Parallel Back-End for the Halide Image Processing Language

Lefteris Ioannidis, *SuperUROP Student, MIT CSAIL*
Shoaib Kamil, *Research Scientist, MIT CSAIL*

***Abstract*—As computers are using increasingly more complex memory hierarchies and programming languages are incorporating more complex parallel semantics, the problem of determining optimal data distributions for parallel programs is becoming more difficult. Finding the optimal execution schedule for every parallel code is an NP-Complete problem, but that is not the end of the road. The Halide high-performance language for computational photography, developed here at CSAIL, uses machine learning and auto-tuning techniques to find a schedule very close to optimal. In the Commit group of CSAIL we are working on a Parallel Back-End for Halide, that will lower explicit parallelism to the level of the optimizer and allow more complex parallel optimization passes to be used. Our back-end is an extension to the existing LLVM Intermediate Representation (IR) and consists of a set of metadata and annotations that explicitly describe parallel loops and vectorizable data structures to the optimizer. Then our Parallel Optimization Passes find the best arrangement of data inside the Halide pipeline and handle instruction parallelization, array tilling and vectorization. Our goal for this new interface is for it to be simple, minimalistic and backwards compatible with the existing IR, to eventually reach the LLVM upstream and to become the de facto parallel semantics for the LLVM compiler. Our work is important because it will enable more languages with parallel front-ends, to take advantage of common automatic performance optimization patterns.**

***Index Terms*—Halide, LLVM, Performance, Image, Back-End, Parallel, Intermediate, Representation, Languages, Compilers.**

I. INTRODUCTION

AFTER the end of Moore's law struck transistor manufacturers we witnessed the dawn of the multicore age. For the past twenty years many frontiers in the domain of parallel programming have been broken, allowing us to seemingly write parallel code without worrying about architecture specific details. Now with the rise of Cloud and Distributed Computing, programmers are called to use more complex parallel and distributed semantics to write scalable software for large computer clusters with thousands of machines. The complexity increases exponentially with every new layer of memory the programmer has to optimize around. It is the job of Programming Language Engineers to build languages, compilers and interpreters that take away some of the burden of manual Performance Optimization and do it automatically. The Halide [1] language is a Domain Specific Language (DSL) for High Performance Image Pipeline Processing. Image processing pipelines are everywhere, and are essential to capturing, analyzing, mining, and rendering the rivers of visual information gathered by our countless cameras and imaging-based sensors. Applications from raw processing, to object detection and recognition, to Microsofts Kinect, to Instagram and Photoshop, to medical imaging and neural scanning all demand extremely high performance to cope with the rapidly rising resolution and frame rate of image sensors and the increasing complexity of algorithms. There are many common use cases of complex pipelines, where manual optimization is nearly impossible, like the local Laplacian filter transform with more than 99 pipeline stages. The Halide language offers rich parallel semantics to describe grid operations on graphics, optimized for different CPU and GPU architectures. It also allows explicit runtime scheduling in terms of parallelization, tilling and vectorization and includes a back-end that compiles the pipeline for multi-core architectures.

Those optimizations happen during runtime. The Halide JIT compiler consumes work items and schedules them opportunistically. The majority of the parallel loop optimizations though could take place in compile time, which would allow better data manipulation and better performance, depending on the loop locality. For example Polly [2], the Polyhedral Loop Optimization Framework for LLVM, statically runs optimizers on nested loops and determines the optimal locality for the given arrays. In Halide we can achieve compile-time parallel loop optimization by propagating the parallel

L. Ioannidis is with the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 02139 USA e-mail: elefthei@mit.edu

Manuscript received April 3, 2015.

semantics down to the IR level. Because we want to take full advantage of optimization hardware offered by the CPU we have to propagate both parallelization and vectorization information from the Halide front-end, so the back-end can determine an optimal static schedule for these. Halide is based on the LLVM compiler suite and uses the LLVM IR internally, so most of the new Parallel constructs we use for Halide can be used by LLVM too. We chose to work with Halide on this because it offers rich Parallel Semantics on the front-end. When explicit parallel semantics are lacking, automatic optimization becomes harder. For example, the cilk [3] project uses strong sequential semantics and implicit parallelization. The parallel semantics on which my work was focused are nested parallel loops with loop-carried dependencies and vectorization potential. Designing a full Parallel IR which is compatible with both Sequential semantics and Parallel Semantics is a task which is beyond the scope of this paper but definitely a direction parallel IRs should be heading in the future. Instead we aim to begin this effort inside LLVM, the most popular compiler suite, by lowering to the IR level parallel and serial loops in the most descriptive way, in terms of both parallelization and vectorization.

II. Previous Work

A. OpenCL SPIR

A lot of research on IRs is motivated by graphics optimizations. One such example of a parallel IR designed for GPUs is OpenCL SPIR [6]. SPIR was also built by extending the LLVM IR to include a few metadata that describe OpenCL kernel functions. It was created when the vendors of OpenCL code decided they did not want to include the kernel source with their product, so instead they created SPIR. An intermediate language which translates OpenCL kernel code to LLVM IR instructions, thus combining the portability of LLVM and the ease of programming of OpenCL C. 1 has an example SPIR code snippet. As you can see, it looks very much like any LLVM IR with the addition of OpenCL metadata.

While using OpenCL kernels would be a very convenient way to describe Halide functions in the IR level, it would be very inconvenient to translate alternative parallel models, such as OpenMP and MPI, to OpenCL and SPIR respectively. We could potentially support a SPIR-like model for describing Halide pipeline stages, but that's not the bigger picture here. The bigger picture is that we require a Parallel IR which is compatible with many different languages and different semantics, not only OpenCL and Halide.

```
define spir_kernel void
        @sum( i32 %size,
              float addrspace(1)* %vec1,
              float addrspace(1)* %vec2
            ) nounwind {

<kernel LLVM IR code>

}

!opencl.kernels = !{!0}
!opencl.enable.FP_CONTRACT = !{}
!opencl.spir.version = !{!6}
!opencl.ocl.version = !{!7}
!opencl.used.extensions = !{!8}
!opencl.used.optional.core.features = !{!8}
!opencl.compiler.options = !{!8}

!0 = metadata !{void (i32, float addrspace(1)*,
                float addrspace(1)*)* @sum,
                metadata !1,
                metadata !2,
                metadata !3,
                metadata !4,
                metadata !5}
!1 = metadata !{metadata !"kernel_arg_addr_space",
                i32 0, i32 1, i32 1}
!2 = metadata !{metadata !"kernel_arg_access_qual",
                metadata !"none",
                metadata !"none",
                metadata !"none"}
!3 = metadata !{metadata !"kernel_arg_type",
                metadata !"int",
                metadata !"float*",
                metadata !"float*"}
!4 = metadata !{metadata !"kernel_arg_type_qual",
                metadata !"const",
                metadata !"",
                metadata !""}
!5 = metadata !{metadata !"kernel_arg_name",
                metadata !"size",
                metadata !"vec1",
                metadata !"vec2"}
```

Fig. 1. SPIR extends the LLVM IR to include OpenCL metadata. In this SPIR code snippet, the OpenCL kernel sum is defined and it's arguments are passed through nested LLVM IR metadata.

B. SPIRE

SPIRE stands for Sequential to Parallel Intermediate Representation, an IR which focuses on parallel runtimes with sequential semantics similarly to Intel Cilk [3] and Cray Chapel [7]. It is built on top of the PIPS IR, a much simpler representation than the one LLVM uses. It would be easy to port SPIRE into LLVM as it is very well defined. There is a set of annotations that appear across multiple parallel runtimes, such as `parallel_for`, `reduce`, `spawn`, `sync`, `lock` and more. SPIRE has it's own versions of these annotations in the IR level.

```
forall i in 1..n do                 forloop(i,1,n,1,
  t[i] = i;                                t[i] = i,
var (sumVal) = + reduce                    parallel);
  ([i in 1..n] f(i),                forloop(i,1,n,1,
   1..n);                             sumVal = sumVal+f(i),
                                      reduced)
```

Fig. 2. SPIRE version of forall and reduce, compared to the same example in Cray Chapel

Figure 2 shows an example of a parallel for loop with a reducer in Cray Chapel and its equivalent in the SPIRE language. As you can see, all information regarding the size of the loop, parallelization and variable management (reducer) are lowered in the IR.

While SPIRE is very close to what we are looking for, it suffers from the issue of being too general and relaxed with the assumptions it makes. For example, both `MPI_Send` and `WriteBuffer` of OpenCL are described with the same SPIRE primitive, `send`. However, the OpenCL `WriteBuffer` operation is synchronous and the MPI `MPI_Send` primitive is asynchronous. So then the equivalent SPIRE primitive, `send`, would have to compromise the synchronicity of OpenCL without any real reason to do so, or even worse, risk an ambiguous implementation. In our implementation of a Parallel IR for LLVM, we do not strive to cover every case of parallel front-ends, but we strive for correctness. The same code written in different parallel runtimes should compile to the same LLVM Parallel IR and it's runtime execution should be predictable and deterministic.

C. Stanford Legion

A good candidate for a data-driven parallel semantics front-end except Halide could be Stanford legion [5]. It describes parallelism during data declarations and marks memory regions as either read, concurrent read or write enabled. Legion runs it's own runtime scheduler and is very effective in generating parallel and distributed machine code. The parallel code can be run using the Legion runtime, which schedules data-independent tasks to multiple CPUs, GPUs with OpenCL as well as multiple machines in a GasNET [9] distributed shared memory system. We turned to Legion as it offers rich parallel semantics and a good parallel and distributed scheduler, something we will have to address when distributed runtime capabilities get added to the Halide runtime.

D. PoCL

PoCL [4] stands for Portable CL, a project meant to compile OpenCL code for any non-OpenCL back-end supported by LLVM. What is interesting about this project is that it compiles OpenCL kernel functions, similar to Halide pipelines and it does that using parallel loop annotations in the LLVM IR level. In fact, the parallel loop annotations used first by PoCL have been a part of the LLVM project since version 3.6. We used some of the techniques used by PoCL to translate kernels into parallel loops and also used their Parallel IR extensions to annotate parallel loops coming from the Halide front-end. Their parallel loop annotations annotate both the returning branch of the loop as well as parallel loads and stores inside the loop body, a necessary action needed to eliminate loop-carried data dependencies.

III. Technical Approach

A. High level Design

Let's consider the problem from a high level for a bit without bothering with the implementation specific details of LLVM. Our goal is, given an explicitly declared parallel schedule coming from the programmer in the Halide language, to statically determine the best loop parallelization, parallel data distribution, loop ordering, unrolling and vectorization. The dependence relations between these different optimizations should be completely hierarchical as shown in figure 3, where the leaves of the tree are vector instructions. Of course data dependencies and loop-carried dependencies are an obstacle to creating this hierarchy so they must be resolved statically before a tree schedule such as the one in figure 3 can be generated. The way we will address the issue of loop-carried dependencies is by using the same dependency solving optimizer used inside Polly [2]. We can compile Polly as a dynamically linked library and link it against the LLVM optimizer, in order to include polyhedral dependency solving in LLVM.
Another issue with statically generating an optimization tree such as the one in figure 3 is conditionals. A conditional which depends on user supplied variables can break our static schedule and inhibit performance for two reasons. Not only due to the emptying of the whole pipeline due to an unexpected branch, but also due to our inability to vectorize across the boundaries of the conditional. Here, our solution is to use the *If converter* optimization, part of the auto-vectorization passes that are included by default in LLVM, in order to predict the branch and flatten the if statement so it doesn't inhibit our hierarchy. Now we will go into the implementation details, of how we can statically get a loop schedule from an arbitrary Halide schedule.

B. Parallel Loops

Our goal for this project is to build from the bottom up, a Parallel IR for LLVM that satisfies the Parallel

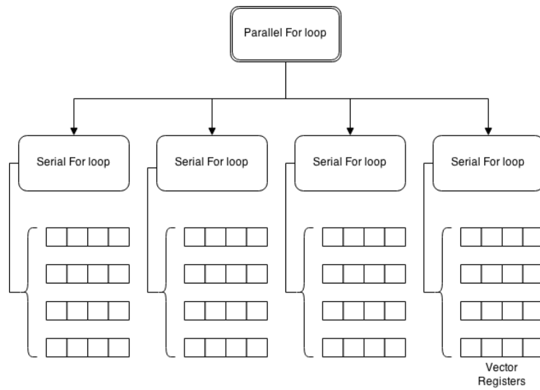


Fig. 3. Manually optimizing parallel loops for performance yields a structure like this. Given the right parallel semantics Halide can generate such a schedule automatically.

```
for.body:
...
%0 = load i32* %arrayidx, align 4,
        !llvm.mem.parallel_loop_access !0
...
store i32 %0, i32* %arrayidx4, align 4,
        !llvm.mem.parallel_loop_access !0
...
br i1 %exitcond,
        label %for.end,
        label %for.body,
        !llvm.loop !0

for.end:

!0 = metadata !{ metadata !0 }
```

Fig. 4. Simple parallel loop in the LLVM IR using parallel annotations. It uses both the `llvm.loop` and `llvm.mem.parallel_loop_access` metadata types that refer to the same loop identifier `!0`.

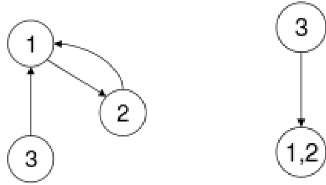
```
outer.for.body:
  ...
  %val1 = load i32, i32* %arrayidx3,
        !llvm.mem.parallel_loop_access !2
  ...
  br label %inner.for.body

inner.for.body:
  ...
  %val0 = load i32, i32* %arrayidx1,
        !llvm.mem.parallel_loop_access !0
  ...
  store i32 %val0, i32* %arrayidx2,
        !llvm.mem.parallel_loop_access !0
  ...
  br i1 %exitcond, label %inner.for.end,
        label %inner.for.body,
        !llvm.loop !1

inner.for.end:
  ...
  store i32 %val1, i32* %arrayidx4,
        !llvm.mem.parallel_loop_access !2
  ...
  br i1 %exitcond,
        label %outer.for.end,
        label %outer.for.body,
        !llvm.loop !2

outer.for.end:
...
!0 = !{!1, !2} ; a list of loop identifiers
!1 = !{!1} ; an identifier for the inner loop
!2 = !{!2} ; an identifier for the outer loop
```

Fig. 5. Nested parallel loops in the LLVM IR using parallel annotations. Notice how the number of metadata needed to describe loop-carried dependencies increases with the nesting level.

semantic requirements of Halide. Then run optimizations on this statically generated IR before we begin Halide specific runtime scheduling. So we started with parallel loops, since in LLVM version 3.6 parallel for loop annotations are included. In particular, LLVM uses two different metadata to denote loop parallelization, `llvm.loop` and `llvm.mem.parallel_loop_access`. Figure 4 shows an example of a single parallel for loop, compiled in the LLVM IR using the new parallel loop annotations and metadata.

Initially, we modified the Halide front-end to translate pipelines scheduled with the `parallel()` command into parallel LLVM loops like the one in 4. This has lead to a smaller code output, since now sequential and parallel loops are very similar, with only the `llvm.loop` and `llvm.mem.parallel_loop_access` metadata making them different. What you may notice is that the metadata in figure 3 are self-referential. This is done to ensure that metadata is distinct and unique in the single parallel loop case. In the nested loops case, we need to use linearly more metadata to denote dependency relations. As you can see in figure 5, the nested metadata `!0 = !{!1, !2}` is how loop-carried dependencies are annotated. The variables which are marked by either `!1` or `!2` do not suffer from loop-carried dependencies, while the ones marked with `!0` are accessed inside both loops. To solve loop-carried dependencies we will use the polyhedral optimization framework Polly [2] as a dynamically linked library to the default LLVM optimizer. Other optimization passes that are needed are explicitly declared using the LLVM flags. Most of the ones we want are in the *auto-vectorization* collection included by default with the latest LLVM version.

```
      for(i=0; i<N; i++) {
(1)     a[i] = b[i] + c[i];
(2)     b[i+1] = a[i]+ d[i];
(3)     c[i+1] = e[i] + f[i];
      }
```




```
c[1:N+1] = e[0:N] + f[0:N];
for(i=0; i<N; i++) {
  a[i] = b[i] + c[i];
  b[i+1] = a[i] + d[i];
}
```

Fig. 6. Demonstration of our dependency elimination algorithm. Statements (1) and (2) form a strongly connected component and are thus executed sequentially. Statement (3) is vectorizable. Applying the steps (2)-(4) of the algorithm produce the second simplified graph. Step (5) produces the final vectorized code. We are using square brackets notation to demonstrate parallelism.

C. Fine-Grain Parallelism

There are two classic techniques for extracting fine-grain parallelism, vectorization and software pipelining. Researchers first developed vectorizing technology for vector supercomputers such as the Cray-1 [10]. More recently, compiler engineers have adopted it for compilation to multimedia extensions. Vectorization identifies and extracts data parallelism, which implies the ability to safely execute the same operation on multiple data elements concurrently. Vectorization converts a sequential loop into a parallel version that utilizes a processor's vector instruction set.This transformation involves a substantial reordering of operations; semantically, a vector instruction computes multiple values before committing any of the results.Vectorization is only legal if it preserves dependencies in the original loop. As a result, it relies heavily on the ability to accurately characterize dependencies among operations. A straightforward reaching-definitions data-flow analysis uncovers dependencies between scalar variables. The primary difficulty therefore, is the accurate identification of dependencies among memory operations. A simple approach that conservatively assumes dependence between any load and store almost always prevents vectorization. Instead of creating new metadata to mark vectorizable loops, we decided to use the existing metadata used to track loop-carried dependencies, which is `llvm.mem.parallel_loop_access` and we used it before for parallel loop annotation. Using the metadata lists that we saw in figure 5 we can infer where a loop-carried dependency may exist.

After identifying dependencies we created an algorithm for solving them. The basic steps for vectorizing a nested loop are the following:

1) Create data-dependence graph for the loop body. Nodes represent statements in the inner loop and edges denote dependencies between statements.
2) Identify cycles in the graph using Tarjan's algorithm for strongly connected components [11]. Statements involved in a dependence cycle must be executed sequentially. The rest are vectorizable.
3) Partition the graph into *pi-nodes*, where each cluster represents a strongly connected component, and remove edges contained within clusters. The resulting graph is acyclic.
4) Run a topological sort algorithm on the graph to determine a valid pi-node ordering.
5) For each vectorizable node, emit a vectorized statement. Otherwise, replace it with a sequential loop to execute all operations in the original program order.

For example, consider the simple for loop in figure 6 and see how the above algorithm is applied to isolate vectorizable operations from serial execution code. Notice that the vector operations execute first, even though the originating scalar statement appears later in the source loop. This reordering is a result of the topological sort, executed in step (4) of our algorithm. Also note that in figure 6, the final version of the vectorized code is only a symbolic representation of the actual machine code that will be executed. We first have to apply strip-mining given the width of vector registers in the given system. The traditional vectorization algorithm isolates vector and scalar computations in separate loops. We can prevent that by performing an optimization technique called *loop fusion* [12] to merge loops of the same type when the dependencies allow it.

IV. Results

In our results we will show that by lowering parallel loops to the IR level and using Polly on the new IR, we can achieve a speedup of 1.6x-8x depending on the Polly flags used. The results of our out-of-order vectorization algorithm with strongly-connected components are also positive and show a speedup of 1.5x-2x. Both those optimizations where made possible by propagating parallelism to the level of the optimizer. We

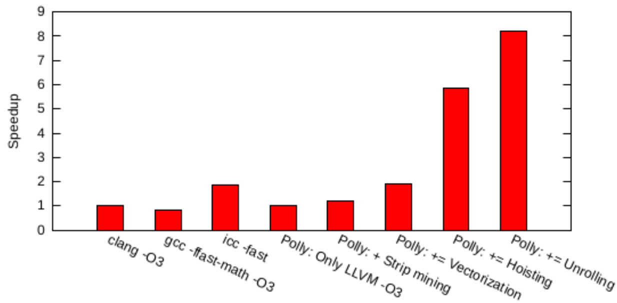


Fig. 7. Optimizing float matrix multiplication in Halide with Parallel Loops IR and Polly, clang 2.8, GCC 4.5 on Intel Core^{TM} i5 CPU at 2.40GHz

have divided our results in two categories. One is the results obtained by polyhedral optimizations in parallel loops using Polly and our Parallel Loop IR. The other is the results obtained from out-of-order vectorization and fined-grained parallelism.

A. *Parallel Loops with Polly*

To benchmark Halide with our Parallel Loops IR and Polly, we wrote a simple floating-point matrix multiplication algorithm in Halide, compiled it to LLVM IR together with our Parallel Loops annotations and then passed it to Polly with a variety of optimization flags. The options we tried were; Polly, Polly with Strip Mining enabled, Polly with it's own vectorization algorithm enabled, Polly with hoisting and finally Polly with loop unrolling, which had the biggest speedup compared to clang and GCC in the O3 level, which we used as our base measurements. As you can see in figure 9, different optimization levels of Polly yield different results, with the best in terms of performance being Polly with the *loop unrolling* flags enabled. Do not be be alarmed by the fact that Polly, with its own vectorization scheme is only 2x faster in terms of performance. Our parallel loops are both polyhedral optimized, unrolled and vectorized using our own algorithm, so we expect to see a greater performance increase than the fastest Polly implementation, which is Polly with loop unrolling. This is because Halide does loop unrolling as part of its runtime optimizations. In the future maybe it would beneficial to try and move loop unrolling in the compile phase of Halide, so it happens statically and before-the-fact.

B. *Out-of-Order Vectorizer*

To benchmark our Out-of-Order vectorization algorithm, we run a series of parallel benchmarking tests

Benchmark	Source	Description
093.nasa7	CFP92	7 kernels used in NASA applications
125.turb3d	CFP95	Turbulence modeling
301.apsi	CFP2000	Meteorology: pollutant distribution
146.wave5	CFP95	Maxwall's equations
103.su2cor	CFP95	Monte-Carlo method
172.mgrid	CFP2000	Multi-grid solver: 3D potential field
101.tomcatv	CFP95	Vectorized Mesh Algorithm

Fig. 8. Benchmark algorithms we evaluated out-of-order vectorization against.

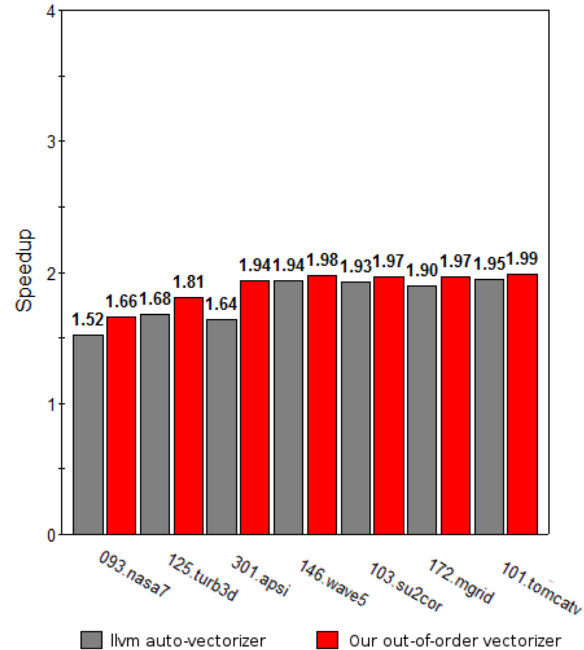


Fig. 9. Speedup of the Out-of-order vectorizer compared to the stock LLVM auto-vectorizer.

which are openly available on the internet. In figure 8 you can see a short explanation of what algorithm each benchmark evaluates against. Then we compare the results of our algorithm with the stock LLVM auto-vectorization algorithm. In all cases, our out-of-order vectorization algorithm that uses parallel loop and loop-carried dependency annotations, is faster than the stock LLVM auto-vectorizer. The performance speedup we observed was 1.6x-2x compared to non-vectorized code and an average of 1.2x better than the LLVM auto-vectorizer. You can see how our vectorizer evaluated against the LLVM one in figure 9. The x-axis includes all the different benchmarks evaluated, as shown in figure 8. The y-axis measures the calculated speedup of the benchmark, compared to the stock LLVM auto-vectorizer for scale.

V. DISCUSSION

The performance increase from lowering parallel loops to the level of the optimizer is staggering. We applied two different optimization passes on top of our Parallel IR metadata. One was the Polly polyhedral optimization framework and the other was our own out-of-order vectorizer. In the first case we saw a performance increase of 2x-8x in a standard matrix multiplication benchmark written in Halide. Writing the matrix multiplication algorithm in Halide was much easier, due to Halide's kernel-like parallel semantics and also faster than writing hand-optimized code. This is another proof that performance optimization should be the job of the compiler and not the programmer. Keeping track of all memory hierarchies and communications is a tedious task and programmers rarely ever explore the set of possible memory arrangements and schedules for any given algorithm. On the other hand, Halide using auto-tuning [13] can test many different memory configurations and schedules before deciding on the best one, by incorporating a metric in the performance search space. This way, the search for an optimal memory configuration and scheduling can converge to a near-optimal one multiple orders of magnitude faster than a human could by trial and error.

So why does LLVM not know about parallelization in the IR level already. There are hundreds of parallel front-ends and runtimes for LLVM, such as Intel cilk [3], pthreads [14], OpenMP [15], MPI [16], UPC [17], with different specifications and features. Creating one intermediate representation that successfully describes all of these is hard. Some efforts have been made in that direction, as we saw with SPIRE [8] but they end up generalizing the execution model or resulting in an ambiguous execution schedule. Parallel loops are something almost all parallel runtimes share, so even if we are not ready to lower all parallelism to the IR, we can feel confident propagating parallel loops to the optimizer level. Due to the lack of a common Parallel IR all runtimes can speak to, most parallel IR code generated by an LLVM based front-end results in function calls to handle parallelism during runtime. By having the explicit parallelism information in the optimizer level we can schedule code execution statically before runtime, yielding a performance speedup, as long as all loop-carried dependencies have been resolved either by Polly or by our out-of-order vectorizer.

When resolving loop-carried dependencies across multiple levels of nested parallel for loops the schedule space increases exponentially with every level. This is another reason why manually searching through the solution space is inefficient. The in-degree of the dependency graph seen in figure 6 increases exponentially with the level of nesting and in many applications this is prohibiting in terms of manual performance optimization. We have showed that by lowering explicit loop parallelism and using auto-tuning as a metric to explore this space, we can achieve a near-optimal performance automatically.

Loop-carried dependencies also affect the speedup observed by vectorization. The LLVM auto-vectorizer we evaluated against doesn't currently use the `llvm.mem.parallel_loop_access` metadata to predict loop-carried dependencies and thus tries to guess these relationships, resulting in a lower performance compared to our out-of-order vectorizer. The performance increase was a speedup of 1.2x compared to LLVM, while our vectorization algorithm is simpler. The reason we can get better performance with a weaker algorithm is the propagation of parallel dependencies, which can be solved statically and before-the-fact, reducing the time needed for runtime optimization.

The next obvious step would be to combine parallel loop optimizations with Polly and our out-of-order vectorizer to create a single pass that is aware of both thread-level parallelization as well as SIMD parallelization. This way we can go deeper in the possible solution space, comparing different memory configurations that schedule loops in a parallel runtime such as OpenMP [15] and executes the leaves in the optimization tree (figure 3 using vector instructions, scheduled by our out-of-order optimizer. We can do that by porting our out-of-order vectorization algorithm into Polly and replacing the current vectorization algorithm, while maintaining the effectiveness of other techniques such as loop unrolling and hoisting, in order to get results even greater than the ones shown in figure 7.

The same way we optimized parallel code to run across threads, we can extend our scheduler with a distributed runtime such as GasNET [9] or Legion [5] to schedule tasks across machines. In that case, the basic principles of generating memory configurations and auto-tuning to find the best one, can also apply in a distributed shared memory environment. There has been a lot of work in generating a dynamic distributed scheduler but not with great results. Our scheduler is static, which means it will have to find the ideal memory arrangement across machines and caches before getting an input or input size. To do that, we will again resolve to the LLVM IR to annotate distributable code similarly to how we annotated parallelizable loops. This way, we can have a general parallel and

```
ImageParam input(UInt(32), 1);
Func blur_x("blur_x");
Var x("x");

//1D blur
blur_x(x) = (input(x) +
             input(x + 1) +
             input(x + 2)) / 3;

// schedule
blur_x.distributed(x)
      .parallel(x)
      .vectorize(x);
```

Fig. 10. Explicit distributed and parallel scheduling example in Halide for a one-dimensional array blur filter

distributed schedule or many general schedules, which will be auto-tuned during runtime provided the input size. The only difference between our scheduler, which optimizes around parallel threads, memory and vector instructions as seen in figure 6 and a static distributed scheduler, would be another level in the optimization tree above the parallel loop level, which we will call distributed loop. Please note here that this scheduler will not be capable of generating distributed schedules automatically, before receiving the Halide schedule from the programmer. An example of a distributed Halide schedule can be found in figure 10, notice how the programmer can explicitly declare the hierarchy of optimizations, calling `distributed(blur_x)` first, then `parallel(blur_x)` and finally `vectorize(blur_x)`. If you notice our optimization tree in figure 6 you can see how a Halide schedule can easily be transformed in an optimization tree, as long as memory dependencies and loop-carried dependencies are resolved.

VI. Conclusion

By lowering parallelization from the Halide front-end to the LLVM IR level we can statically generate parallel IRs. These representations are then optimized using the Polly polyhedral optimization framework and our own out-of-order vectorizer, before runtime. During runtime, Halide can schedule parallel loops better, since it knows everything about their size, scope and loop-carried dependencies, and generate a parallel execution tree. This creates a performance increase of 2x-8x for the Polly polyhedral optimizer and 1.2x-1.6x for the out-of-order vectorizer. Combining these two techniques we can get a theoretical performance gain of up to 10x. In conclusion, propagating high level parallelism to the lower levels of the compiler can offer performance benefits. In the future, distributed annotations will also be added to the LLVM IR, to add another level to the optimization tree and making it possible to run high-level, high-performance, distributed, image processing code. We have shown that taking away the responsibility of memory arrangement, communication, loop unrolling, parallelizing and vectorizing from the programmer and giving it to the compiler can yield staggering performance increases. In the future, we would like to include more metadata to the LLVM IR, making it possible to express rich parallel semantics in the lowest levels of the compiler.

VII. Acknowledgments

I would like to thank my Research Advisor Saman Amarasinghe, for allowing me to work on the things that I like and for providing constant guidance and turning me to the right direction whenever that was needed. I would also like to thank Research Scientist Shoaib Kamil, for providing me with guidance even in low-level implementation details and for teaching me everything I know about compilers. I would also like to thank MIT and CSAIL, for providing a space where ideas can flourish, as well as the SuperUROP program, that allowed me to go further down the road of research before even getting my degree. I hope more students will benefit from that program in the future and more departments will offer such opportunities to their undergraduates.

References

[1] Ragan-Kelley, Jonathan, et al. ”Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines.” ACM SIGPLAN Notices 48.6 (2013): 519-530.

[2] Grosser, Tobias, et al. *Polly-Polyhedral optimization in LLVM*, Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT). Vol. 2011. 2011.

[3] Blumofe, Robert D., et al. Cilk: An efficient multithreaded runtime system. Vol. 30. No. 8. ACM, 1995.

[4] Jskelinen, Pekka, et al. ”pocl: A performance-portable OpenCL implementation.” International Journal of Parallel Programming (2014): 1-34.

[5] Bauer, Michael, et al. ”Legion: expressing locality and independence with logical regions.” Proceedings of the international conference on high performance computing, networking, storage and analysis. IEEE Computer Society Press, 2012.

[6] Chang, Chen-Ting, et al. ”A translation framework for automatic translation of annotated LLVM IR into OpenCL Kernel function.” Advances in Intelligent Systems and Applications-Volume 2. Springer Berlin Heidelberg, 2013. 627-636.

[7] Chamberlain, Bradford L., David Callahan, and Hans P. Zima. ”Parallel programmability and the chapel language.” International Journal of High Performance Computing Applications 21.3 (2007): 291-312.

[8] Khaldi, Dounia, et al. ”The Incremental Design of Parallel Compiler Intermediate Representations using SPIRE.”

[9] Bonachea, Dan. ”GASNet Specification, v1. 1.” (2002).
[10] Russell, Richard M. ”The CRAY-1 computer system.” Communications of the ACM 21.1 (1978): 63-72.
[11] Tarjan, Robert. ”Depth-first search and linear graph algorithms.” SIAM journal on computing 1.2 (1972): 146-160.
[12] Kennedy, Ken, and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. Springer Berlin Heidelberg, 1994.
[13] Ansel, Jason, et al. ”Opentuner: An extensible framework for program autotuning.” Proceedings of the 23rd international conference on Parallel architectures and compilation. ACM, 2014.
[14] Nichols, Bradford, Dick Buttlar, and Jacqueline P. Farrell. ”Pthreads programming.” (1998).
[15] Dagum, Leonardo, and Ramesh Menon. ”OpenMP: an industry standard API for shared-memory programming.” Computational Science & Engineering, IEEE 5.1 (1998): 46-55.
[16] Gropp, William, et al. ”A high-performance, portable implementation of the MPI message passing interface standard.” Parallel computing 22.6 (1996): 789-828.
[17] UPC Consortium. ”UPC language specifications v1. 2.” Lawrence Berkeley National Laboratory (2005).

Lefteris Ioannidis Lefteris Ioannidis is a student in the Electrical Engineering and Computer Science department of MIT and a SuperUROP researcher in the COMMIT group of CSAIL. He has research work in Compilers and Programming Languages, Computer Architecture and Systems. He will be graduating from MIT in 2016 with an M.Eng in Computer Systems.