Breaking SIMD Shackles with an Exposed Flexible Microarchitecture and the Access Execute PDG

Venkatraman Govindaraju
Department of Computer Sciences
University of Wisconsin-Madison
Email: venkatra@cs.wisc.edu

Tony Nowatzki
Department of Computer Sciences
University of Wisconsin-Madison
Email: tjn@cs.wisc.edu

Karthikeyan Sankaralingam
Department of Computer Sciences
University of Wisconsin-Madison
Email: karu@cs.wisc.edu

***Abstract*—Modern microprocessors exploit data level parallelism through in-core data-parallel accelerators in the form of short vector ISA extensions such as SSE/AVX and NEON. Although these ISA extensions have existed for decades, compilers do not generate good quality, high-performance vectorized code without significant programmer intervention and manual optimization. The fundamental problem is that the architecture is too rigid, which overly complicates the compiler's role and simultaneously restricts the types of codes that the compiler can profitably map to these data-parallel accelerators.**

We take a fundamentally new approach that first makes the architecture more flexible and exposes this flexibility to the compiler. Counter-intuitively, increasing the complexity of the accelerator's interface to the compiler enables a more *robust* and *efficient* system that supports many types of codes. This system also enables the performance of auto-acceleration to be comparable to that of manually-optimized implementations.

To address the challenges of compiling for flexible accelerators, we propose a variant of Program Dependence Graph called the Access Execute Program Dependence Graph to capture spatio-temporal aspects of memory accesses and computations. We implement a compiler that uses this representation and evaluate it by considering both a suite of kernels developed and tuned for SSE, and "challenge" data-parallel applications, the Parboil benchmarks. We show that our compiler, which targets the DySER accelerator, provides high-quality code for the kernels and full applications, commonly reaching within 30% of manually-optimized and out-performs compiler-produced SSE code by $1.8\times$.

***Index Terms*—SIMD, Vectorization, DySER, Access Execute Program Dependence Graph, Accelerators**

I. Introduction

Most modern processors include ISA extensions for vector operations like SSE/AVX, Altivec or NEON, which are designed to accelerate single thread performance by exploiting data-level parallelism (DLP). These SIMD operations provide energy efficiency by reducing per-instruction overheads, and performance by explicitly defining parallel operations. Although programs can be vectorized manually with assembly or compiler intrinsics, automatic support is a desirable solution because it relieves the burden of performance and portability from the programmer.

To this end, decades of compiler research has yielded a plethora of automatic vectorization techniques [4], [23], [21], [22], [26], [36], [8]. Yet, most modern compilers fail to come close to the performance of manually vectorized code. Maleki et al. show that for the GCC, XLC, and ICC compilers, only 45-71% of synthetic loops, and 13-18% of media applications can be vectorized [17]. Moreover, for these applications, manual vectorization achieves a mean speedup of $2.1\times$ compared to the automatic vectorization.

We posit that this enormous disparity is not because of insufficient compiler development or missing optimization modules, rather, it alludes to fundamental limitations of short-vector SIMD architectures. By studying auto-vectorizing compilers and the applications that are poorly vectorized, we observe that there are limitations imposed by short-vector SIMD architectures. Essentially, SIMD acceleration suffers overheads when executing control flow, loops with carried dependencies, accessing strided or irregular memory, and partially vectorizable loops. Table I describes the "shackles" that limit SIMD acceleration for each of the above code features, and summarizes solutions proposed by researchers to alleviate these limitations. Specifically, it lists the architecture support for each code feature, the responsibility of the compiler in generating code for the feature, and the overall effectiveness of the approach for that feature. We elaborate on these approaches below, which are classified into three broad categories.

SIMD Extensions: As shown in the first three rows in Table I, prior works propose several extensions to the SIMD model to address these challenges to exploit DLP [34], [8]. In general, the compiler is unable to effectively map applications to the architecture mechanisms. There are many compiler-only approaches [13], [35], [15], but they are all somewhat limited in the end by SIMD's rigidity.

Other DLP Architectures: Another approach is to use alternative architectures focused on data-level parallelism. GPUs [19] are a mainstream example and address SIMD's challenges, providing significant performance through data-parallel optimized hardware. The disadvantages are that programs in traditional languages have to be rewritten and optimized for the specific architecture used. From a hardware and system perspective, the design integration of a GPU with a general purpose core is highly disruptive, introduces design complexity, requires a new ISA or extensive ISA extensions, and adds the challenges associated with a new system software stack.

The Vector-Thread architecture is a research example that is even more flexible than the GPU approach, but is difficult to program [16]. Sankaralingam et al. develop a set of microarchitectural mechanisms designed for data-level parallelism without being inherently tied to any underlying architecture [29]. One of the recent DLP architectures is the Intel's Xeon

		Control Flow	**Strided Access**	**Loop Carried Dep.**	**Partial Vectorization**	**Impossible Vectorization**	
SIMD Arch.	**Shackle** / **Strategy**	Masking Overhead, Computation Redundancy	Shuffling Overhead, Complex Data Structure Transforms	Fixed Parallel Datapath, Costly Dependence Breaking Transforms	Shuffling Overhead, Difficult Cost-Benefit Analysis	Fixed Parallel Datapath	**Foremost Limitation**
Traditional Vector Machines [34]	Very efficient highly-parallel loops.	A. Masked Operations C. Manage Condition Subsets E. Medium Effectiveness	No Solution	No Solution	No Solution	No Solution	Limited Applicability
Vector + Scatter/ Gather [34]	Flexible Memory Access	A. S/G & IOTA instruction C. Manage Condition Subsets E. Medium Effectiveness	A. Naturally Supported C. Manage Index Vector E. High Effectiveness	No Solution	A. Naturally Supported C. Manage Index Vector E. High Effectiveness	No Solution	Compiler Complexity
Vector+ Multi-Layout Memory [8]	Highly efficient&general strided access	No Additional Support	A. Special Hardware/Instrs. C. Programmer Macros E. Very High Effectiveness	No Solution	No Solution	No Solution	Programmer Burden
Vector Threads [16]	Efficient DLP and TLP	A. Multi-threading C. Splitting Loop Iterations E. High Effectiveness	No "vectorized" strided access.	A. Cross-VP Queue C. Identify Deps./Add Comm. E. High Effectiveness	A. Thread and Vector Ops C. Compiler tradeoff Analysis E. High Effectiveness	A. Multi-threading C. Splitting Loop Iterations E. High Effectiveness	Integration to GPP/Compiler Complexity
GPUs [19]	Programing Model +hardware relieves compiler burden.	A. Warp Divergence C. Annotate Splits/Merges E. Medium Effectiveness	A. Dynamic Coalescing C. No Compiler Cost E. High Effectiveness	Programmer Responsible	A. Multi-Threading C. Little Compiler Cost E. High Effectiveness	A. Multi-Threading C. Little Compiler Cost E. High Effectiveness	Programmer Burden
CGRA: DySER [10], [9]	Broadens Applicability, + energy efficient	A. Native Control Flow C. Utilize PDG Information E. High Effectiveness	A. Flexible I/O Interface C. Utilize AEPDG E. High Effectiveness	A. Configurable Datapath C. Identify Deps., Unroll E. High Effectiveness	A. Flexible I/O Interface C. Utilize AEPDG E. High Effectiveness	A. Pipelined Datapath C. Utilize PDG Information E. High Effectiveness	Unproven Research Architecture

Legend: A: Architectural Support, C: Compiler Responsibility, E: Effectiveness Overall

TABLE I. TECHNIQUES TO ADDRESS SIMD SHACKLES

Phi, which accelerates data parallel workloads through wider SIMD [32] and hardware support for scatter/gather. In general, DLP architectures do not perform well outside the data parallel domain and have additional issues when integrating them with a processor core.

Coarse-grained Reconfigurable Architectures (CGRAs): Recent research efforts in accelerator architectures like C-Cores [38], BERET [11], and DySER [10], provide a high-performance in-core substrate. We observe that they are converging toward a promising set of mechanisms that can alleviate the challenges of compiling for SIMD. In this work, we argue that CGRAs address the microarchitectural rigidity of SIMD and improve compiler effectiveness, while also avoiding programmer burden. By judiciously exposing their mechanisms through well defined interfaces, we propose that in-core accelerators can be freed of SIMD's limitations, and that conventional CPU architectures can provide performance on a broad spectrum of DLP applications with little or no programmer intervention. The three microarchitectural mechanisms we focus on are configurable datapaths, native control-flow execution and flexible vector I/O between the accelerator and the core. We show how these three mechanisms and a co-designed compiler are enough to address the challenges in exploiting DLP with in-core accelerators.

Our accelerator aware compiler solution consists of two steps. First, we develop a variant of the program dependence graph called the Access Execute Program Dependence Graph (AEPDG) that captures the spatial and temporal communication and computation aspects of the accelerator and accelerator-core interface. We then develop a series of compiler transformations, leveraging existing vectorization techniques to operate on this AEPDG to handle various "irregular" and SIMD-shackled code, ultimately creating a flexible and efficient compiler. To evaluate these ideas and concretely describe them, this paper uses the DySER architecture as the compiler's target. Overall, our paper's contributions are:

- We develop a variant of Program Dependence Graph, called the Access Execute Program Dependence Graph (AEPDG), to capture the temporal and spatial nature of computations.
- We develop compiler optimizations and transformations, using the AEPDG, to produce high quality code for accelerators.
- We describe the overall design and implementation of a compiler that constructs the AEPDG and applies these optimizations. We are publicly releasing the source code of our LLVM-based compiler implementation that targets DySER [2].
- We demonstrate how a CGRA's flexible hardware (specifically DySER), the AEPDG representation, and compiler optimizations on the AEPDG can enable specific transformations which break SIMD's shackles and expand the breadth of data parallel acceleration.
- We perform detailed analysis on two benchmark suites to show how close the performance of automatically compiled DySER code comes to its manual counterpart's performance, and how our automated compiler outperforms ICC compilation for SSE by $1.8\times$.

This remainder of the paper is organized as follows. Section II presents the background on challenges that a vectorizing compiler face, and on the DySER architecture and its flexible mechanisms. Section III presents the AEPDG, and section IV describes our design and implementation of a compiler that uses the AEPDG to generate optimized code for DySER. Section V describes how DySER and the compiler transformations broadens the scope of SIMD acceleration. Section VI presents the evaluation and section IX concludes.

II. BACKGROUND

This section first describes classes of loops that vectorizing compilers face, and describes the issues with compiling these "challenge loops". We then describe the DySER architecture and its flexible mechanisms as a concrete example of an in-core accelerator. We also contrast its mechanisms with SIMD to show the benefits of the DySER's flexibility as outlined in Figure 1. The challenge loops and architectural details serve as background and motivation for the development and description of the AEPDG construct, compiler design, and transformations.

A. SIMD Challenge Loops

In this subsection, we describe the SIMD approach to vectorizing five classes of loops, explaining the difficulties SIMD compilers face using examples in the first two columns of Figure 6. The examples in this figure are later revisited to demonstrate the DySER compiler's approach.

Reduction/Induction: Loops which have contiguous memory access across iterations and lack control flow or loop dependencies are easily SIMD-vectorizable. Figure 6(a) shows an example reduction loop with an induction variable use. The SIMD compiler can vectorize the reduction variable "c" by accumulating to multiple variables (scalar expansion), vectorizing the induction variable by hoisting initialization out of the loop, and performing non vector-size divisible loop iterations by executing a peeled loop (not shown in diagram).

Control Dependence: SIMD compilers typically vectorize loops with control flow using if-conversion and masking. Though vectorization is possible, the masking overhead can be significant. One example, shown in Figure 6(b), is to apply a masking technique where both "sides" of the branch are executed, and the final result is merged using a mask created by evaluating the predicate on the vector "C". Note that four extra instructions per loop are introduced for masking.

Strided Data Access: Strided data access can occur for a variety of reasons, commonly for accessing arrays of structs. Vectorizing compilers can sometimes eliminate the strided access by transforming the data structure into a struct of arrays. However, this transformation requires global information about data structure usage, and is not always possible. Figure 6(c) shows the transformations for a complex multiplication loop, which cannot benefit from array-struct transformations. A vectorized version, provided by Nuzman *et al.* [22], packs and unpacks data explicitly with extra instructions on the critical path of the computation.

Carried Dependencies: SIMD compilers attempt to break loop-carried memory dependencies by re-ordering loops after loop fission, or reordering memory operations inside a loop. These techniques involve difficult tradeoffs and can have significant overheads. The example code in Figure 6(d) shows a loop with an unbreakable carried dependence, which cannot be SIMD vectorized. The statements cannot be re-ordered or separated because of the forward flow dependence through `c[i]` and the backwards loop anti-dependence on `a[i]`, creating a serial dependence chain.

Partially Vectorizable: When contiguous memory patterns occur only on some streams in a loop, SIMD compilers must carefully weigh the benefits of vectorization against the drawbacks of excessive shuffling. One example is in Figure 6(e), where the loop has two streaming access patterns coming from the arrays "a" and "b". The accesses from "a" are contiguous, but "b" is accessed indirectly through the "index" array. Here, the compiler has chosen to perform scalar loads for non-contiguous access and combine these values using additional instructions. This transformation's profitability relies on the number of instructions required to construct vector "D2".

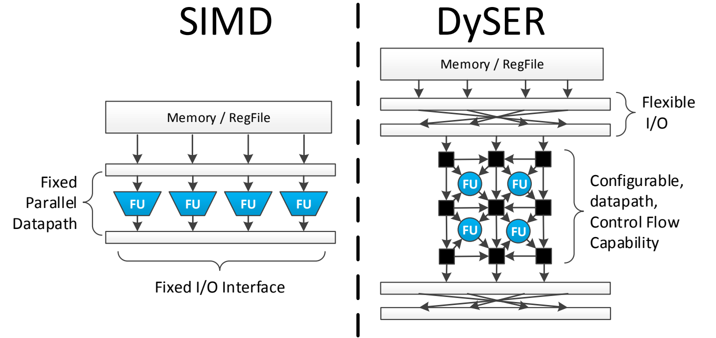


Fig. 1. Conceptual Models of Vector SIMD and DySER

B. DySER's Architecture and Execution Model

To address the challenges of SIMD compilation, we leverage the DySER architecture as our in-core accelerator. In this subsection we briefly describe DySER, and further details are in Govindaraju et al. [10], [9].

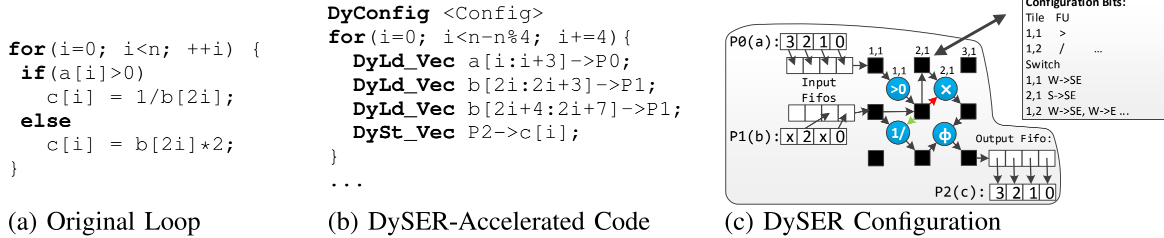
Architecture DySER is an array of configurable functional units connected with a circuit switched network of simple switches. A functional unit can be configured to receive its inputs from any of its neighboring switches. When all its inputs arrive, it performs the operation and delivers the output to a neighboring switch. Switches can be configured to route their inputs to any of their outputs, forming a circuit switched network. With this configurable network of functional units, a specialized hardware datapath can be created for a sequence of computation. It supports pipelining and dataflow execution with simple credit based flow control. The switches in the edge of the array are connected to FIFOs, which are exposed to the processor core as DySER's input/output ports. DySER is tightly integrated with a general purpose processor pipeline, and acts as a long latency functional unit that has a direct datapath from the register file and from memory. The processor can send/receive data or load/store data to/from DySER directly through ISA extensions.

Execution Model Figure 2 shows DySER's execution model. Before a program uses DySER, it configures DySER by providing the configuration bits for functional units and switches, as shown in Figure 2c. Then it sends data to DySER either from registers or from memory. Once data has arrived to DySER's input FIFO, it follows the configured path through the switches. When the data reaches the functional units, the functional units perform the operation in dataflow fashion. Finally, the results of the computation are delivered to the output FIFOs, from which the processor fetches the outputs and sends them to the register file or to memory.

C. Overcoming SIMD Challenges with DySER

As shown in Figure 1, SIMD units and DySER exhibit key similarities. They are tightly integrated to the core, are composed of many functional units to exploit fine-grained parallelism and have wide memory interfaces. However, DySER's capability to overcome the challenges with SIMD arise from three flexible mechanisms: i) configurable pipelined datapaths; ii) native control capability; and iii) a flexible vector I/O interface.

Configurable Datapath A SIMD unit's datapath is fixed to perform many equivalent operations in parallel. In contrast,



(a) Original Loop (b) DySER-Accelerated Code (c) DySER Configuration

Fig. 2. DySER Execution Model

DySER extends the flexibility of this model by providing a configurable datapath. Complex dependencies can be expressed inside the DySER hardware substrate, which is simply a heterogeneous grid network of functional units and switches.

Control Mapping Executing control flow on a SIMD accelerator requires performing both paths of a control decision, and must use explicit masking instructions to attain the final output. DySER simplifies this by natively supporting control flow inside the hardware substrate by augmenting the internal datapath with a predicate bit, and provides the ability to perform select operations depending upon the validity of predicate bit. This select operation is similar to the ϕ-function in the Single Static Assignment (SSA) form of the code.

Flexible Vector I/O SIMD instructions, without the support for scatter/gather operations–as is the case for most modern SIMD implementations like SSE/AVX–can only load and store contiguous sections of memory. This simplifies the hardware for fetching vectors from the memory system by only requiring one cache line fetch, or perhaps two if unaligned vector access is supported. DySER retains this simplicity by requiring contiguous memory on vector loads, but provides a flexible I/O mechanism which can map locations in an I/O vector to arbitrary ports in DySER. To support this, DySER's configuration includes vector port definitions, which map sequences of DySER's ports to virtual vector ports [9].

This mapping mechanism allows DySER to utilize vector instructions for communication in different paradigms, as shown in Figure 3. First, when the elements of a vector correspond to different elements of the computation, this is a "wide" communication pattern (Fig. 3a). This is most similar to SIMD's vector interface. When the elements of a vector correspond to the same element of the computation, this is a "deep" communication pattern (Fig. 3b). This corresponds to explicitly pipelining a computation. The combination of the above results in a "hybrid" pattern (Fig. 3c). Finally, when certain vector elements are masked-off or ignored, this is an "irregular" pattern (Fig. 3d). The flexibility of DySER's I/O interface, in part, gives rise to the need for a sophisticated compiler intermediate representation.

III. Access Execute PDG

The motivation to develop a new compiler intermediate representation is that, for an accelerator compiler to be successful, it must be aware of the internal operation of the architecture through spatial (via contiguous memory) and temporal (via pipelined execution) dimensions.

The Program Dependence Graph (PDG) [7] makes the data and control dependencies between instructions explicit,

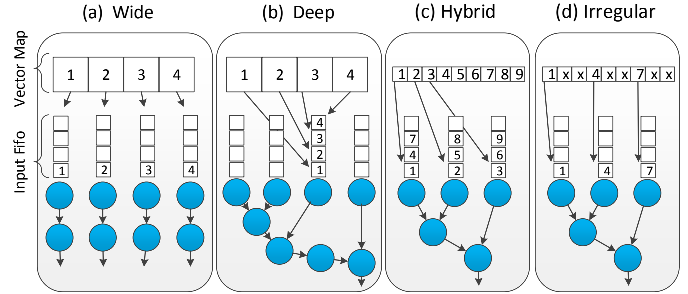


Fig. 3. Flexible I/O Mechanisms in DySER

which closely matches two flexible mechanisms that DySER provides, namely configurable datapath and control capability. However the PDG does not explicitly capture the notion of spatial access, meaning that it is unaware of the potentially contiguous access for a computation. Also, it does not have a notion of temporal execution, which corresponds to pipelining computations through the accelerator. More fundamentally, the PDG lacks a representation for the relationship between spatial access to the memory and temporal execution in the accelerator, i.e. the PDG is unaware of the correspondence between the contiguity of inputs and outputs of a particular computation through subsequent iterations. To address this shortcoming, we develop the AEPDG, which captures exactly this relationship with special edges in PDG.

Definition and Description: The AEPDG is simply a traditional PDG, partitioned into an *access-PDG* and an *execute-PDG*. The execute-PDG is defined as a subgraph of the AEPDG which is executed purely on the accelerator hardware. The access-PDG is simply the remaining portion of the AEPDG. Additionally, the AEPDG is augmented with one or many (instance, offset) pairs at each interface edge between the access and execute PDGs. The "instance" identifies the ordering of the value into the computation, and the "offset" describes the distance from the base of the memory address. This decoupling and added information allows the compiler to efficiently coordinate pipelined instances of an accelerator's computations through a flexible vectorized interface.

An Example: Figure 4 illustrates the usefulness of the AEPDG. It shows the traditional PDG on the left pane, which corresponds to original loop in Figure 2. In order to exploit the data parallelism in the loop, we can perform unrolling, which results in the "Unrolled PDG" in the second pane of figure 4. Note how this traditional PDG representation lacks awareness of the relationship between contiguous inputs and pipelineable computations. We construct the AEPDG by determining the relationship between memory accesses through iterations of the loop. Here each edge between the access and execute PDGs has an instance number and offset number. In the "AEPDG"

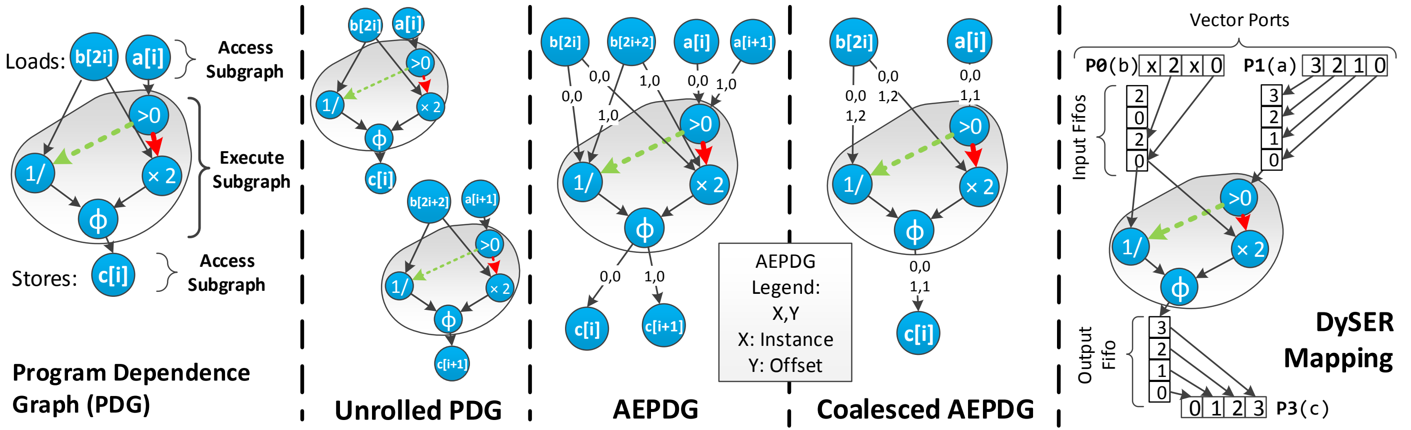


Fig. 4. Construction of the AEPDG, and Mapping to DySER

pane of Figure 4, all offset numbers are 0, because the loads and stores have not been coalesced. The next pane shows how the AEPDG keeps track of multiple instances of the computation through subsequent iterations. Some edges have multiple pairs, indicating multiple loads from the same address, and some computations are for two separate instances, indicating pipelined execution. The final pane, "DySER Mapping", shows how it is now simple to configure DySER's flexible I/O interface using the AEPDG. Each access pattern is simply given a vector port, which, when utilized by an I/O instruction, initiates a hardware mapping between the vector port and the corresponding DySER port(s).

IV. Compiling for DySER Using the AEPDG

We now describe the AEPDG compiler's design and implementation which consists of four main phases: i) Selection of regions from the full program PDG that are candidates for accelerator mapping. ii) Formation of the basic AEPDG encapsulating those code regions. iii) AEPDG transformations and optimizations tailored to the architecture. iv) Code generation of the access-PDG and execute-PDG.

A. Region Selection for Acceleration

Applications have many candidate code regions that can be accelerated, and identifying most frequently executed regions is an important task because it amortizes the cost of reconfiguration. Many conventional techniques can be repurposed for identifying the regions, including profiling, hyperblocks, and pathtrees [10] or using static techniques like identifying loops with high trip counts or using inner loops. In this work, we identify the regions with programmer inserted pragmas. Once the region for acceleration is identified, we construct the corresponding PDG using existing techniques [7].

B. Initial AEPDG Formation

Forming the AEPDG means partitioning the PDG into the access-PDG and execute-PDG. This task is important because it influences the effectiveness of the acceleration. DySER's configurable datapath and control capability give the compiler great flexibility in determining the execute-PDG.

We employ two heuristics to perform the partitioning. In the first approach, we find the backward slices of all address calculation from loads and stores inside the candidate region, and place them in the access-PDG. The remaining instructions form the execute-PDG. This works well for many data parallel applications, where each loop loads many data elements, performs a computation, and stores the result.

For applications where the primary computation is to compute the address of a load or a store, the method described above places almost all instructions in the access-PDG. Instead, we employ a method which first identifies loads/stores that are dependent on prior loads/stores. Then, we find the backward slices for the non-dependent loads/stores as we did in our first approach. For the dependent loads/stores, we identify the forward slices and make them also part of the access-PDG, leaving the address calculation of the dependent loads/stores as the execute subregion.

To select between these techniques, we simply chose the one which provides the largest execute-PDG. It is possible to develop more advanced techniques or selection heuristics, but in practice, these two approaches are sufficient.

C. AEPDG Transformations and Optimizations

To accelerate effectively, the compiler should create execute-PDGs whose number of operations is proportional to the accelerator's resources, and whose interface with the access-PDG has few connections, minimizing the I/O cost. Also, the compiler should schedule the execute-PDG to the accelerator with high throughput and low latency as the primary concerns. The initial AEPDG will need transformations to achieve the above goals. In this subsection, we present how we apply a suite of compiler transformations on the AEPDG that make it suitable for DySER. Because these transforms impose dependencies on each other, due to the close interaction of the access-PDG and execute-PDG, we conclude by commenting on how we coordinate these transformations.

Loop Unrolling for PDG Cloning: If the execute-PDG under utilizes DySER, the potential performance gains will be suboptimal. To achieve high utilization for a loop which has no loop-carried dependencies, we can clone the execute-PDG, potentially many times, until a sufficiently large execute-PDG is created. This corresponds to unrolling in the access-PDG, and reduces the trip count of the loop. This transformation exploits the spatial aspect of the AEPDG to track the links among the access subgraph nodes and cloned execute subgraph

```
Input: aepdg, dyser_model, vec_len
Output: transformed aepdg

pdgclone(aepdg, dyser_model,vec_len):=
  numClones = max_epdgs_in_dyser(aepdg.epdg,dyser_model)
  numUnroll = vec_len / numClones;

  //copy epdg numClones times
  for i = 1 to numClones
     clonedEPDG[i] = aepdg.epdg.clone()

  //unroll apdg
  for i = 1 to numUnroll
     unrolledAPDG[i] = aepdg.apdg.unroll()

  // Insert I/O edges for Execute-PDG inputs
  forall node ∈ aepdg.epdg.inputs()
     for i = 1 to numUnroll
        idx =  i%numClones
        // find the clone for the node.
        clonedNode = clonedEPDG[idx].find(node)

        forall pred ∈  clonedNode.predecessors()
           unrolledNode = unrolledAPDG[i].find(pred)
           edge(unrolledNode, clonedNode).addLabel(<i, 0>)

  // insert I/O edges for Execute-PDG outputs.
  ...

  aepdg.apdg.update_loop_control(vec_len)
  aepdg.delete_dead_nodes()
```


(a) PDG cloning and Vector Deepening (vectorizable loop)

```
Input: aepdg
Output: aepdg with coalesced loads/stores

coalesce(aepdg):=
 S = aepdg.memory_nodes.sort_in_program_order()
 while |S| != 0
   candidate = first node in S
   coalescedNodes = {Candidate}
   forall node ∈ S in sorted order
     if candidate.isLoad() == node.isLoad()
          hasConstOffset(candidate.Addr, node.addr))
       coalescedNodes = coalescedNodes ∪ {node}
   //check dependences
   aliasedNodes = S.getAliasedNodes(coalescedNodes)
   if |aliasedNodes| != 0 // We cannot coalesce them
       S = S - (aliasedNodes ∪ coalescedNodes)
       continue

   cn = coalescedNodes.sort_by_offset()
   vecNodes = cn.split(maxVecSize)
   if candidate.isLoad()
     forall vecLoad ∈ vecNodes
       forall load ∈  vecLoad
          offset = load.get_offset_from(vecNode)
          for dy_edge ∈ aepdg.dyio_edges(load)
            edge(vecLoad, dy_edge.use).addLabel(
                             <dy_edge.instance, offset>)
   // Handle stores
   ...
   //delete coalesced nodes and update S
 ...
```


(b) Load/Store Coalescing

Fig. 5. Compilation Algorithms

nodes. In the load/store coalescing transformation, described later, the compiler uses these links to combine consecutive accesses.

Strip Mining for Vector Deepening: In addition to parallelizing the loop for achieving the correct execute-PDG size, the compiler can further parallelize the loop by pipelining computations. This transformation, called strip mining, means that additional loop iterations are performed in parallel by pipelining data through the execute-PDG. The effective "depth" of the vectors is increased as an effect of this transformation.

Figure 5(a) shows the PDG cloning and vector deepening algorithm, which creates edges to track the links between the access and execute subgraphs. First, it uses the size and types of instructions in the execute-PDG and the DySER model, which includes the quantity of and capabilities of the functional units in DySER, to determine the number of execute-PDG clones and number of times the access-PDG should be unrolled. After cloning the execute-PDG and unrolling the access-PDG, it creates edges between appropriate nodes in each. These interface edges are labeled to track the spatio-temporal information.

Subgraph Matching: If the size of an execute-PDG is "larger" than the size of DySER, many configurations will be required, resulting in excess overhead per computation performed. If computations in the execute-PDG share a common structure, or formally an isomorphic subgraph, this can be exploited to pipeline the computations through this subgraph. This transformation, called subgraph matching, merges the isomorphic subgraphs, and modifies the access nodes to use temporal information encoded in the AEPDG to pipeline data.

Since computing the largest matching subgraph is a NP-complete problem, in this work, we identify these subgraphs manually. We can adapt previously proposed heuristics by Clark et al. [6] and make them AEPDG complaint. Most program regions we considered do not have common subgraphs, as they most commonly arise when the code is unrolled before AEPDG formation.

Execute PDG Splitting: When subgraph matching is insufficient to reduce the size of the execute-PDG, or when there is not an isomorphic subgraph, it becomes necessary to split the execute-PDG and insert nodes in the access-PDG to orchestrate the dependencies among the newly created execute-PDGs.

Scheduling Execute PDGs: Once the final execute PDG has been determined, we need to create a mapping between the execute-PDG and the DySER hardware itself. We use a greedy algorithm that places instructions in DySER with the lowest additional routing cost. This greedy algorithm is similar to other spatial architecture scheduling algorithms, completes quickly, and is suitable for use in production compilers.

Another potential approach is to use the recently proposed general constraint centric scheduler to map the execute-PDG to the DySER hardware [20], [30].

Unrolling for Loop Dependence: When the AEPDG represents loops without data dependence, we can use it to trivially unroll and vectorize the nodes in the access-PDG just like traditional SIMD compilers. When loops have memory dependencies across iterations, SIMD compilers usually fail or use complex techniques, such as the polyhedral model, to transform the loop such that they can be vectorized. In contrast, we simply unroll the loop multiple times and combine the dependent computation with the execute-PDG. It can accelerate the loop considerably since the execute-PDG is pipelined using DySER. Again, the AEPDG tracks the links between the unrolled nodes in the access-PDG, which can be used to combine the nodes in the load/store coalescing transform.

Traditional Loop Vectorization: We leverage several techniques developed for SIMD compilers to vectorize loops when the iterations are independent. These include loop peeling and scalar expansion [23], to maintain correctness and to increase parallelism respectively. In our compiler, we implement these traditional vectorization techniques on the AEPDG, and the techniques are designed not to interfere with its temporal and spatial properties.

Challenge Loops & SIMD Approach — Original Code | SIMD Acceleration

DySER Approach — DySER Acceleration | Execute-PDG

(a) Regular

Original Code:
```
for(i=0; i<n; ++i){
    c += a[i] * i;
}
```

SIMD Acceleration:
```
I={0,1,2,3}, C={0,0,0,0};
FOUR={4,4,4,4}
for(i=0; i<n-n%4; i+=4){
  Ld_Vec A = a[i:i+3];
  T = A * I;
  C = C + T;
  I = I + FOUR;
}
c = C[0]+C[1]+C[2]+C[3];
```

DySER Acceleration:
```
I={0,1,2,3}, C={0,0,0,0};
for(i=0; i<n-n%4; i+=4){
  DyLd_Vec a[i:i+3] => P0;
  DySnd_Vec I => P1;
  DySnd_Vec C => P2;
  DyRcv_Vec P3 => C;
  DyRcv_Vec P4 => I;
}
c = C[0]+C[1]+C[2]+C[3];
```

Execute-PDG: P0 a(i), P1 I, P2 C, ×, +4, +, ..., P3 C, P4 I

(b) Control Dep.

Original Code:
```
for(i=0; i<n; ++i) {
  if(a[i]>0) {
    c[i]=b[i]+5;
  } else {
    c[i]=b[i]-5;
  }
}
```

SIMD Acceleration:
```
for(i=0; i<n-n%4; i+=4){
  Ld_Vec A=a[i:i+3];
  Ld_Vec B=b[i:i+3];
  Temp1 = B+5;
  Temp2 = B-5;
  Mask = A>0;
  C = (Temp1 & Mask) |
      (Temp2 & ~Mask);
  St_Vec c[i:i+3]=C;
}
```

DySER Acceleration:
```
for(i=0; i<n-n%4; i+=4){
  DyLd_Vec a[i:i+3]->P1;
  DyLd_Vec b[i:i+3]->P2;
  DySt_Vec P3->c[i:i+3];
}
```

Execute-PDG: P1 A, P2 B, >, +5, -5, ϕ, P3 C

(c) Strided Access

Original Code:
```
for(i=0; i<n; ++i) {
 c[2i] = a[2i]*b[2i]
     - a[2i+1]*b[2i+1];
 c[2i+1]=a[2i]*b[2i+1]
     + a[2i+1]*b[2i];
}
```

SIMD Acceleration:
```
for(i=0; i<n-n%4; i+=4){
  Ld_Vec A1=a[2i:2i+3];
  Ld_Vec A2=a[2i+4:2i+7];
  AO = extract odds(A1,A2);
  AE = extract evens(A1,A2);
  Ld_Vec B1=b[2i:2i+3];
  Ld_Vec B2=b[2i+4:2i+7];
  BO = extract odds(B1,B2);
  BE = extract evens(B1,B2);
  CE = (AE * BE) - (AO * BO);
  CO = (AE * BO) + (AO * BE);
  c[2i:2i+3]=itrl low(CE,CO);
  c[2i+4:2i+7]=itrl high(CE,CO);
}
```

DySER Acceleration:
```
for(i=0; i<n-n%4; i+=4){
  DyLd_Vec a[2i:2i+3] => P0;
  DyLd_Vec a[2i+4:2i+7] => P0;
  DyLd_Vec b[2i:2i+3] => P1;
  DyLd_Vec b[2i+4:2i+7] => P1;
  DySt_Vec P2 => c[2i:2i+3];
  DySt_Vec P2 => c[2i+4:2i+7];
}
```

Execute-PDG: Vector Ports, P0(a) 0 1 2 3, P1(b) 4 5 6 7, Input Fifo, Output Fifo, P2(c) 0 1 2 3

(d) Unbreakable Dep.

Original Code:
```
for(i=1; i<n; ++i) {
  c[i] = a[i-1]+b[i];
  a[i] = c[i]*k;
}
```

SIMD Acceleration: Not SIMD Vectorizable

DySER Acceleration:
```
for(i=1; i<=n-n%4; i+=4){
  DyLd_Vec a[i-1] => P0;
  DyLd_Vec b[i:i+3] => P1;
  DySt_Vec P2 => a[i:i+3];
  DySt_Vec P3 => c[i:i+3];
}
```

Execute-PDG: P0(a), P1(b) 0 1 2 3, P2(a) 0 1 2 3, P3(c) 0 1 2 3

(e) Part. Vectorizable

Original Code:
```
for(i=0; i<n; i++){
  d1 = a[i];
  index = ind[i];
  d2 = b[index];
  c[i] = d1*d2;
}
```

SIMD Acceleration:
```
for(i=1; i<n-n%4; i+=4){
  Ld_Vec D1 = a[i:i+3];
  Ld b0 = b[ind[i+0]]
  Ld b1 = b[ind[i+1]]
  Ld b2 = b[ind[i+2]]
  Ld b3 = b[ind[i+3]]
  D2 = {b1,b2,b3,b4};
  C = A * B;
  St_Vec c[i:i+3]=C;
}
```

DySER Acceleration:
```
for(int i=0; i<n-n%4; i+=4) {
  DyLd_Vec a[i:i+3] => P0;
  DyLd b[ind[i+0]] => P1;
  DyLd b[ind[i+1]] => P2;
  DyLd b[ind[i+2]] => P3;
  DyLd b[ind[i+3]] => P4;
  DySt_Vec P5 => c[i:i+3];
}
```

Execute-PDG: Scalar Ports, Vector Port, P1 P2 P3 P4, P0(b) 0 1 2 3, P5(c) 0 1 2 3

Fig. 6. Limitations of SIMD Acceleration, and the DySER Approach (Peeled loop not shown)

Load/Store Coalescing: DySER's flexible I/O interface enables the compiler to combine multiple DySER communication instructions which have the same base address, but different offsets. We use the order encoded in the interface edges between access and execute PDGs and leverage existing alias analysis to find whether multiple access nodes can be coalesced into a single node.

Figure 5(b) shows the load/store coalescing algorithm, which tracks the offset information between the coalesced loads and the computation in the execute-PDG. It iterates through the memory instructions in program order and attempts coalescing with nodes of the same type (i.e both loads or stores) which also access addresses with a constant offset (relative to the loop induction variable). Then, if any of the coalesced nodes are dependent on other memory nodes in the AEPDG, it discards the memory dependent loads from coalescing. Coalesced nodes are split into vector-sized groups, and for each group, a new node is created with updated instance and offset information.

D. Coordinating Compiler Transformations

The compiler coordinates these transformations as follows. First, the compiler uses unrolling, subgraph matching or splitting to create AEPDGs with execute-PDGs that can be mapped to DySER successfully. Second, if the loop is vectorizable, we use vector deepening/stripmining and other traditional loop vectorization to further pipeline DySER. For non-vectorizable loops, we use the "loop unrolling for loop dependence" transform to accelerate the loop. Finally, after attaining the correct region size for DySER, we perform the load/store coalescing to reduce the communication cost.

E. Implementation

To implement our compiler, we leverage the LLVM compiler framework and its intermediate representation (IR). First, we implement an architecture independent compiler pass that processes LLVM IR and constructs the AEPDG. Second, we develop a series of optimization passes that transform the AEPDG to attain high quality code for DySER. Third, we implement a transformation pass that creates LLVM IR with DySER instructions from the access-PDG. Finally, we extend the LLVM X86 code-generator to generate DySER configuration bits from the execute-PDG. With this compiler, we can generate executables that target DySER from C/C++ source code. Our implementation is publicly released with this paper and more documentation is available here [2].

V. Broadening the Scope of Vectorization

In this section, we illustrate how the DySER compiler broadens the scope of vector-SIMD acceleration by analyzing the challenge loops introduced in Section II, which are generalizations of those found in the applications we evaluated.

Reduction/Induction: The example in 6(a) demonstrates that the techniques for traditional SIMD vectorization are also applicable for DySER acceleration, as it provides a superset of SIMD mechanisms. The third column shows transformed code after DySER acceleration, while the fourth column shows the execute-PDG, which directly corresponds to the DySER accelerator's configuration.

Control Dependence: The DySER compiler leverages the AEPDG structure to represent control flow inside vectorizable regions. The example in figure 6(b) shows how the DySER compiler can trivially observe that the control is entirely in the execute-PDG, enabling this control decision to be offloaded from the main processor. This eliminates the need for any masking instructions, reducing overhead significantly.

Strided Data Access: When non-contiguous memory prevents straight-forward loop vectorization, the DySER compiler can leverage the spatio-temporal information in the AEPDG to configure DySER's flexible I/O hardware to perform this mapping. For the code in Figure 6(c), the compiler creates interleaved wide ports to coordinate the strided data movement across loop iterations. Since the DySER port configuration is used throughout the loop's lifetime, this is more efficient than issuing shuffle instructions on each loop iteration.

Carried Dependencies: While vectorizing compilers will attempt to break loop carried dependencies, the DySER compiler takes advantage of these. The example in Figure 6(d) shows a loop which has a non-breakable loop-carried dependence. By unrolling the loop until the execute-PDG uses a proportional number of resources to the hardware, contiguous memory accesses are exposed. The loop dependencies, which are now explicit in the execute-PDG, become part of DySER's internal datapath, enabling efficient vectorization.

Partially Vectorizable: Though partially vectorizable loops pose complex tradeoffs for vector-SIMD compilers, the DySER compiler represents these naturally with the AEPDG, which is made possible by the flexible I/O interface that the DySER hardware provides. For the loop in Figure 6(e), accesses to the "a" array are vectorized, and scalar loads are used for "b". Compared to the SIMD version, the DySER compiler eliminates the overhead of additional shuffle instructions.

VI. Evaluation

This section quantitatively evaluates the AEPDG-based compiler implementation for DySER to support data-parallel execution and is organized around two main questions: i) How close to the performance of manually-optimized code does our automatically-compiled code reach? ii) How does our co-designed architecture and compiler compare to the auto vectorized GCC/ICC-compiled code for the SSE/AVX architecture?

A. Evaluation methodology

Compilers We implemented the DySER compiler in LLVM v3.2, and compared it against GCC 4.7 and Intel's compiler ICC 12.1. Since GCC auto-vectorized code always performs worse than ICC-compiled code, we only show the results for ICC. All benchmarks include the __restrict__ keyword on array pointers, where appropriate, to eliminate the need for interprocedural analysis of array aliasing.

Simulation Framework To attain performance results for DySER and SSE, we use the gem5 simulator [1], with extensions to support the DySER instructions and its microarchitecture. DySER is integrated into a 4 wide out-of-order processor with 64KB L1-D$, 32KB-L1-I$, and a tournament branch predictor with 4K BTB entries.

As described in Govindaraju et al. [9], we consider a 64-tile heterogeneous (16 INT-ADD, 16 FP-ADD, 12 INT-MUL, 12-FP-MUL, 4 FP-DIV, 4 FP-SQRT) functional-unit DySER array. It takes about 64 cycles to reconfigure DySER with 64 functional units, assuming that the L1I cache contains the configuration bits for DySER and can sustain a bandwidth of 128 bits/cycle. Area analysis comparing to SSE and AVX shows this configuration has the same area as an AVX unit and twice the area of a SSE unit.

Benchmarks We evaluate our compiler on two sets of benchmarks. First, we use a suite of throughput kernels similar to those of Satish et al. [31], which are easier to analyze, to evaluate and compare against SSE performance. This suite includes CONV (5x5 2D convolution), MERGE (merge phase of bitonic sort), NBODY (N-Body simulation), RADAR (1D Complex Convolution), TREESEARCH (Search a key through a binary search tree), and VR (Volume Rendering). Second, we consider the Parboil benchmark suite [24] as a "challenge" benchmark suite, since its scalar code is not written with any

Loop Clasification	Affected Benchmarks
No Shackles	CONV, RADAR, NBODY, MM, STENCIL, KMEANS
Loop Body Control Flow	TSRCH, VR, CutCP, LBM
Strided Data Access	FFT, MRI-Q, NNW, TPACF, LBM
Loop-Carried Dependence	NEEDLE, MERGE
Partially Vectorizable	SPMV, NEEDLE
Impossible Vectorization	None

TABLE II. CLASSIFICATION OF LOOPS EVALUATED

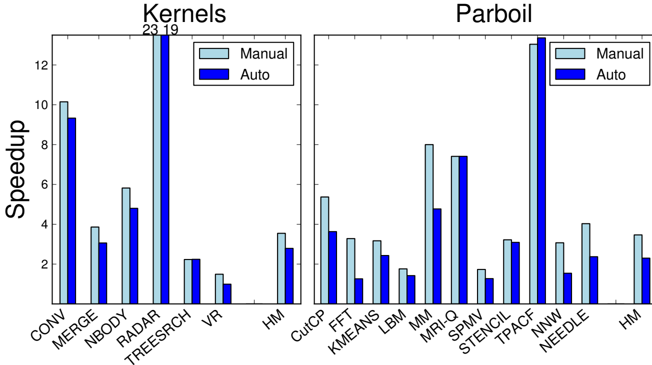


Fig. 7. Manual vs. Automatic DySER Performance

particular architecture in mind. We chose these benchmarks because they have good data level parallelism and hence are good candidates for acceleration. For both cases, we also implemented hand-optimized DySER code, and either wrote or obtained CUDA code for comparison with GPUs. Table II classifies the benchmarks according to the five challenges.

B. Automatic vs Manual DySER Optimization

Figure 7 shows the speedup of manually-optimized and compiler-generated DySER code relative to the baseline.

Kernels

Result: Manually-optimized DySER code achieves a harmonic mean speedup of $3.5\times$*, while automatic DySER compilation yields* $2.8\times$.

Analysis: As expected, manually-optimized code is faster than the auto generated code, since programmers can apply application specific knowledge when utilizing the accelerator. What is notable in these results is the number of cases where the DySER compiler generates code that achieves comparable performance to the manually optimized code. For five out of six kernels, our flexible mechanisms give the compiler enough leverage to create a datapath for the loop bodies, and also provide an efficient interface to memory. The only exception from this suite is Volume Rendering (VR), which is difficult to automatically parallelize with DySER because it requires indirect data access and cannot use DySER's flexible vector I/O. The manual version, however, computes multiple rays in parallel using the loop flattening [37] transformation on the outer loop to expose parallelism. This could be an additional optimization for our compiler.

Compiler Behavior	Benchmarks
Compiler effective	All kernels (except VR) MRI-Q, STENCIL, TPACF, KMEANS
Heuristic Tuning Reqd.	MM
Missing optimization	VR, FFT, NEEDLE, CutCP
Architecture ineffective	LBM, SPMV

TABLE III. SUMMARY OF DYSER COMPILER EFFECTIVENESS

"Challenge" benchmarks - Parboil

Result: Automatic DySER compilation yields $2.3\times$*, which comes close to the Manually-optimized speedup of* $3.4\times$

Analysis: These provide a spread of behavior and we analyze the results for the four categories in Table III.

Compiler effective (4 of 11): MRI-Q, STENCIL, KMEANS and TPACF perform equally as well in both manual and automatic compilation. This is because the flexible-IO enables the strided pattern in MRI-Q, the "deep" access pattern in STENCIL, and load coalescing TPACF. KMEANS also attains high performance, but doesn't reach that of the manual version because it uses an outer-loop unrolling technique to expose extra parallelism.

More heuristic tuning required (1 of 11): For MM, our compiler implementation fails to recognize when mapping reduction to DySER is better than mapping scalar expansion, and it suboptimally chooses scalar expansion.

Missing optimizations (4 of 11): FFT, NEEDLE, NNW, and CutCP achieve less than 70% of manually-optimized code due to missing optimizations. In FFT, the vector length needs to be dynamically chosen. NEEDLE has a long dependence chain caused by unrolling, and CutCP uses long latency functional units, causing long latency execute-PDGs for both. These benchmarks would benefit from software pipelining invocations using a outer loop. Also, the NNW benchmark uses a constant memory lookup table, which makes it hard for the compiler to reason about contiguous access. The manual version exploits the patterns in this lookup table, while the DySER compiler falls back on only partial vectorization.

Architecture ineffective (2 of 11): The architecture is ill-suited for LBM and SPMV, since even manually-optimized code provides speedup less than 80%.

C. Automatic DySER vs SSE Acceleration

We now compare the compiler+architecture performance of DySER to SSE/AVX. Figure 8 shows the speedup of auto-vectorized SSE and AVX and the speedup of compiler generated code for DySER, both measured against the same baseline.

Result: Auto-vectorization provides only about $1.3\times$ *mean speedup with SSE and* $1.4\times$ *mean speedup with AVX, whereas compiler generated code for DySER provides about* $2.5\times$ *mean speedup. In 3 of 6 kernels, and in 4 of 11 "challenge" benchmarks, DySER is* $2\times$ *faster than AVX.*

Analysis: Auto-vectorization is generally effective in the presence of regular access and no control flow. For example, the automatic compilation of CONV and NBODY performs well

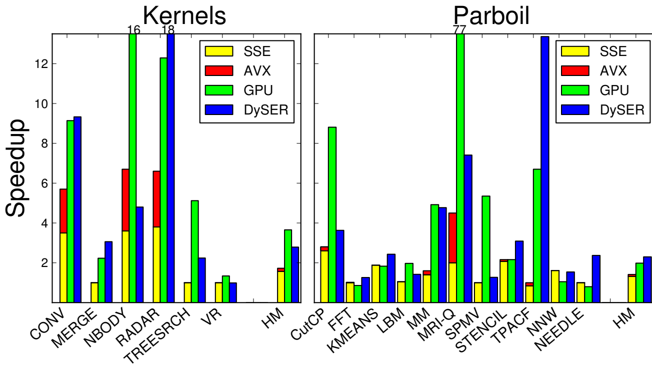


Fig. 8. Performance of DySER compiled code vs. SSE/AVX and GPU

for either SIMD or DySER. With complex access patterns or complex control flow, SIMD compilers provide no speedup (6 of 11 Parboil, and 2 of 6 kernels). DySER compilation, on the other hand, shows speedup in all but two cases. These results indicate DySER's AEPDG based compilation for flexible architectures is more effective than SIMD compilers.

Although both techniques reduce per-instruction overheads, energy reduction from DySER is significantly better than SSE, because DySER is able to handle more types of code and produce more speedups. At times, SSE increases energy consumption because of meager speedups and extra power consumption by the SIMD register file and functional units.

VII. Automatic DySER vs GPU Acceleration

GPGPUs tackle the challenges in exploiting DLP with fundamentally new programming models such as CUDA, OpenCL etc., and with an out-of-core accelerator. In contrast, our approach addresses the SIMD challenges with an in-core accelerator and targets programs written in a traditional programming model. In this section, we address how our approach compares to that of GPUs, which are a popular solution for addressing data-parallel limitations. For this study, we are comparing our fully automatic acceleration against a combined architecture and programming model, so achieving even similar performance would be a significant success. To compare against a GPU, we consider a 1-SM/8-wide GPU since its area and functional units are comparable to one DySER block integrated with a 4-wide OOO processor.

Result: For kernels, the GPU provides a mean speedup of $3.6\times$ *and compiled DySER provides* $2.8\times$. *For Parboil, the GPU provides only mean speedup of* $1.9\times$, *whereas compiler generated code for DySER provides about* $2.3\times$ *mean speedup. The GPU reduces mean energy consumption by 82% and DySER, with compiled code, reduces this by 65%.*

Analysis: Since the workloads we considered are highly data parallel and represent the best case scenario for GPU, the GPU performs better as it removes the SIMD shackles with mask generation and load coalescing. However, even with highly favorable workloads for the GPU, DySER performs better than or similar to the GPU. For example, DySER performs better than GPU in NEEDLE, because it accelerates a loop with unbreakable dependency as described in section V, whereas the GPU diagonally accesses the data, which inhibits memory coalescing and incurs runtime overheads due to extra synchronization and shared memory accesses. CutCP and MRIQ heavily use `sqrt` and `sine/cosine` and perform better on the GPU because these operations are supported with native datapath implementations (*not* because of architecture or compiler reasons). SPMV is interesting, because the GPU's use of heavy multithreading to hide memory latency works well even when accesses are very irregular, but DySER cannot vectorize these because of indirect memory accesses.

	Datapath	**Control**	**Flexible I/O**
GARP [12]	FPGA-Like	Control-mux	Memory Queue
C-Cores [38]	Synthesized	ϕ-function	Serial load/store
BERET [11]	Compound FU	None	Scalar I/O

TABLE IV. CGRA engines and their inherent DLP support

VIII. Related Work

Compiler writers have been working around SIMD's limitations in various ways to get high performance from data parallel code [23], [5]. Specific examples include if-conversion with masking to handle control-flow [3], branch-on-superword-condition-code to skip vector instructions [33], overcoming strided access limitations [22], general data permutations [28], and loop-fission to handle loop-carried dependence and partially vectorizable loops [14]. All of these techniques will necessarily incur overheads that the DySER compilation approach seeks to avoid.

The ispc compiler attacks the same problems as we do, but targets SIMD and adopts new language semantics to overcome compiler problems [27], whereas we stick with C/C++ and make the architecture more flexible. Intel Xeon Phi [32], a recent SIMD architecture, and its compiler help programmers to tackle the challenges of SIMD through algorithmic changes such as struct-of-arrays to array-of-structs, blocking, and SIMD friendly algorithms, compiler transformations such as parallelization, vectorization, and with scatter/gather hardware support [31]. However, to successfully use them, these changes require heavy programmer intervention and application specific knowledge.

Like DySER, there are numerous coarse grain reconfigurable architectures (CGRAs) that utilize a configurable datapath for acceleration [11], [38], [12], [18]. However, they have not demonstrated or evaluated SIMD compilation capability. Table IV shows the mechanisms in three representative CGRAs which provide inherent DLP support, making them potentially amenable to an AEPDG based compiler approach.

Similar to our approach, the newly proposed LIBRA architecture also uses the principles of heterogeneity and dynamic reconfigurability to build a flexible accelerator [25]. It augments a SIMD architecture with a flexible network to improve the scope of SIMD acceleration. Though this approach shows promise, effective compilation techniques have not been fully explored.

IX. Conclusion

In this work, we find that exposing an accelerator's flexible mechanisms to the compiler can liberate SIMD from its

shackles. We proposed a program representation, the AEPDG, to effectively manage the spatio-temporal relationship of the computation of an in-core accelerator. We develop a series of transformations on top of the AEPDG to generate optimized code for accelerators. We designed and implemented a LLVM based compiler, which we are publicly releasing, that leverages the AEPDG to exploit the DySER architecture's flexible microarchitecture mechanisms. Our results show the compiler is effective and outperforms SIMD compilation and architecture. Across a broad spectrum of data parallel applications, DySER achieves an average performance improvement of $2.5\times$, whereas SSE and AVX can only achieve speedup $1.3\times$ and $1.4\times$ respectively. In terms of maximum performance, we find that the DySER compiler is still falling short of manually optimized code in some cases, with 30% average performance difference. As our analysis shows, much of this is simply heuristic tuning, while a few benchmarks are ill-suited for the data-parallel model.

It is widely accepted that compiler tuning for an architecture is a multi-year effort. In that light, that one year of effort is enough to enable the DySER compiler to outperform ICC shows this approach holds promise as an alternative to SIMD.

X. ACKNOWLEDGMENTS

We thank the anonymous reviewers and the Vertical group for their comments, and the Wisconsin HTCondor project and the UW CSL for their assistance. Support for this research was provided by NSF under the following grants: CCF-0917238 and CCF-0845751. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

REFERENCES

[1] "The gem5 simulator system, http://www.m5sim.org."

[2] "Slicer - compiler for dyser. http://research.cs.wisc.edu/veritcal/dyser-compiler."

[3] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *POPL '83*.

[4] R. Allen and K. Kennedy, "Automatic translation of fortran programs to vector form," *ACM Trans. Program. Lang. Syst. 1987*.

[5] A. J. C. Bik, *Software Vectorization Handbook, The: Applying Intel Multimedia Extensions for Maximum Performance*. Intel Press, 2004.

[6] N. Clark, A. Hormati, S. Mahlke, and S. Yehia, "Scalable subgraph mapping for acyclic computation accelerators," in *CASES '06*.

[7] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, 1987.

[8] C. Gou, G. Kuzmanov, and G. Gaydadjiev, "Sams multi-layout memory: providing multiple views of data to boost simd performance," in *ICS '10*.

[9] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy efficient computing," *IEEE Micro*, vol. 33, no. 5, 2012.

[10] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *HPCA 2011*.

[11] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, "Bundled execution of recurring traces for energy-efficient general purpose processing," in *MICRO-44*.

[12] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," in *FCCM '97*.

[13] J. Holewinski, R. Ramamurthi, M. Ravishankar, N. Fauzia, L.-N. Pouchet, A. Rountev, and P. Sadayappan, "Dynamic trace-based analysis of vectorization potential of applications," *SIGPLAN Not.*, 2012.

[14] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[15] S. Larsen and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," in *PLDI '00*.

[16] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović, "Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators," in *ISCA '11*.

[17] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua, "An evaluation of vectorizing compilers," in *PACT '11*.

[18] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu, "Tartan: evaluating spatial computation for whole program execution," in *ASPLOS-XII*.

[19] J. Nickolls and W. J. Dally, "The gpu computing era," *IEEE Micro*, vol. 30, no. 2, Mar. 2010.

[20] T. Nowatzki, M. Sartin-Tarm, L. De Carli, K. Sankaralingam, C. Estan, and B. Robatmili, "A general constraint-centric scheduling framework for spatial architectures," in *PLDI 2013*.

[21] D. Nuzman and R. Henderson, "Multi-platform auto-vectorization," in *CGO '06*.

[22] D. Nuzman, I. Rosen, and A. Zaks, "Auto-vectorization of interleaved data for simd," in *PLDI '06*.

[23] D. A. Padua and M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *Commun. ACM*, 1986.

[24] "Parboil benchmark suite, http://impact.crhc.illinois.edu/parboil.php."

[25] Y. Park, J. J. K. Park, H. Park, and S. Mahlke, "Libra: Tailoring simd execution using heterogeneous hardware and dynamic configurability," in *MICRO '12*.

[26] Y. Park, S. Seo, H. Park, H. K. Cho, and S. Mahlke, "Simd defragmenter: efficient ilp realization on data-parallel architectures," in *ASPLOS '12*.

[27] M. Pharr and W. R. Mark, ""ispc: A spmd compiler for high-performance cpu programming"," in *InPar 2012*.

[28] G. Ren, P. Wu, and D. Padua, "Optimizing data permutations for simd devices," in *PLDI '06*.

[29] K. Sankaralingam, S. W. Keckler, W. R. Mark, and D. Burger, "Universal Mechanisms for Data-Parallel Architectures," in *MICRO '03: Proceedings of the 36th Annual International Symposium on Microarchitecture*, December 2003, pp. 303–314.

[30] M. Sartin-Tarm, T. Nowatzki, L. De Carli, K. Sankaralingam, and C. Estan, "Constraint centric scheduling guide," *SIGARCH Comput. Archit. News*, vol. 41, no. 2, pp. 17–21, May 2013.

[31] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey, "Can traditional programming bridge the ninja performance gap for parallel computing applications?" in *ISCA 2012*.

[32] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," in *SIGGRAPH 2008*.

[33] J. Shin, "Introducing control flow into vectorized code," in *PACT '07*.

[34] J. E. Smith, G. Faanes, and R. Sugumar, "Vector instruction set support for conditional operations," in *ISCA '00*.

[35] K. Stock, L.-N. Pouchet, and P. Sadayappan, "Using machine learning to improve automatic vectorization," *TACO 2012*.

[36] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen, "Polyhedral-model guided loop-nest auto-vectorization," in *PACT '09*.

[37] R. v. Hanxleden and K. Kennedy, "Relaxing simd control flow constraints using loop transformations," in *PLDI '92*.

[38] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: reducing the energy of mature computations," in *ASPLOS '10*.