Polly-ACC
Transparent compilation to heterogeneous hardware

Tobias Grosser
Department of Computer Science, ETH Zurich
tobias.grosser@inf.ethz.ch

Torsten Hoefler
Department of Computer Science, ETH Zurich
htor@inf.ethz.ch

ABSTRACT

Programming today's increasingly complex heterogeneous hardware is difficult, as it commonly requires the use of data-parallel languages, pragma annotations, specialized libraries, or DSL compilers. Adding explicit accelerator support into a larger code base is not only costly, but also introduces additional complexity that hinders long-term maintenance. We propose a new *heterogeneous compiler* that brings us closer to the dream of *automatic accelerator mapping*. Starting from a sequential compiler IR, we automatically generate a hybrid executable that - in combination with a new data management system - transparently offloads suitable code regions. Our approach is almost regression free for a wide range of applications while improving a range of compute kernels as well as two full SPEC CPU applications. We expect our work to reduce the initial cost of accelerator usage and to free developer time to investigate algorithmic changes.

CCS Concepts

•Computer systems organization → Single instruction, multiple data; •Software and its engineering → Compilers; *Runtime environments;*

Keywords

Polyhedral Compilation, GPGPU, Auto-Parallelization

1. INTRODUCTION

To ensure continuous growth in compute performance hardware platforms have become increasingly parallel and heterogeneous. Today, a typical workstation node does not only provide a powerful multi-core CPU, but is often combined with an even more powerful GPU accelerator. With 240 Gflop/s double precision performance for a 10-core Intel Sandybridge and 1,707 Gflop/s for an NVIDIA Titan black GPU accelerator, the accelerator is a significant source of compute power.

ICS '16, Jun 01-June 03, 2016, Istanbul, Turkey

ISBN 978-1-4503-4361-9/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2925426.2926286

Existing approaches for exploiting this compute power commonly require developers to part from the general-purpose sequential imperative programming model. Instead, developers can choose from a set of new options. Data parallel programming models [39, 43] are used to manually program accelerator-specific high-performance kernels. DSLs [18, 30, 28, 41, 12] allow for the automatic generation of high-performance code for a given problem domain. Furthermore, general purpose parallel libraries [13, 8] enable accelerator programming through the use of higher-level language abstractions that are not limited to specific problem domains. Even though these programming approaches have clear benefits, introducing them does not only result in (sometimes large) initial development costs, but the increased complexity of the ported application also hinders future development.

When software maintenance is of high importance, minimizing developer involvement is critical. For large software projects, the use of high-performance library implementations [19, 17] or pragma based programming models [15, 50] is consequently often recommended.[1] However, even though these approaches preserve the sequential C code, they still require developers to understand GPU programming. Automatic accelerator mapping techniques, as developed in the context of source-to-source compilation [48, 11, 6], can remove the need for such kind of expert knowledge, but similarly to recent work on automatic compilation towards a "GPU-first execution model" [37], their scope is limited.

To make heterogeneous computing as pervasive as automatic SIMDization, we propose a minimally invasive approach directly integrated into an imperative compiler that relies on the automatic detection and compilation of compute kernels for heterogeneous systems. Similar to vectorizing compilers, we do not focus on algorithmic changes or the generation of auto-tuned compute kernels. Instead, we aim for making heterogeneous compute available to software that otherwise would not (or only with large efforts) be ported to a heterogeneous system. Even though large legacy applications are likely to benefit, this work also provides solid foundations for the optimization of modern HPC software written in idiomatic C++ with templates, iterators, and lambda functions.

Implementing the compilation approach just proposed, we

[1]The programmer's guide for Summit [3], the next generation Oak Ridge HPC cluster, suggests: "1) Using accelerated programming libraries whenever possible, 2) Preferring high-level compiler directives such as Open MP/Open ACC over low-level frameworks such as CUDA or OpenCL."

present with Polly-ACC[2] an automatic, fully integrated heterogeneous compute compiler, that enables normal LLVM-based compilers [32] to directly generate multi-device binaries from a range of input languages.

Our contributions are:

- A minimal-invasive and fully integrated heterogeneous compute compiler.
- Extraction of heterogeneous compute specific information from low-level IR and translation to a schedule tree.
- A lightweight and effective runtime library for automatic data allocation and data transfer management.
- Experimental validation on the LLVM nightly test suite, 30 Polybench kernels, as well as SPEC 2006.

2. OVERVIEW OF THE HETEROGENEOUS COMPILER

We first introduce the high-level design (Figure 1) of our work. In a nutshell, our compiler follows a classical compiler design with a set of front-ends, a mid-level optimizer, and target-specific back-ends. We extend this design with a new heterogeneous compute mid-level optimizer, which enables sequential compilers to produce hybrid binaries.

Kernel extraction from a variety of languages

To optimize a variety of programming languages frontends are often used to translate source code to a common intermediate representation (IR). For our work we use LLVM as compiler infrastructure and apply optimizations on its IR (LLVM-IR). Given a source code file (1a) - (1d), one of the many LLVM based compilers (2a) - (2d) translates it to a language-independent IR (3) on which all subsequent transformations are performed. As a result, a range of programming languages (C/C++, Fortran, Julia, Go, ...)[3] can be optimized by our compiler.

At the compiler IR level interesting compute kernels are detected, extracted, and modeled. To prepare kernel extraction, we first apply a set of canonicalization passes (4). Next, we detect program parts consisting of (mostly) static control (SCoPs). To extract such regions we rely on an enhanced version of Polly [25], a polyhedral loop optimizer for LLVM. From Polly we obtain for each compute kernel a mathematical description in terms of Presburger sets (Section 3.1) which describe in combination with a set of precise data-dependence relations, the exact memory access behavior of each kernel.

Accelerator mapping

The execution strategy of each compute kernel is described as a schedule tree [27], which maps the individual dynamic computations in the kernel to relative execution times. A schedule tree is a hybrid data structure for modeling complex loop kernels that uses Presburger sets to describe the performed computation, but models the execution strategy as a tree of multi-dimensional schedules (and more specialized tree nodes) that can be transformed with simple tree operations. The transformation from our sequential kernel to a heterogeneous compute program takes place entirely on schedule trees.

Starting off with an initial schedule tree corresponding to the sequential source code, heterogeneous compute code is introduced. To expose parallelism and increase data locality an improved sequential schedule is obtained through a modified version of the Pluto scheduler [16] as available through isl [45] (6). We then map the parts of the schedule tree to the accelerator (7) for which this mapping is profitable. Components that cannot be mapped efficiently remain unchanged. As part of this mapping we introduce host-device data transfers and distribute the computation to device threads and thread groups. Within each kernel, necessary transfers between global and shared memory are introduced. Thread private memory is used, if deemed profitable. The mapping strategy we use corresponds to ppcg [46] and is indeed performed by calling into a recent version of ppcg.

CPU and device code generation

As a next step, the heterogeneous compute schedule is translated back to imperative compiler IR. We first translate the heterogeneous-compute schedule tree to an imperative AST. For this we use a new AST generator [27] that supports AST generation from schedule trees. The resulting AST is then translated further down to LLVM-IR. When (re)generating LLVM-IR we first translate the outer levels of the imperative AST to host code (9a), until we reach device kernels.

When reaching device kernels (9b) - (9c), some special processing is necessary. First, CUDA-specific run-time calls are emitted that launch kernels. In the kernels, accesses to private/shared memory as well as inter-thread synchronization primitives are emitted according to the applied GPU mapping decisions. Once kernel code has been generated, it is extracted into a separate IR module which is then passed to LLVM's integrated GPU backend and translated to PTX code (10). The PTX code is integrated back into our host binary by storing it as a global variable (11). At run time, the kernel code is (re)loaded just before it is scheduled for execution. The possible, but not yet implemented, generation of OpenCL code (12)/(13) follows the generation of CUDA code. Finally, the complete multi-device executable is emitted (14).

2.1 Data management library

The resulting binary is run in combination with a run-time library that manages allocations and data transfers (15) ((16)). The primary goal of this library is to optimize hybrid binaries where execution switches regularly between sequential host code and parallel device code. In this situation our run-time library ensures that data is kept on the device even while running sequential host code and is only moved back if needed.

3. PRESBURGER SETS AND SCHEDULES

The following sections introduce important concepts used in our work and give a first understanding of what kind of compute kernels can be automatically mapped by our compiler.

3.1 Presburger sets and relations

[2] http://spcl.inf.ethz.ch/Polly-ACC

[3] How effective we can optimize different input languages depends on how well language abstractions are eliminated when lowering to LLVM-IR.

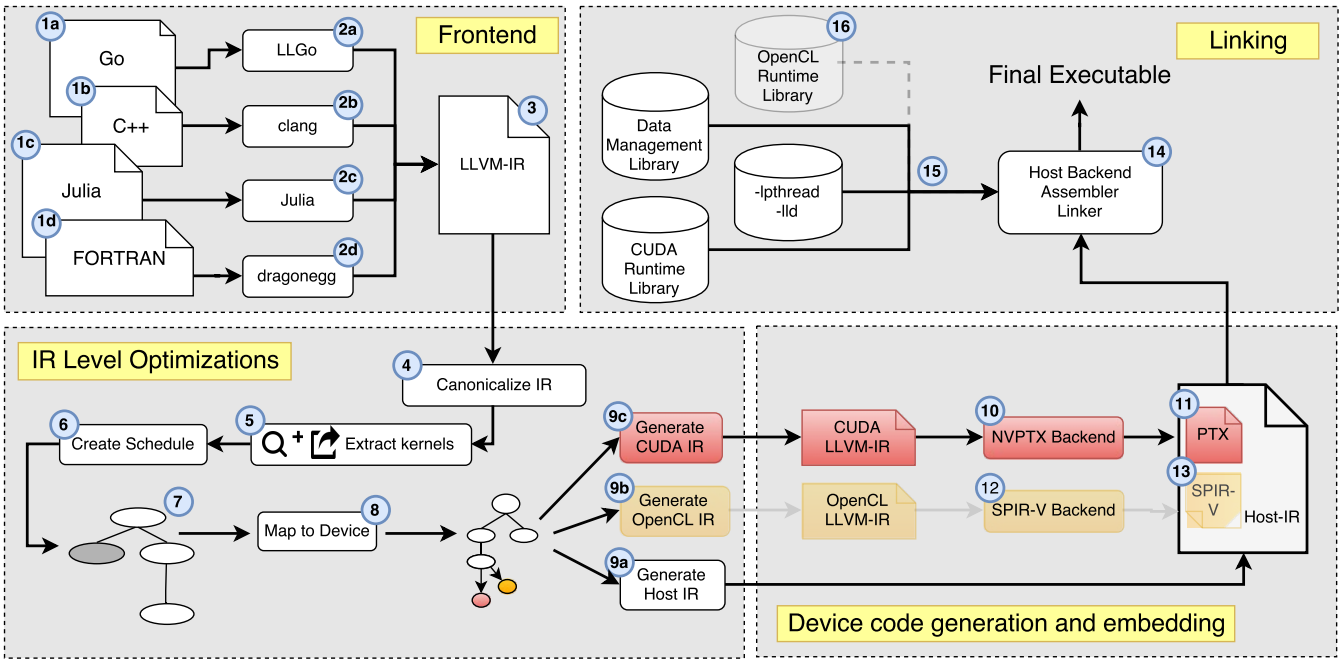


Figure 1: The architecture of our heterogeneous compiler

An affine expression e is either an integer constant (c), a variable (n), or the result of negating an affine expression ($-e$), adding or subtracting two affine expressions ($e_1 + e_2$ or $e_1 - e_2$), or multiplying an integer constant with an affine expression ($c \times e$). A quasi-affine expression additionally allows floor divisions by an integer constant ($\lfloor e/c \rfloor$) and the remainder of such a division ($e \bmod c$).

A Presburger formula p is either a boolean constant ($\top, \bot$), the result of a boolean operation ($\neg p, p_1 \wedge p_2, p_1 \vee p_2$), a quantified expression ($\forall x : p, \exists x : p$), or a comparison between different (quasi) affine expressions ($e_1 \oplus e_2, \oplus \in \{<, \leq, \geq, >\}$). An n-dimensional Presburger set s is a subset of $\mathbb{Z}^n$ where the elements of s are described by a Presburger formula. An example of a two-dimensional Presburger set is $\{\vec{d} = (d_0, d_1) \mid 0 < d_0 < 100 \wedge d_0 \leq d_1 < n\}$, of an empty Presburger is $\{(d_0, d_1, d_2) \mid \bot\}$, and of a universal set is $\{(d_0, d_1)\}$. Named Presburger sets can contain elements from differently named spaces. The set $\{[A, (i, j)] \mid i < j; [B, (i)] \mid i < 100\}$ contains for example elements from space A and B. A Presburger relation r is a subspace of $\mathbb{Z}^n \times \mathbb{Z}^m$ that is constrained by a Presburger formula and has the form $\{(d_1, d_2) \to (f_1) \mid d_1 + d_2 > f_1\}$. Presburger sets and relations are closed under normal set operations such as union, intersection, or subtraction and allow the projection onto subspaces. See Verdoolaege [47] for more background.

3.2 Modeling programs with schedule trees

For certain *sufficiently regular* program regions the computations they perform and their memory access behavior can be precisely modeled at compile time. We call such regions static control parts (SCoPs) [23]. They consist of if-conditions and for-loops, where control-flow as well as loop exit conditions are Presburger expressions and loop strides are constants. As an extension, data-dependent conditions are permitted using may-write accesses to approximate the memory behaviour. Listing 1 shows an artificial example of a static control part. It contains two statements. S1 is executed inside three loops, while S2 is sharing just the outermost loop with S1 and is additionally guarded by a data-dependent condition.

Listing 1 Simple static control program

```
   int n; int p1; int p2;
   float A[][n], float B[i];
    for (int i = 0; i < n; i++) {
      for (int j = i; j < n; j++)
        for (int k = 0; k < p1 || k < p2; k++)
S1:       A[i][j] = k * B[i]
      // Mark "A"
S2:   if (B[i]) A[i][i] = A[i][i] / B[i];
    }
```

The computational behavior of a SCoP can be modeled with schedule trees [27]. A schedule tree consists of different nodes. At its root, the *domain node* describes the statement instances executed in the SCoP using a Presburger set. As child of the domain node, a tree containing nodes of different types defines the execution order of the instances declared in the domain node. The most general node type is a *band node* which uses a Presburger relation to assign each instance a partial (possibly multi-dimensional) relative execution time. In case multiple groups of instances should be executed one after another, a *sequence node* can be used. It defines a list of schedule subtrees that are executed in sequence and is followed by filter nodes that limit the set of statement instances considered in each subtree. Besides these basic types, there is also a *marker node* to identify subtrees in the schedule and some other nodes not discussed here.

Figure 2 illustrates a schedule tree that models the example given in Listing 1. At its root, a domain node de-

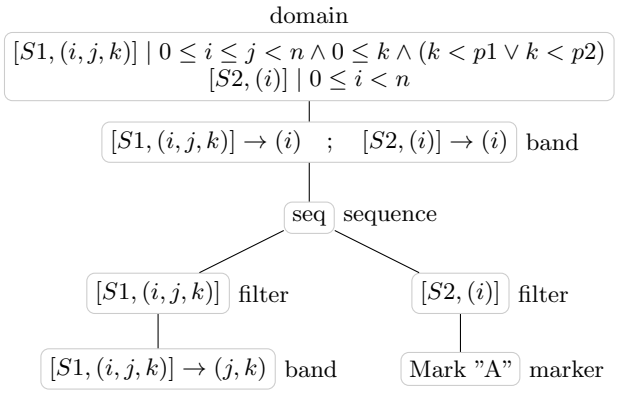


Figure 2: Possible schedule tree for code in Listing 1

scribes the iteration spaces of S1 and S2 as Presburger sets. Right after the domain node, a single dimensional band node defines a partial order that maps statement instances executed in different i-loop iterations to distinct execution times. Next, a sequence and filter node combination models the textual order of the loop nest around S1 and the code around S2. In the S1 subtree, an additional two-dimensional band node defines a partial order that corresponds to the original execution order of the j and k-loop. Combined with the previous nodes, a total execution order for S1 is established. In the S2 branch, all statement instances are already mapped to distinct execution times. The only node visible is a marker node, which is used to model the corresponding comment in the source code. Overall, the schedule tree corresponds well to the structure of the source code, but imperative control flow structures are lifted to a more abstract execution order definition.

4. LOW-LEVEL IR TO HYBRID CODE

We now discuss in detail the IR level analyses and transformations necessary to obtain a hybrid binary following the general structure introduced in Section 2.

4.1 IR canonicalization

Before our optimizer, a set of canonicalization passes is scheduled (4). These passes include memory to register translation, constant propagation, and the simplification of induction variables and control flow. We also (optionally) schedule an iterative inline – simplification – inline cycle to eliminate abstractions such as C++ templates. For large programs, running global value numbering (GVN) can reduce the number of memory loads in the program, which often reduces compile time in later passes. However, as load elimination can introduce scalar dependences that limit parallelism and hinder later device mapping, GVN remains optional.

4.2 Translate low-level IR to schedule tree

Iteration spaces and an initial schedule tree are computed by traversing the control flow graph (6). During this traversal, run-time branch conditions from if-conditions and loop bounds are collected and translated to iteration space constraints. The computational statements we model are formed by entire basic blocks. For those basic blocks, all explicit memory loads and store instructions are modeled as memory accesses. We introduce additional loads and stores to model inter-basic-block scalar use-def relations as well as PHI nodes [5]. We do not model scalar dependences introduced by induction variables, as this information is already available in the iteration space description.

4.3 Modeling memory accesses

Deriving a precise model of a program's memory access behavior is essential for accelerator mapping. In the following we discuss multi-dimensionality, multi-element-type arrays, as well as may-accesses resulting from non-affine control.

Understanding the multi-dimensionality of memory accesses is important for efficient accelerator mapping. Even though programmers commonly process multi-dimensional data, the lack of native support for multi-dimensional arrays of parametric size (C before C99, C++, Rust, ...) often forces them to use one-dimensional manually linearized arrays. Surprisingly, even languages that natively provide complete support for multi-dimensional arrays (Julia, Fortran) loose part of this information when lowering to LLVM-IR level. An interesting exception are fixed-sized arrays, for which size information is often kept at IR level. However, even in this case we cannot reliably exploit this information without proving that all memory accesses remain within bounds, as out-of-bound accesses are well defined for LLVM-IR as long as the address accessed falls into an allocated memory region.

Due to the lack of reliable dimensionality information, the original version of Polly [25] modeled all memory accesses as accesses to flat, one-dimensional arrays. As a result, otherwise easy to analyze accesses to multi-dimensional arrays of parametric size are only visible as complex polynomial index expressions (`A[i*s+j]` instead of `A[i][j]`). As such expressions can only be over-approximated with affine sets, computation of precise data dependence information becomes impossible. In the context of accelerator mapping such approximation is even more problematic, as lacking information about the precise set of data elements accessed prevents the computation of inter-device data-transfers. To ensure our compiler can derive efficient memory transfers and mappings, we recover multi-dimensional arrays of parametric size using optimistic delinearization [26] as available in Polly. To support Fortran we additionally added support for `max(p, 0)` array size expressions, which are common in gfortran generated code. In case the necessary type information is available, we can also derive accesses to multi-dimensional arrays of fixed size and validate them with validity conditions similar to the ones introduced by Polly's optimistic delinearization.

Some programs use arrays that contain data elements of different types or only conditionally access certain memory locations. Polly originally required all accesses to a given array to be of the same type. As part of this work, we extended Polly to also model arrays that contain elements of different size or alignment by introducing a smaller artificial element type which evenly divides all alignments and element type sizes in the SCoP and by expressing all memory accesses in function of this smaller type. Another common code pattern are data-dependent conditions. Polly originally required all control flow conditions to be statically analyzable, but recently gained support for non-affine conditions by over-approximating write accesses enclosed by such conditions as may-write accesses [38]. Both extensions are im-

portant for the case studies in Section 6.3 and Section 6.4.

4.4 Generate GPU-specific schedule

To generate well optimized GPU code we use the GPU optimization infrastructure of ppcg [49, 46] to translate a given schedule into a GPU-specific schedule tree. As a result of this translation the computation is partitioned between the different GPU workgroups/threads and a schedule tree is generated that models the code a specific thread instance executes. Besides the computation of this thread, the resulting schedule tree can also contain data transfers to and from shared/private memory as well as thread synchronization primitives.

4.5 Pinned host memory

CUDA distinguishes between normal and "pinned" host memory. Whereas normal memory must first be transferred into a page-locked memory buffer before it can be transferred to the device, already pinned memory can be directly transfered by the DMA engine. For manually written CUDA code it is common to directly allocate pinned memory. However, in our case we have no control over the memory allocation, as users may allocate memory at arbitrary program points. Hence, we can only pin memory at the beginning of the SCoP we optimize. If pinning should be used it is necessary to understand for each array the minimal and maximal address that will be accessed during the execution of the SCoP. We obtain this information by computing the set of multi-dimensional memory locations (Section 4.3) accessed for a given array and by taking the lexicographic minimal and maximal offset of these accesses. These offsets are then lowered to single-dimensional offset expressions, in combination with the array base pointer translated to memory addresses, and transformed in a tuple of start address and allocation size. Using this information, Polly-ACC can optionally take advantage of "pinning" at the cost of reducing swappable memory.

4.6 Cost model for offloading program regions

Even though many program regions could be mapped to an accelerator, the cost of transferring data and control limits the set of program regions for which this is beneficial. To ensure only program regions are offloaded that are likely to benefit, we use a three-layer heuristic.

The first layer verifies — before modeling the SCoP — that the program region will perform some non-trivial computation. At the moment, we require the region to either contain at least two loops or a single loop with a minimal number of instructions (currently ≥ 40). As a second heuristic we use a strategy proposed by ppcg, which — after modeling and initial scheduling — maps a subtree of the schedule tree to the accelerator, if at least one band node can be found with at least one parallel dimension and for which the dimensions of the band node are permutable (which means tiling can be applied to map these schedule dimensions to threads/workgroup). Both heuristics do not require dynamic checking and consequently do not affect program run time, if they evaluate to 'unprofitable'.

Our third and final heuristic ensures that the number of dynamic computations in the SCoP region is not lower than a configurable threshold. To verify this condition we over-approximate the domain of each statement within the SCoP as a box and derive a (possibly symbolic) expression for the product of the number of statement executions in the box and the number of instructions executed in each statement instance (currently we require at least $10*512*512$ dynamic instructions). In case the loop bounds are known constants, this expression can be evaluated at compile time. In case of parametric loop bounds, this expression will be evaluated dynamically. However, as we already ruled out the simple cases and as we ensured through over-approximation that the expression remains simple, the execution time overhead introduced by these dynamic checks is likely to be negligible.

5. DATA MANAGEMENT RUN-TIME

Multi-device binaries run on compute elements that do not share a single memory space. To ensure data is always available when and where needed, it is necessary to introduce explicit memory allocation and data transfer operations. Doing so carefully is important to ensure fast program execution.

Within each individual SCoP, we statically generate code for all data allocations and data transfers. All device memory is allocated whenever a SCoP is reached during program execution and freed right before the SCoP is left. Data transfer code is emitted under the assumption that for each individual SCoP all data needs to be transferred from the host to the accelerator and directly back to the host. Throughout the execution of the SCoP, data is maintained on the accelerator, but may be transfered back to the CPU in case parts of the SCoP are scheduled to run on the host system. Within a single SCoP, this is very efficient. However, when considering the full program, static data management is often not optimal.

When executing a full application, the interleaved execution of general-purpose host code and device-mapped SCoPs results in program behavior that is difficult to analyze statically. Repeated switches between host and device-code execution may result in subsequently executed SCoPs that work on the same data. When only relying on our static data management strategy, we would repeatedly issue (almost) identical allocations and data transfers each time a SCoP is reached. To eliminate these otherwise costly transfers, we provide a run-time library that caches allocations during program execution allowing data to remain on the device even across SCoP boundaries. Redundant host-to-device data-transfers are skipped in case data is already available on the device and is known to not have been modified by the host since being placed on the device. Similarly, device-to-host data-transfers can be delayed up to the first host-side data access such that, in the best case, data is transferred out of device memory only once for a sequence of executed SCoPs.

Listing 2 shows a 1D heat kernel that illustrates the problem just described. Each of the functions defined in "File One" is by itself a SCoP that can be run on the accelerator. When analysing the dynamic sequence of memory management operations for T = 3 as illustrated in Listing 3, we see that per-SCoP memory management introduces a large number of redundant data transfers where only a small subset is really necessary. Having a global perspective we see that only one allocation for A is needed and, due to the increasing size of its allocations, two allocations for B. As A is fully overwritten, there is no need for any host-to-device (H2D) data transfers from A. The data transfers for B are more complicated, as its allocation needs to be expanded

Listing 2 1D heat kernel consisting of multiple functions

```
// File One
void initialize(int n, X[n], float val) {
  for (int i = 0; i < n; i++)
    X[i] = value;
}
void setCenter(int n, X[n], float val,
               int offset) {
  for (int i = offset; i < n - offset; i++)
    X[i] = value;
}
void average(int n, float In[n], float Out[n]) {
  for (int i = 2; i < n - 2; i++)
  Out[i] = 0.2f * (In[i-2] + In[i-1] + In[i]
                   + In[i+1] + In[i+2]);
}
// File Two
void heat1D(int n, float A[n], float hot,
            float cold) {
  float *B = malloc(sizeof(float) * n);
  B[0] = B[1] = B[n-2] = B[n-1] = 0;
  initialize(n, A, cold);
  setCenter(n, A, hot, n/4);

  for (int t = 0; t < T; t++) {
    average(n, A, B);
    average(n, B, A);
    printf("Iteration %d finished\n", t);
  }
}
```

Listing 3 Memory transfers of heat1D kernel (Listing 2)

SCoP-Local	Global (library managed)
`// initialize`	`// initialize`
`A0 = devAlloc(&A, n)`	`A0 = devAlloc(&A, n)`
`D2H(&A, &A0, n);`	
`free(A0);`	
`// setCenter`	`// setCenter`
`A1 = devAlloc(&A, n/2)`	
`D2H(&A+n/4, &A1, n/2);`	
`free(A1);`	
`// average`	`// average`
`A2 = devAlloc(&A, n);`	
`B0 = devAlloc(&B+2, n-4);`	`B0 = devAlloc(&B+2, n-4);`
`H2D(&A, &A2, n);`	
`D2H(&B+2, &B0, n-4);`	
`free(A2);`	
`free(B0);`	
`// average`	`// average`
	`D2H(&B+2, &B0, n-4);`
	`free(B0);`
`B1 = devAlloc(&B, n);`	`B1 = devAlloc(&B, n);`
`A3 = devAlloc(&A+2, n-4);`	
`H2D(&B, &B1, n);`	`H2D(&B, &B1, n);`
`D2H(&A+2, &A3, n-4);`	
`free(B1);`	
`free(A3);`	
`// average`	`// average`
`A4 = devAlloc(&A, n);`	
`B2 = devAlloc(&B+2, n-4);`	
`H2D(&A, &A4, n);`	
`D2H(&B+2, &B2, n-4);`	
`free(A4);`	
`free(B2);`	`// A[?] host access`
	`D2H(&A, &A4, n);`

on the GPU. Before the first allocation of B is deleted, the subset of B stored on the device needs to be saved with a device-to-host (D2H) data transfer. Then, after having allocated more memory for B, the full B array is loaded back to the device. From this point we are in a steady state and no further data transfers are needed for later iterations of the time loop. Only at some point back in the host code, e.g., when A is printed, data needs to be moved back from device to host memory. It is interesting to note that calls such as the `printf()` call in the compute loop should not trigger any additional data transfers as long as they do not access memory placed on the device.

Using our run-time library we cache allocations. To do so we pass with each device memory request (larger than 4K) the host memory range it corresponds to. We then distinguish four cases: 1) a strictly larger allocation exists, 2) a strictly smaller allocation exists, 3) a partially overlapping allocation exists, and 4) no allocation for this memory range exists. For case 1) we reuse the existing allocation, in case 2) and 3) the old allocation is freed and a new one is created and for case 4) just a new allocation is created. In case of insufficient device memory, cached allocations from older SCoPs are freed.

Unnecessary data-transfers are avoided by keeping track of which device memory regions are up-to-date, by carefully observing host memory activities, and by executing device-to-host transfers lazily. Each time a host-to-device transfer is executed we register in the corresponding allocation the region of memory that has been updated and then immediately protect the host memory region just copied using the `mprotect` system call. In case a later host-to-device data-transfer corresponds to the same memory region we skip this transfer. This is safe for two reasons: First, our kernels do not perform device-memory-writes without a corresponding host-memory write. This means the device memory always corresponds to the host memory, as long as no host code has modified the host memory region. Second, in case the host memory was unexpectedly accessed, we catch the resulting SIGSEGV exception and invalidate all existing cache information. This ensures later host-to-device accesses are not skipped.

Device-to-host memory transfers are performed lazily and only triggered in case the host code actually accesses the corresponding host memory. Whenever a device-to-host memory transfer request is registered in our library the corresponding memory region is again protected with `mprotect` and the data-transfer request is registered (but not yet performed). In case subsequent data-transfers to related memory regions are requested, the existing transfer requests are expanded. Only in case a host-memory access is indicated by a SIGSEGV exception or the corresponding device allocation is evicted, the actual data transfers are issued.

Certain properties are important for this system to be effective: Because handling of page faults is not expensive if it occurs rarely, but has a high-cost if it occurs on each memory access, we only cache transfers and allocations that are sufficiently large ($\geq$ 4KB - the size of a standard page). Furthermore, unrelated segmentation faults may occur in case unrelated data is present on the pages that belong to the memory range we protected. Depending on the `malloc(3)` implementation used, this can be an issue. We observed that the implementation of libc in Linux can place large and small allocations on identical pages. In contrast, jemalloc [22], a memory allocator with focus on "fragmentation avoidence and scalable concurrency" used for example in FreeBSD and Firefox, stores allocations larger than the minimal page size

(a) Mobile system

CPU	i7-4710MQ	GPU	GT730M
Architecture	Haswell	Architecture	Kepler
Cores	4	Shader blocks	2
Frequency	2.5 GHz	Cores / block	192
		Cores (total)	384
Performance		Performance	
- float	320 Gflop/s	- float	552 Gflop/s
- double	160 Gflop/s	- double	24 Gflop/s

(b) Workstation system

CPU	E5-2690	GPU	Titan Black
Architecture	SandyBridge	Architecture	Kepler
Cores	10	Shader blocks	15
Frequency	3.0 GHz	Cores / block	192
		Cores (total)	2,880
Performance		Performance	
- float	480 Gflop/s	- float	5,121 Gflop/s
- double	240 Gflop/s	- double	1,707 Gflop/s

Table 1: Hardware specifications used in evaluation

always separately from smaller objects. As a result, memory that is larger than one page size is guaranteed to not share memory pages with any other allocation.

6. EVALUATION

We evaluate our work on the LLVM nightly test suite (over 50 benchmark suites and full programs), the 30 Polybench 3.2 applications, and SPEC CPU 2006. As hardware we use a workstation with a 10-core Intel Xeon CPU and an NVIDIA Kepler GPU (Table 1b) running Ubuntu 14.04.3 LTS with Linux 3.19.0-43-generic, CUDA tools V7.0.27 and CUDA module 352.68. For our case studies we provide additional performance data on a mobile system with an Intel 4 core i7-4710MQ Haswell CPU and a NVIDIA GT730M mobile GPU (Table 1a) using Ubuntu 15.10 with Linux 4.2.0-30, CUDA tools V7.5.17 and CUDA module 352.63.

6.1 LLVM nightly test suite (LNT)

As first part of our evaluation we analyze the impact of Polly-ACC on a diverse set of applications. The principle objective here is not to impress with speedups. Instead, we want to demonstrate the functionality of our heterogeneous compute optimizer and understand: 1) how many SCoPs can theoretically be mapped to an accelerator and 2) how often do unprofitable mapping choices result in performance regressions.

For our analysis we choose the LLVM nightly test suite (LNT), a large collection of open source applications and benchmark suites used in the LLVM community. It consists of 1,955 files of C code (1,196,209 lines) as well as 648 files of C++ code (251,459 lines). It contains 43 benchmark suites including Dhrystone, Linpack, CoyoteBench, aand SciMark2, 25 full applications including ClamAV, lua, and sqlite as well as various individual tests. LNT also provides support for running the C/C++ benchmarks of SPEC CPU 2006. Overall, this results in 515 individual executables.

We compile the LNT with Polly-ACC and collect statistics (Table 2) on the number of device-mapped SCoPs. Without any heuristics Polly-ACC detects 2,202 device-mappable SCoPs and introduces 2,991 kernels. However, many kernels only perform little computation[4] and can consequently not be mapped profitably to the accelerator. When enabling compile-time profitability heuristics (Section 4.6), 159 SCoPs are mapped and 323 kernels are introduced.

	No Heuristics	With Heuristics
SCoPs	2,202	159
Kernels (0-dim grid)	541	86
Kernels (1-dim grid)	2,167	187
Kernels (2-dim grid)	264	48
Kernels (3-dim grid)	19	2

Table 2: SCoPs and Kernels in LLVM LNT

We also analyze the execution time impact of Polly-ACC in comparison to plain clang+LLVM 3.8 pre (r252936). Running 5 iterations of the benchmark suite with both compilers we see 13 benchmarks that show more than 40% reduction in execution time of which nine show more than 90% reduction. Besides Polybench (Section 6.2) and lbm (Section 6.3), larger speedups are also visible on Shootout/ary3 and Misc/dt. Introducing GPU parallelism results in 4 slowdowns of 2-8% and only one larger regression from 0.33 to 8.0 seconds for the parametric version of Polybench's dynprog. Overall, we see clear speedups with almost zero negative impact.

6.2 Polybench C - 3.2

As first performance benchmark we analyze in detail the Polybench C 3.2 kernels [40], which have already shown runtime benefits in the context of the LLVM test suite (Section 6.1). We compare against icc 15.0.0 20140723, a development version of clang/LLVM 3.8 (r252936), as well as clang in combination with the Polly sequential data-locality optimizations. To obtain fast multi-threaded code we use icc thread-level auto-parallelization (`-parallel`). All codes are compiled with `-O3 -march=native`, single precision floating point operations, and use fixed power-of-two problem sizes scaled to around one second of execution time when compiled with Polly-ACC. All execution times are end-to-end times that include data transfers as well as the time CUDA needs to load the kernel and compile it to device code. We run five samples per test case and report the median.

Our results are illustrated in Figure 3 using icc generated sequential code as baseline. Performance numbers for clang without Polly are not illustrated as they are strictly slower than clang in combination with Polly. We first look briefly at the sequential execution times. In 18 cases icc is faster than clang+Polly whereas in 8 cases clang+Polly produces faster code. The most interesting cases are correlation, covariance and mvt where clang+Polly generated sequential code is more than 3x faster than icc. The poor performance of icc is likely caused by an inefficient loop structure in covariance and correlation which the icc loop optimizer does not optimize well.

We now analyze the performance gains obtained through parallel execution. The icc thread-parallel code shows up to 10x speedup (on a 10 core system) for codes like 2mm, 3mm, correlation, covariance, syr2k and syrk. Polly-ACC outperforms the icc-generated thread-parallel code for 11 out of 30 benchmarks, even though the sequential code generated by clang is in many cases slower than icc generated code. In

[4]average execution is less than 2 seconds

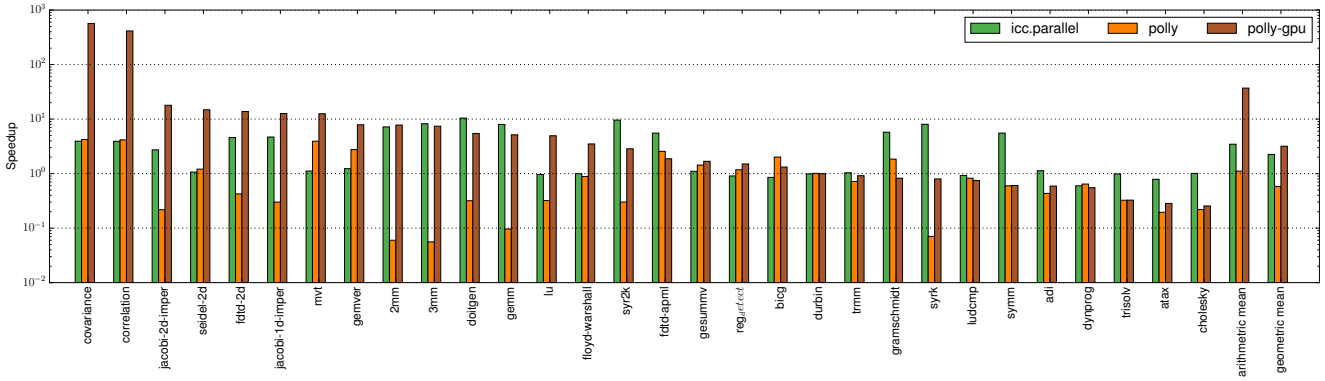


Figure 3: Speedup over `icc -O3` on polybench 3.2 kernels (single)

certain cases, e.g., correlation/covariance, performance has increased by more than 100x (a part of this speedup is due to the slow sequential baseline). In case efficient thread parallel code is generated (the perfect case for icc seem to be gemm-like computations) the difference between GPU and host code is generally smaller. Interesting to note are also the stencil computations jacobi-1d, jacobi-2d, fdtd-2d (but not adi) for which multi-threaded execution seems to be less beneficial, but GPU acceleration indeed is. Finally, it is interesting to note that, when comparing clang+Polly and Polly-ACC, accelerator usage never results in larger performance regressions. Over all benchmarks thread parallel execution results in 2.1x improvement compared to 3.7x using Polly-ACC (geom. mean). Hence, we can conclude that for a wide range of compute kernels automatic accelerator mapping can results in performance improvements even beyond what can be obtained with thread-parallel execution.

6.3 Lattice Boltzmann - SPEC CPU 2006

In the following, we present a case study of 470.lbm. This SPEC benchmark implements the "Lattice Boltzmann Method" (LBM) to compute the behavior of incompressible fluids in three-dimensional space, an important simulation in the field of material science [29]. Its core computation is a time iterated loop (Listing 4) that applies at each timestep two compute functions on a data grid, swaps the grid pointers, and prints grid statistics at every 64th iteration. The two compute functions consist of together 270 lines of code, which scan the data grid multiple times to update the individual grid points. The loop structure itself is rather simple, but the computation performed is non-trivial. First, the working-data is stored in multi-element-type arrays, which contain data elements of different size. Second, the control flow in the loop body is data-dependent, such that it is necessary to introduce a statement region that encapsulates the data-dependent control flow and is scheduled jointly. Modeling such properties requires extensions discussed in Section 4.3. Furthermore, 470.lbm has for two reasons a very irregular memory access pattern: First, the neighbors of a point in a three-dimensional grid are far apart in memory if laid out sequentially. Second, 470.lbm stores all data points of a cell next to each other (array-of-structs), such that subsequent iterations of the data loop do not access neighboring addresses.

GPU execution of 470.lbm is interesting for two reasons:

	Mobile		Workstation	
	Time [m]	Speedup	Time [m]	Speedup
icc	5:19		3:54	
icc -openmp	7:05	-25%	1:41	+130%
clang	5:33	-5%	3:35	+8%
polly GPU	**4:17**	**+24%**	**0:55**	**+325%**

Table 3: 470.lbm run-time performance

First, the larger memory bandwidth commonly available on GPUs allows for fast memory accesses despite the irregular access pattern. Second, the use of predicated instructions in the GPU program simplifies vectorization of data-dependent control flow. However, just by itself each kernel cannot be profitably optimized due to the cost of transferring data between host and accelerator. By using link-time optimization in combination with heavy inlining it may be possible to obtain a single very large SCoP that could be mapped to the device, but both difficult to model pointer swapping in `LBM_swapGrids` as well as IO code in `LBM_showGridStatistics` make this difficult.

Listing 4 470.lbm - Core computational loop

```
for(t = 1; t <= param.nTimeSteps; t++) {
  if (param.simType == CHANNEL)
    handleInOutFlow(*srcGrid);

  performStreamCollide(*srcGrid, *dstGrid);
  swapGrids(&srcGrid, &dstGrid);

  if ((t & 63) == 0)
    showGridStatistics(*srcGrid);
}
```

The key to avoid data-transfer overhead while keeping the GPU mapping strategy local is the lazy data transfer management discussed in Section 5. The only functions that can possibly access kernel data are `handleInOutFlow`, `performStreamCollide`, and `showGridStatistics`. The first two are fully mapped to the GPU whereas the last one is only rarely executed. Hence, data transfers are rarely required and keeping data on the accelerator optimistically is likely to be profitable.

We report performance results (Table 3) for two systems, a standard workstation system as well as a second mobile system, each time using using the SPEC reference data size.

	Kernel 1	Kernel 2
Block sizes	(8, 32)	(8, 32)
PTX size [Bytes/Lines]	9k/216	9k/209
Registers/Thread	40	40

	Kernel 3	Kernel 4
Block sizes	(8, 32)	(8, 32)
PTX Size [Bytes/Lines]	48k/1257	10k/227
Registers/Thread	255	42

Table 4: Cactus ADM statistics

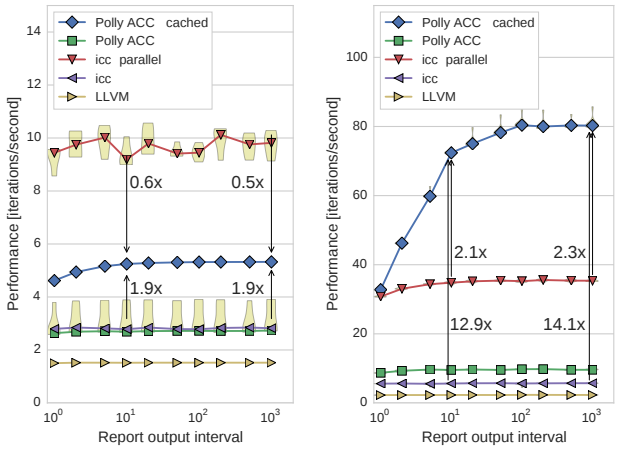
We compare against icc as well as clang and, as lbm comes with OpenMP annotations, also provide OpenMP parallelized numbers for icc. Both icc and clang reach comparable performance during single thread execution. The benefit of thread level parallelism with OpenMP (using the available OpenMP annotations) is less clear. On the mobile system, additional thread level parallelism causes a 25% reduction in overall performance most likely due to cache trashing by competing threads. On the workstation system we see a speedup of 130%, which suggest that the memory system of this platform requires more than one thread to be fully saturated. On both systems, our automatic GPU parallelization is clearly beneficial. On the mobile system we see only 24% performance improvement and on the workstation we see with 325% more than 3x performance improvement over sequential execution and 195 additional percent points over the thread-parallel code. We also looked into the benefits of our memory transfer management system. It reduces the original 6,000 host-to-device transfers and 3,000 device-to-host transfers to only 190 and 142 transfers.

6.4 Cactus ADM - SPEC CPU 2006

Cactus ADM, an application that solves the Einstein evolution equations, is our second full-program case study. It consists of two components, the Cactus problem solving environment [4] as well as benchADM, a kernel representative for many applications in numerical relativity. Together they solve ten coupled non-linear partial differential equations using a staggered-leapfrog method. Especially interesting is that Cactus schedules computations using launch-time parameters to control both the precise computation as well as frequency and content of status reports. As a result, deriving statically a minimal set of data-transfers is impossible even assuming full program knowledge.

The implementation consists of 265 C files (87,060 source lines) for the problem solving environment as well as 5 Fortran files (2,689 source lines) for the computational kernel. The majority of the computation takes place in a single Fortran method. It consists of over 800 lines of comment-free Fortran code, has 9 loops nested to a depth of three, and contains 194 program statements in the body of the largest loop. The full method contains almost 300 array references and slightly more than 350 compute operator calls.

Polly-ACC detects the entire compute function as a single SCoP and maps it automatically to the accelerator. The first challenge to statically reason about this SCoP are the memory accesses, which need to be delinearized (Section 4.3). Delinearization is here additionally complicated by the use of modulo operations in the array index expressions. The second challenge is the size of the code. With only minimal



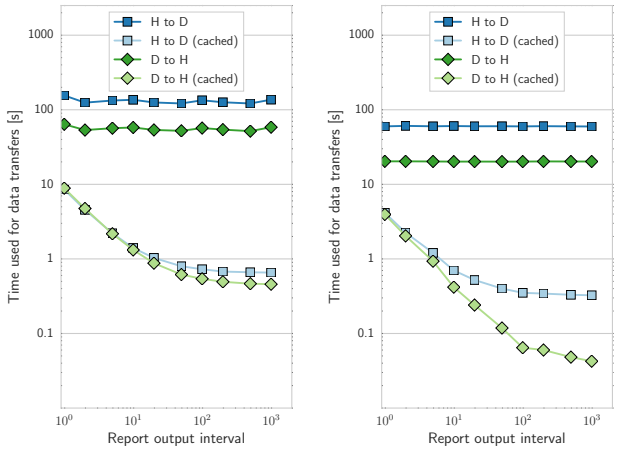
(a) Mobile (b) Workstation

Figure 4: Cactus ADM performance

preprocessing the SCoP that is extracted contains 38 three dimensional arrays, 4 statements, 232 read accesses, 45 write accesses, and uses 79 parameter dimensions. As many operations on integer sets are (double) exponential in the number of dimensions, generating code for the resulting SCoP takes a long time[5] (we aborted after an hour). It might be possible to tune isl for such complex inputs, but in the optimal case we can avoid to reason about so complex spaces. The majority of the parameters have been introduced to model run-time bounds checks. Most of them can be eliminated by enabling global value numbering in the canonicalization phase (Section 4.1). This does not only reduce the number of read accesses to 186, but - more importantly - only leaves 9 parameter dimensions. As a result, Polly-ACC processes the SCoP in less than 30 sec. and generates four different kernels (Table 4). Three with around 200 PTX instructions, while the largest has over 1,200 PTX instructions and uses 255 registers per thread.

We compare Polly-ACC (with/without caching) to icc, icc with automatic parallelization (`-parallel`), and LLVM (using clang and dragonegg as frontends) considering 10 different report output intervals. We run each configuration 10 times and report the median. The violin plots indicating the distribution of the test result of each configuration remain almost invisible for most experiments and show only a small variation for the cached execution of Polly-ACC. Looking at the performance values, we see for both platforms that neither icc nor LLVM performance is affected by a change of the reporting output interval and that icc reaches about twice the performance of LLVM, which can be accounted to better SIMDization. When compiling with `icc -parallel`, we see benefits from multi-threading which result in 3.4x (mobile) and 6.2x speedup (workstation) over the best sequential code. Thread parallel code shows a slight, but visible, performance degradation in case of frequently emitted reports. On the workstation system Polly-ACC without any caching is able to show speedups over sequential icc, despite costly host-device data-transfers at each individual iteration.

[5] isl and Polly provide a compute-out facility to handle these cases gracefully



(a) Mobile (b) Workstation

Figure 5: Cactus ADM data transfer cost

When eliminating this overhead using automatic allocation and data transfer caching as described in Section 5 performance increases notably. Even with frequent reporting, Polly-ACC slightly outperforms icc -parallel. When reporting every 10th iteration, as done with the SPEC reference run parameters, a speedup of 2.1x over thread parallel code can be achieved. This speedup grows to 2.3x when reducing reporting. On the mobile system, Polly-ACC with caching enabled shows 1.9 speedup over sequential code, but only reaches half the performance of thread parallel code due to the low double precision performance of the GPU (Table 1a).

The remaining SPEC CPU applications.

The remaining applications do not show any performance changes, but at least two additional benchmarks, GemsFDTD and bwaves, have hot functions that could benefit from automatic GPU mapping. However, bwaves uses parametric modulos (exp % N) which require some special handling to work in the context of Presburger formulas and gemsFDTD uses difficult array index expressions for which a delinearization approach has been presented, but which has not yet been implemented [26].

6.5 Compilation time

Fast compile times are important. Hence, we compare clang and Polly-ACC (compiled in Release+Asserts mode). The full compilation of all 30 polybench kernels takes 22 seconds with `clang -O3` and 38 seconds ($\approx$ 1.3 seconds / benchmark) with Polly-ACC for the complete flow from parsing over CPU/GPU code generation to linking. This corresponds to about 70% compile time increase. Compiling the full LNT test suite took 27 minutes with clang and required 30 minutes using Polly-ACC, only a 10% increase.

7. RELATED WORK

The steadily growing use of GPU accelerators has motivated the development of a wide set of programming models for heterogeneous compute devices. The most direct approaches are CUDA [39] and OpenCL [44], which — at the cost of program complexity — give precise control over the accelerator. Higher-level C++ libraries such as Thrust [13], C++ AMP [24], Bolt [1] as well as Boost.Compute [2] raise the level of abstraction by providing a C++-STL style programming interface, but still require explicit GPU programming. Directive based languages such as OpenMP for accelerator [14], OpenMPC [33], OpenARC [34], OpenACC [50], HMPP [21], and OpenMP [15] use sequential loops complemented with pragma directives to describe mapping and memory management strategies. Shared memory usage is controlled in OpenACC by developer-provided "cache" annotations. Polly fills an important gap by enabling accelerator mapping without the need for any kind of annotation.

There have been a range of projects for automatically deriving GPU code at the source level. Par4All [7] uses a non-polyhedral approach based on abstract interpretation which enables powerful inter-procedural analyses. Polyhedral compilation techniques have first been used for GPU code generation by Baskaran [11] and have later been improved as part of the R-Stream compiler [35]. An alternative mapping approach that relies on the counting of integer points to tightly fill shared memory caches has been proposed by Baghdadi et. al. [10], but the resulting memory accesses have been shown to be too costly in practice. With CUDA-Chill [42] generating GPU codes based on user provided scripts has been proposed. The state-of-the-art in polyhedral source-to-source compilation is ppcg [49, 46], which provides effective GPU mappings that exploit shared and private memory. The main focus of these tools is the generation of GPU kernel code for often carefully preprocessed code [9]. In the context of PPCG, a PENCIL runtime-library (PRL) is being developed (not published) that allows for kernel code caching, but does not provide memory transfer optimizations. Polly-ACC is the first solution that brings advanced GPU mapping techniques to a large set of programs by not enforcing specific coding styles and by automatically choosing when to perform GPU mapping. Finally, the Polly-ACC run-time library clearly pushes the optimization of GPU kernels beyond single-kernel optimization.

Offloading from within a compiler has been first proposed by GRAPHITE-OpenCL [31] which allowed for the static mapping of parallel loops, but did not considering inter SCoP data reuse. In the context of Polly, Kernelgen [37] proposed a new approach in which it aims to push as much execution as possible on the GPU, using the CPU only for system calls and other program parts not suitable for the GPU. The final executables are shipped with a sophisticated run-time system that supports just-in-time accelerator mapping, parameter specialization and provides a page-locking based run-time system to move data between devices. Damschen et. al. [20] introduce a client-server system to automatically offload compute kernels to a Xeon-Phi system. These approaches are based an early version of Polly (or GRAPHITE), without support for non-affine subregions, modulo expressions, schedule trees or delinearization and are consequently limited in the kind of SCoPs they can detect. Finally, with Hexe [36] a modular data management and kernel offloading system was proposed which does to our understanding not take advantage of polyhedral device mapping strategies. All previous techniques do not use polyhedral modeling to automatically exploit software managed caches and do neither show regression free execution on a large set of benchmarks nor improvements on well-known SPEC kernels.

8. CONCLUSION

We presented with Polly-ACC a new heterogeneous compute compiler for the translation of sequential programs to multi-device executables that transparently take advantage of heterogeneous hardware. Working on a low-level intermediate IR and recovering all necessary information at this level, we showed that even advanced device mapped strategies can be applied — language independent — from within a static compiler. Using a carefully chosen, conservative device mapping strategy we were not only able to compile a wide range of general-purpose codes introducing almost no performance regressions, but also showed notable performance improvements for a range of kernels. These improvements show clearly that automatic accelerator mapping techniques allow certain codes to be accelerated without initial cost and hopefully free developer time to work on algorithmic changes or codes not yet amenable to automatic accelerator mapping.

Acknowledgements.

We thank Swissuniversities for support in the context of PASC initiative (ComPASC) and ARM Inc. for support in the context of Polly Labs. We also thank Albert Cohen and Sven Verdoolaege for their comments as well as Armin Groesslinger and Christian Lengauer for providing compute resources for our evaluation. Finally, we owe many thanks to the whole Polly and isl developer community including Yabin Hu who worked with us on an early prototype of this work in the context of Google Summer of Code.

9. REFERENCES

[1] Bolt C++ template library. http://developer.amd.com/tools-and-sdks/opencl-zone/bolt-c-template-library/. Accessed: 2016-01-21.

[2] boostorg/compute: A C++ GPU computing library for OpenCL. https://github.com/boostorg/compute. Accessed: 2016-01-21.

[3] Oak Ridge Leadership Computing Facility - Summit. https://www.olcf.ornl.gov/summit/. Accessed: 2015-11-16.

[4] G. Allen, Werner Benger, T. Goodale, H.-C. Hege, G. Lanfermann, A. Merzky, T. Radke, E. Seidel, and J. Shalf. The cactus code: a problem solving environment for the grid. In *High-Performance Distributed Computing, 2000. Proceedings. The Ninth International Symposium on*, pages 253–260, 2000.

[5] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 1–11, New York, NY, USA, 1988. ACM.

[6] Mehdi Amini, Fabien Coelho, FranÃğois Irigoin, and Ronan Keryell. Static compilation analysis for host-accelerator communication optimization. In Sanjay Rajopadhye and Michelle Mills Strout, editors, *Languages and Compilers for Parallel Computing*, volume 7146 of *Lecture Notes in Computer Science*, pages 237–251. Springer Berlin Heidelberg, 2013.

[7] Mehdi Amini, Béatrice Creusillet, Stéphanie Even, Ronan Keryell, Onig Goubier, Serge Guelton, Janice Onanian McMahon, François-Xavier Pasquier, Grégoire Péan, and Pierre Villalon. Par4all: From convex array regions to heterogeneous computing. In *IMPACT: Second Int. Workshop on Polyhedral Compilation Techniques HiPEAC 2012*.

[8] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, and Lawrence Rauchwerger. STAPL: An adaptive, generic parallel c++ library. In HenryG. Dietz, editor, *Languages and Compilers for Parallel Computing*, volume 2624 of *Lecture Notes in Comp. Sci.*, pages 193–208. Springer Berlin Heidelberg, 2003.

[9] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F. Donaldson, Jeroen Ketema, Javed Absar, Sven Van Haastregt, Alexey Kravets, Anton Lokhmotov, Robert David, and Elmar Hajiyev. PENCIL: a platform-neutral compute intermediate language for accelerator programming. In *International Conference on Parallel Architectures and Compilation Techniques*, San Francisco, US, 2015.

[10] Soufiane Baghdadi, Armin Größlinger, and Albert Cohen. Putting Automatic Polyhedral Compilation for GPGPU to Work. In *Proceedings of the 15th Workshop on Compilers for Parallel Computers (CPC'10)*, Vienna, Austria, July 2010.

[11] Muthu Manikandan Baskaran, J Ramanujam, and P Sadayappan. Automatic c-to-cuda code generation for affine programs. In *Compiler Construction*. Springer, 2010.

[12] Ulysse Beaugnon, Alexey Kravets, Sven van Haastregt, Riyadh Baghdadi, David Tweed, Javed Absar, and Anton Lokhmotov. Vobla: A vehicle for optimized basic linear algebra. In *Proc. of the 2014 SIGPLAN/SIGBED Conf. on Languages, Compilers and Tools for Embedded Systems*, LCTES '14, pages 115–124, New York, NY, USA, 2014. ACM.

[13] Nathan Bell and Jared Hoberock. Thrust: Productivity-oriented library for CUDA. *Astrophysics Source Code Library*, 1:12014, 2012.

[14] James C Beyer, Eric J Stotzer, Alistair Hart, and Bronis R de Supinski. Openmp for accelerators. In *OpenMP in the Petascale Era*, pages 108–121. Springer, 2011.

[15] OpenMP Architecture Review Board. OpenMP Application Program Interface, Version 4.0. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, July 2013. Accessed: 2015-11-16.

[16] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Notices*, 43(6):101–113, 2008.

[17] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[18] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687. IEEE, 2011.

[19] Nvidia Corporation. CUBLAS LIBRARY - User Guide. http://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf, September 2015. Accessed: 2016-01-21.

[20] Marvin Damschen, Heinrich Riebler, Gavin Vaz, and Christian Plessl. Transparent offloading of computational hotspots from binary code to Xeon Phi. In *Proc. of the 2015 Design, Automation & Test in Europe Conf. & Exh*, DATE '15, pages 1078–1083. EDA Consortium, 2015.

[21] Caps Enterprise. HMPP Directives, HMPP Workbench 3.0. https://www.olcf.ornl.gov/wp-content/uploads/2012/02/HMPPWorkbench-3.0_HMPP_Directives_ReferenceManual.pdf. Accessed: 2015-11-16.

[22] Jason Evans. A scalable concurrent malloc (3) implementation for freebsd. In *Proc. of the BSDCan Conference, Ottawa, Canada*, 2006.

[23] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[24] Kate Gregory and Ade Miller. C++ AMP: accelerated massive parallelism with Microsoft Visual C++. 2014.

[25] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly – performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[26] Tobias Grosser, J Ramanujam, Louis-Noel Pouchet, Ponnuswamy Sadayappan, and Sebastian Pop. Optimistic delinearization of parametrically sized arrays. In *Proc. of the 29th ACM Int. Conf. on Supercomputing*, pages 351–360, 2015.

[27] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. Polyhedral AST generation is more than scanning polyhedra. *ACM Trans. Program. Lang. Syst.*, 37(4):12:1–12:50, July 2015.

[28] Tobias Gysi, Carlos Osuna, Oliver Fuhrer, Mauro Bianco, and Thomas C. Schulthess. STELLA: A domain-specific tool for structured grid methods in weather and climate models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 41:1–41:12, New York, NY, USA, 2015. ACM.

[29] John L Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[30] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on GPU architectures. In *Proci. of the 26th ACM Int. Conf. on Supercomputing*, ICS '12, pages 311–320. ACM, 2012.

[31] A Kravets, A Monakov, and A Belevantsev. Graphite-opencl: Automatic parallelization of some loops in polyhedra representation. *GCC DevelopersâĂŹ Summit, GCC DevelopersâĂŹ Summit*, 2010.

[32] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[33] Seyong Lee and Rudolf Eigenmann. OpenMPC: Extended openmp programming and tuning for GPUs. In *SC*, pages 1–11. IEEE, 2010.

[34] Seyong Lee and Jeffrey S. Vetter. Openarc: Open accelerator research compiler for directive-based, efficient heterogeneous computing. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 115–120, New York, NY, USA, 2014. ACM.

[35] Allen Leung, Nicolas Vasilache, Benoît Meister, Muthu Baskaran, David Wohlford, Cédric Bastoul, and Richard Lethin. A mapping path for multi-gpgpu accelerated computers from a portable high level programming abstraction. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, pages 51–61, New York, NY, USA, 2010. ACM.

[36] Christos Margiolas. A heterogeneous execution engine for llvm. *LLVM Developers Meeting*, 2015.

[37] Dmitry Mikushin, Nikolay Likhogrud, Zheng (Eddy) Zhang, and Christopher Bergstrom. Kernelgen – the design and implementation of a next generation compiler platform for accelerating numerical models on GPUs. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW)*, 2014.

[38] Simon Moll, Johannes Doerfert, and Sebastian Hack. Input space splitting for opencl. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 251–260, 2016.

[39] NVIDIA. CUDA Programming Guide. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, September 2015. Accessed: 2016-01-21.

[40] Louis-Noël Pouchet. Polybench C 3.2. *http://www.cs.ucla.edu/~pouchet/software/polybench/*. Accessed: 2015-11-16.

[41] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[42] Gabe Rudy, MalikMurtaza Khan, Mary Hall, Chun Chen, and Jacqueline Chame. A programming language interface to describe transformations and code generation. In Keith Cooper, John Mellor-Crummey, and Vivek Sarkar, editors, *Languages and Compilers for Parallel Computing*, volume 6548 of *Lecture Notes in Computer Science*, pages 136–150. Springer Berlin Heidelberg, 2011.

[43] J.E. Stone, D. Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66–73, May 2010.

[44] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[45] Sven Verdoolaege. isl: An integer set library for the

polyhedral model. In *Mathematical Software (ICMS'10)*, LNCS 6327, 2010.

[46] Sven Verdoolaege. PENCIL support in pet and PPCG. Technical Report RT-0457, INRIA Paris-Rocquencourt, March 2015.

[47] Sven Verdoolaege. Presburger formulas and polyhedral compilation, 2016.

[48] Sven Verdoolaege and Tobias Grosser. Polyhedral extraction tool. In *Second International Workshop on Polyhedral Compilation Techniques (IMPACTâĂŹ12), Paris, France*, 2012.

[49] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization*, 9(4):54:1–54:23, January 2013.

[50] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. OpenACC – first experiences with real-world applications. In *Euro-Par 2012 Parallel Processing*, pages 859–870. Springer, 2012.