06 PLDI*, pages 118–131, 2006.

[26] D. Seal. *ARM Architecture Reference Manual*. Addison-Wesley, London, UK, 2000.

[27] L. Seiler et al. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Gr.*, 27(3):1–15, 2008.

[28] F. Semiconductor. Altivec, 2009. www.freescale.com/altivec.

[29] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. 02 CC*, pages 179–196, 2002.

[30] Tilera. Tile64 processor - product brief, 2008. http://www.tilera.com/pdf/.

[31] P. Wu, A. E. Eichenberger, and A. Wang. Efficient simd code generation for runtime alignment and length conversion. In *Proc. 2005 CGO*, pages 153–164, 2005.