Flynn's reconciliation: Automating the register cache idiom for cross-accelerator programming*

DANIEL THUERCK, NEC Laboratories Europe and TU Darmstadt, Germany
NICOLAS WEBER, NEC Laboratories Europe, Germany
ROBERTO BIFULCO, NEC Laboratories Europe, Germany

A large portion of the recent performance increase in the High Performance Computing (HPC) and Machine Learning (ML) domains is fueled by accelerator cards. Many popular ML frameworks support accelerators by organizing computations as a computational graph over a set of highly optimized, batched general-purpose kernels. While this approach simplifies the kernels' implementation for each individual accelerator, the increasing heterogeneity among accelerator architectures for HPC complicates the creation of portable and extensible libraries of such kernels. Therefore, using a generalization of the CUDA community's warp-register cache programming idiom, we propose a new programming idiom (CoRe) and a virtual architecture model (PIRCH), abstracting over SIMD and SIMT paradigms. We define and automate the mapping process from a single source to PIRCH's intermediate representation and develop backends that issue code for three different architectures: Intel AVX512, NVIDIA GPUs, and NEC SX-Aurora. Code generated by our source-to-source compiler for batched kernels, borG, competes favorably with vendor-tuned libraries and is up to 2x faster than hand-tuned kernels across architectures.

CCS Concepts: • **Software and its engineering → Source code generation**; • **Computer systems organization → Parallel architectures**.

ACM Reference Format:
Daniel Thuerck, Nicolas Weber, and Roberto Bifulco. 2020. Flynn's reconciliation: Automating the register cache idiom for cross-accelerator programming. *ACM Trans. Arch. Code Optim.* 1, 1 (March 2020), 25 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The inability to further scale single processor performance has triggered significant developments in widely parallel processor designs [18]. The TOP500 list of supercomputers shows that today's performance improvements in HPC setups are fueled by two trends: multi-core processors and massively parallel accelerator cards, such as GPUs and vector processors. Unfortunately, upcoming generations of processors are likely to include significant architectural changes to overcome the emerging technological limitations of silicon-based computers [26]. While these innovations help processors achieve higher performance, they usually introduce software stack modifications, including changes to programming models and languages, instruction sets and SDKs. In HPC

*New Paper, not an Extension of a Conference Paper

Author's addresses: Daniel Thuerck, Nicolas Weber and Roberto Bifulco, NEC Laboratories Europe, Kurfürsten-Anlage 36, 69115 Heidelberg, Germany; E-Mails: {daniel.thuerck, nicolas.weber, roberto.bifulco}@neclab.eu
Parts of this work were done while D. Thuerck was with the Artificial Intelligence and Machine Learning Lab at TU Darmstadt.

XXXX-XXXX/2020/3-ART $15.00
https://doi.org/10.1145/1122445.1122456

environments, where software often requires a high degree of specialization and hand-tuning to achieve maximum performance on a given system, the cost of introducing new processors (e.g., GPUs) is therefore significant, since it requires changing the application software.

A second complication comes from the appearance of new diverse applications with quickly changing requirements, such as deep learning (ML) applications. In the ML field, new developments outpace the ability of system researchers to provide efficient software implementations, as also pointed out by a recent report [7]. In fact, ML application frameworks, such as TensorFlow and PyTorch, make it easy to define deep learning computations in Python and then map to highly tuned, vendor-provided accelerator libraries, e.g., cuDNN, cuBLAS or OneDNN. Nonetheless, such frameworks mainly support common computations and hardware, lagging behind cutting edge developments, which therefore often need custom compute kernels for the target accelerators.

At the same time, the paramount *computational graph* abstraction in the ML field helps with simplifying the porting of e.g., deep learning workflows. While the control flow, in form of the graph, is assembled and maintained on the host, its vertices represent kernels that are, due to the nature of training in ML, *batched*. These kernels are executed on large batches of relatively small input problems, e.g., a batched dense matrix multiplication in BERT-like networks. Besides ML, batched kernels are used in e.g., sparse linear algebra [2] or integer optimization [6]. Still, when using operations that are not part of vendor's libraries inside a computational graph on accelerators, users have to implement these high-performance kernels themselves, either manually, using architecture-specific compilers [43] or code synthesizers [52] for different architectures.

In this paper, we address this issue by presenting an abstraction over the Single Instruction Multiple Thread (SIMT) and Single Instruction Multiple Data (SIMD) paradigms used in accelerator architectures to program high-performance batched compute kernels, and a compiler that is capable of mapping them to different target architectures. Avoiding the temptation of defining yet another programming model, our approach is instead targeted at extending existing abstractions that are a potential candidate for high-performance cross-architecture programming. In particular, we focus our attention on the *warp register cache* idiom, as commonly used for, amongst others, batched kernels in the CUDA community [15, 35, 56]. The idiom combines descriptions of computations that are performed by a small set of parallel executors (warp-centric, bulk-synchronous programming) with the explicit use of executors' registers as memory caches (cf. Section 2). While adhering to these two principles limits the expressiveness of CUDA code, the resulting compute kernels can offer a 2-3x speedup over other implementations, e.g., those that use a shared memory approach [48]. Our key insight is that these restrictions also make the implemented compute kernels easier to port to different architectures, while maintaining high performance.

Starting from this observation, we generalize the warp register idiom into the *collective register cache* (CoRe) by treating the number of parallel executors as a compiler-time constant. To abstract over the target architectures, we then define a virtual architecture (PIRCH) which captures the information provided by a compute kernel that adheres to CoRe into an intermediate representation (IR) that subsumes representations for SIMT and SIMD compute models (cf. Section 3).

Building on this basis, we design borG, a source-to-source compiler that compiles batched general-purpose kernels following the CoRe idiom to different architectures (cf. Section 4) via PIRCH's IR, an extension to the vanilla LLVM IR. The current borG implementation supports three different devices: Intels' AVX512 subsystem (SIMD), NVIDIA GPUs (SIMT) and NEC SX-Aurora vector processors (wide SIMD) as well as an LLVM IR backend. Different from program synthesis approaches, borG is a classical, one-shot compiler and does not require a time-consuming search over different variants of a kernel. We show that borG can build compute kernels that compete in performance with hand-optimized ones, and in some cases with vendor-tuned libraries (cf. Section 5) using 7 kernels from various applications and areas.

Class	GPU / SIMT	SIMD	Vector / long SIMD
Representative	NVIDIA Titan V	Intel Xeon 6126 Gold	NEC SX-Aurora 10B
RAM [GB / TB/s]	12 / 0.653	- / 0.119	48 / 1.2
TFLOPs [SP / DP]	14.9 / 7.4	2.6 / 1.8	4 / 2.15
Release	2017	2017	2018
Price point	$3,000	$2,100	$3,000

Table 1. Hardware statistics for our evaluation hardware, representing the three accelerator classes. The price points are averages for the US market in early 2020.

In summary, our contributions are:

- we introduce the virtual architecture PIRCH and a corresponding extension to LLVM IR capturing the semantics of programs that follow the CoRe idiom, and to abstract away the specific properties of target SIMD and SIMT architectures;
- we present borG, a source-to-source compiler for batched OpenCL C programs targeting three different accelerator architectures (AVX512, CUDA and NEC SX-Aurora) and LLVM IR;
- we provide details on specific tuning steps implemented in each of the accelerator backends and discuss their optimized primitives;
- we implement 7 different computational kernels from 3 areas of application using the CoRe idiom and evaluate borG on them against reference kernels and on each of the three supported architectures.

We provide a discussion about related work in Section 6, and give an outlook on future directions and open research questions in Section 7. For the remainder of this paper, we assume that the reader is familiar with OpenCL (or CUDA) and its vocabulary.

2 BACKGROUND

When discussing "accelerators", we usually refer to PCIe cards such as GPUs or Intel's discontinued Xeon Phi series. Here, we include AVX512-enabled CPUs as well. Otherwise, we limit our scope to devices that can still run general-purpose code. This excludes FPGAs and Google's TPUs. While some accelerators, notably NEC's SX-Aurora, can run code as a host, we focus on the use of those accelerators as offload targets for hot spots in the code.

This leaves us with two categories of accelerators: SIMD and SIMT. Roughly speaking, SIMT architectures run many lightweight threads with added primitives for fast communication inside groups of threads and branched or predicated execution. For SIMD architectures, the vector length is a crucial factor that massively influences the way computation is structured. With this in mind, we consider three classes of accelerators used in the HPC community today: CUDA-capable GPUs (SIMT), a short-SIMD extension (AVX512) and "very long SIMD" or *vector* processors (NEC SX-Aurora) – Table 1 lists one example product per class.

2.1 Implementations and Programming Models

SIMD. *Advanced Vector eXtensions* (AVX) are (short) SIMD-oriented instruction set extensions on x86/x64 CPUs. The latest generation – AVX512 – offers 512 bit wide registers Programmers can access these registers through different means: assembly mnemonics or C intrinsics, opaque by auto-vectorization, using languages such as Sierra [33] or through the SPMD-on-SIMD compiler ISPC [43]. All approaches come with their own programming models and abstractions; SIMD-based data parallelism may be combined with task-based parallelism through multi-threading or multi-processing. ISPC, inspired by shader languages from graphics programming, offers a CUDA-like execution model, depending on the processor architecture, several cores may share caches.

GPUs / SIMT. Graphics Processing Units (GPUs) follow a data-parallel SIMT execution model. NVIDIA's GPUs are mainly programmed in CUDA, a C++ extension that expresses computations from the viewpoint of a single *thread* inside a *(thread) block* in a *compute grid*. Threads within a block may synchronize through a piece of on-chip shared memory, which is faster than the device-wide global memory. The scheduling of individual blocks to the actual execution units, streaming multiprocessors (SMs), is performed in hardware. Multiple blocks may be mapped to the same SM if enough registers and shared memory remain; in addition, over-provisioning of warps to SMs is used to hide latencies. To achieve their full potential, GPUs require coalesced memory accesses, thousands of independent tasks and a high FLOP/byte ratio.

With CUDA and its libraries, NVIDIA offers a full stack of software and development environment. In recent years, the HPC community helped supporting a wide range of libraries and applications for and with GPUs. In the deep learning community, GPUs are currently the de facto standard device. With CUDA being the dominant programming model on the GPU market, there is also OpenCL [47] standard for programming CPUs and GPUs, among others, in a similar SIMT-Manner.

Vector / long SIMD. NEC's SX-Aurora continues a concept from early times of HPC: vector processing. The current PCIe-variant offers 8 compute cores, each with a scalar and a vector unit. Vector register are 16,384 bit wide for 256 `double` or 512 `float` elements. Each core offers 256 vector registers, of them 64 are exposed to the developer. There are different functional units per core, e.g., for FMA, memory accesses or cross-lane operations. The hardware supports limited out-of-order processing with separate pipelining in all functional units. NEC's NCC compiler auto-vectorizes standard C/C++ and FORTRAN code and supports multithreading through OpenMP. There is an experimental LLVM compiler backend for C intrinsics (LLVM-VE), which allows to program the SX-Aurora similar to AVX512. Further support for vectors of arbitrary length in LLVM IR (LLVM-VE-VL) is under active development, but still in an early stage. Aurora's unique selling point is its very high memory bandwidth of 1.20 TB/s, combined with a 16MB *last-level-cache* shared by all cores. This results in the highest byte/FLOP ratio of the accelerators presented in this section.

2.2 The Warp Register Cache Idiom

CUDA's structuring element of "thread blocks", does not map 1:1 to GPU hardware units. Instead, blocks are decomposed by the GPU's scheduler into *warp*s, sets of 32 contiguous threads. Starting with the Kepler architecture, NVIDIA has introduced warp-level primitives that expose (sub-)warp synchronization and data exchange between threads in a warp to the programmer. The central primitive is the `_shfl_sync`[1] function that enables threads within the same warp to access other active threads' registers. Besides `_shfl_sync`, there are voting functions like `_ballot_sync` and predicates such as `_any_sync` [42]. However, all of these primitives can be built on top of the `_shfl_sync` primitive. For the remainder of this discussion, we thus limit ourselves to this central primitive and refer to it just as "shuffle". Before the Volta architecture, threads within a warp operated in lockstep, all sharing one program counter and stack. With "Thread Independent Scheduling", each thread stores its own program counter and stack, enabling synchronization and shuffles on the sub-warp level. Most notably, the current implementation of the `_shfl_sync` primitive accepts a mask of threads within the warp that participate in this shuffle. As a side effect, these warp primitives that share the suffix `_sync` also serve as a synchronizing barrier for all threads marked in the mask argument.

The availability of warp-level primitives led to the *warp register cache* idiom, a combination of exploiting the synchronization of threads within a warp and the large register file as a user-managed cache to limit global or shared memory access. This idiom mirrors the established strategy of dividing work up over threads and synchronizing them in a block, sharing data through shared

[1]We use the functional interface rather than the object-oriented functions of NVIDIA's cooperative groups API.

memory on the warp level. Due to the higher bandwidth of registers, strictly following this idiom can lead to high speedups. The prime use case for this idiom are batched computations on small portions of data held in registers, e.g., batched GEMM or batched matrix factorizations [5, 48]. In those examples, each thread inside a warp loads, e.g., a row of the warps' matrix and uses shuffle whenever it needs access to the row stored by another thread. Since CUDA's language does not expose registers to the language level, developers commonly try to adhere to a few restrictions in order to "motivate" the compiler into mapping variables uniquely to registers:

(1) Do not use thread-private arrays. Such arrays trigger allocations in global memory and there is no guarantee that NVCC caches accesses in registers. Instead, express every single scalar in an array as a separate variable.
(2) When dynamic selection is necessary, partition the data in a way such that shuffles may be used. If, e.g., each thread holds a row of the matrix (with each column as a single variable), we may use a shuffle to fetch data from a row determined at runtime. This is a GPU-specific hardware feature.
(3) Balance the workload between warps – since the number of blocks scheduled onto an SM is limited, this could result in long-running warps that hog SMs while leaving resources unused.

In general, the first restriction is the most important one and also the one complicating manual implementations the most. It leads to bloated kernel code (see e.g., the source code for [50]) that requires some kind of code generation in order to be maintainable. Although research into the use of SIMD registers for register caching on CPUs [45] exists, its embedding into a SIMT context with fast shuffle instructions has not been investigated as a means to implement performance-portable kernels *across architectures*. In the remainder of this paper, we propose exactly that. Simply put, we answer the question: *How (and if) can code following the warp register cache idiom run fast on other accelerators than just GPUs?*

3 PIRCH - A COMMON VIRTUAL ACCELERATOR ARCHITECTURE

The warp register idiom binds the programmer to warps with 32 threads, which is clearly specific to the NVIDIA's architecture, limiting the portability of the idiom to different architectures. We therefore define the *collective register cache* (CoRe) idiom by simply removing this limitation, and instead allowing a configurable size for the number of threads. For instance, in code processing $3x3$ matrices, one would select 3 threads and have them share the input matrix collectively. With the idiom defined, we now need a way to map it to different models, i.e., SIMD and SIMT.

While SIMT and SIMD share similar concepts, their programming models are different. In the SIMT-implementation CUDA, we implement algorithms from the perspective of a single thread and the warp-based execution model provides implicit vectorization. For SIMD systems, both assembly and C with intrinsics use fixed-width vectors as a data type (i.e. explicit vectorization). Thus, their respective intermediate representations are also designed in different contexts. A prime example is `gpucc` [55] where host (x86, incl. SIMD) and device (GPU) code each have their own respective IR.

As a common abstraction over the two models within the context of batching, we propose PIRCH (the *Partitioned Infinite Register maCHine*), a theoretical architecture that offers a well-defined representation from which both SIMD and SIMT codes may be extracted. In a nutshell, PIRCH can be described as SIMT-on-SIMD (the distinction to ISPC's [43] SPMD-on-SIMD is outlined in Section 6).

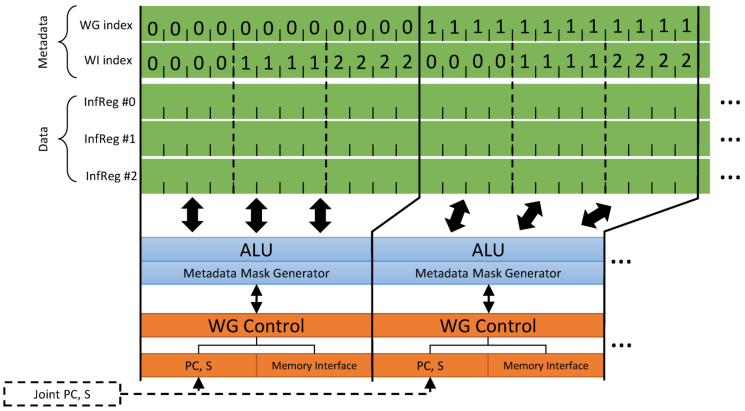


Fig. 1. Theoretical PIRCH architecture: Multiple vector registers of infinite width (InfRegs) are partitioned into equal-sized segments, each holding data for one work group (WG). Within a WG, InfReg lanes are further partitioned to into work items' (WI) private data. The partitioning is encoded in the form of WG and WI indices as "metadata" in separate InfRegs. Each partition has its own vector ALU and shuffle unit for cross-lane interactions. Each WG maintains its own execution state in form of a program counter and stack, executing independently from other WGs. WIs in a WG execute in lockstep. A solid black line marks the InfReg parts and units of one WG, the dashed black line marks the data for one WI. PIRCH supports SIMT-like execution of divergent code by generating masks from the metadata. For mostly convergent code, several WGs may be steered by a joint program counter. Owing to the infinite register space, all batch items can execute concurrently.

3.1 Architecture

We design PIRCH with batched kernels in mind. Assuming that batch items do not interact, all batch items may be, theoretically, run concurrently. As the batch size is not known until runtime, the amount of data stored in registers cannot be bound a priori.

Description. Fig. 1 visualizes the theoretical PIRCH architecture, which can best be described as a "SIMT-on-SIMD" concept. While InfRegs and vector ALUs resemble a SIMD model, the partitioning into WGs and WIs follows the SIMT paradigm. Moreover, metadata can be used in computations to generate vector masks, implementing SIMT-like divergent control flow between WIs within the same WG. A per-WG shuffle unit implements the shuffle primitive similar to CUDA's warps. All WGs execute independently from others with their own control units. When the amount of divergent execution between WGs is low, they may optionally use a common program counter. Lastly, these control units share an infinite amount of main memory from and to which they can issue WG-sized vector loads and stores.

In the execution model, we deviate from the typical fixed-size warp scheduling model: We execute the whole WG, independent of its size, as if it was one warp, in lockstep without explicit synchronization and we use `shuffle` instructions (without support for sub-warp masks) to exchange data between WIs. Therefore, we set aside the term "wavefront" for the rest of this paper, instead using the now equivalent "WG" and denote the compile time-constant wavefront size as `WG_SIZE`.

Mapping Kernels. We continue the discussion with a concept how batched kernels are mapped to PIRCH. In order to be able to follow the CoRe idiom, we propose to use OpenCL C extended by a `shuffle` instruction as input language. PIRCH's mapping of variables to InfRegs follows the rules listed in Fig. 2. It draws on the experience from current SIMD systems where interactions between different lanes are costly, but interactions between two vectors within the same lanes are cheap. Notably, there is only a unique association from program variables to InfRegs, but not vice

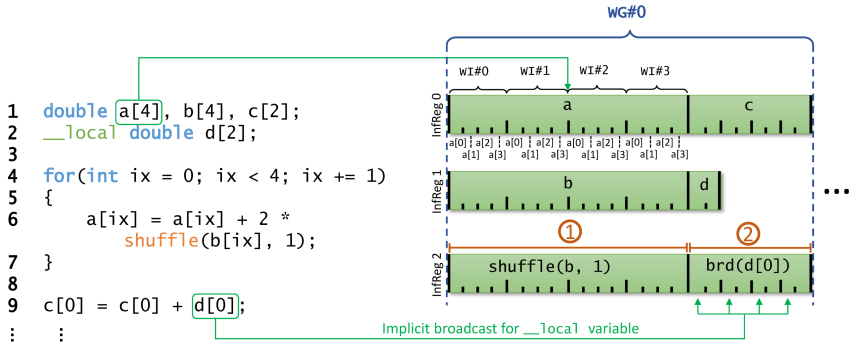


Fig. 2. Mapping an OpenCL program fragment to PIRCH with its WG-centered execution model follows 5 rules: (1) each private variable (arrays) is concatenated WG-wise into a single InfReg whereas (2) local variables are only held once. (3) Variables that interact must be aligned to the same lanes, leading to a partitioning into interaction sets (1 and 2). (4) Variables with non-intersecting liveness ranges and compatible type may be assigned to the same InfReg and lanes and, (5) reordering InfReg lanes within an WG triggers implicit broadcasts whereas cross-WI data exchange requires a `shuffle` instruction. Lastly, to compute multiple batch items in a single run, the area marked as "WG#0" may be repeated horizontally.

versa. While in theory, every lane in an InfReg may have different scalar types, we limit this to one common scalar type per InfReg in order to facilitate the mapping to actual hardware.

3.2 InfReg-IR

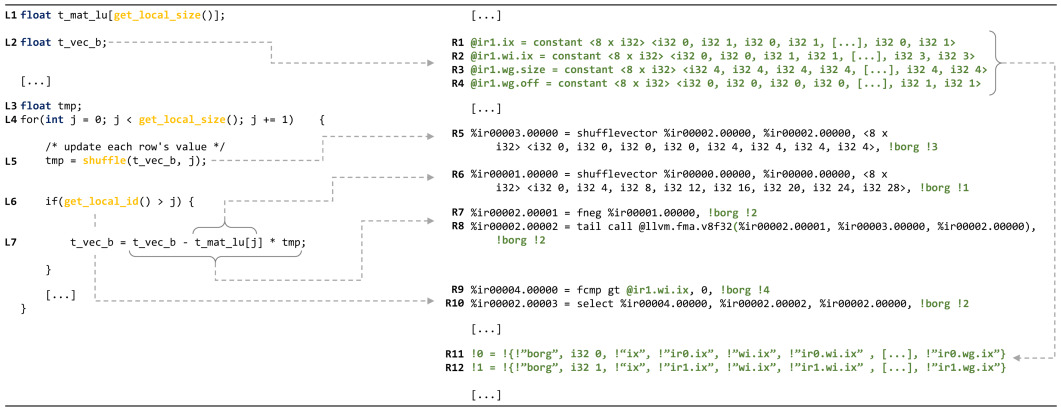
PIRCH subsumes both SIMT and SIMD, but mapping to a target architecture requires specializing for either at some point. When encoded properly, the PIRCH model itself can be seen as an IR in a cross-accelerator toolchain for batched kernels. This section describes such an encoding in form of a textual IR. While many typed IRs can be used to encode programs using PIRCH, we decided to extend the broadly adopted LLVM IR [30]. LLVM IR is a single static assignment (SSA) based, typed low-level representation. In favor of brevity, we only describe additions to this established IR and refer to [30] for an introduction.

As all WGs in PIRCH are independently scheduled units and only depend on their metadata, it is sufficient to represent a single WG's computation and parameterize it by metadata. As LLVM offers a flexible system for the definition of custom metadata, we employ to represent PIRCH's metadata in addition to the computation. Ex. 1 outlines our principles along kernel fragment.

In line with the mapping principles from Fig. 2, a variable such as `t_vec_b` in line L2, Ex. 1 maps to an InfReg, triggering the creation of constant vectors in the IR (R1-R4) for its associated InfReg #1 (`ir1`). These constant vectors may also be references in computations (R9). When multiple WGs share one program counter, vectors are concatenated such that that the IR then represents the computation of these WGs. In this case, only the first WG in each kernel receives its global index as an argument and we add local constant WG offset (R4) onto it to determine the global indices of all participating WGs. In Ex. 1, we set `WG_SIZE` to 4 and use the share a program counter between pairs of WGs. For scalar variables such as `t_vec_b`, this results in a size for the InfReg chunk of $\texttt{WG\_SIZE} \times 1 \times 2 = 8$.

These vectors are collected in LLVM metadata nodes (R11, R12), each representing the metadata for one InfReg, associating keywords `ix, wi.ix, wg.size, wg.off, wg.ix` with the respective constant vectors or, in case of `wi.ix`, the runtime-computed values. Any instruction that creates an SSA-value representing an InfReg's state is annotated by the respective metadata nodes. The scalar types of InfRegs are implied through the types of LLVM's operations, e.g., `f32`.

Example 1 IR translation for a part of the TRSV (triangular solve) kernel according to the description in Sec. 3.2. For InfReg #X, `irX.wg.size` corresponds to OpenCL's local size, `irX.wg.off` to a WG's offset when packing multiple WGs, `irX.wi.ix` to a WI's local id and, finally, `irX.ix` to the index of an array variable (as, e.g., private array variable `a[4]` in Fig. 2). **Lx** and **Rx** are line numbers and all additions to LLVM IR are colored in **green**.



Mapping the so-modified LLVM IR to SIMD processors is straightforward. However, for SIMT systems, we have to enforce *location constraints*: Two lanes of vector types may interact only if they share the `wi.ix`. The only notable exception here is the LLVM IR instruction `shufflevector` to which PIRCH's `shuffle` maps (R6). By evaluating the metadata the in the lanes of metadata node `!2` corresponding to InfReg #2, we reconstruct a shuffle pattern. R6's shuffle pattern corresponds to a broadcast from WI 1 to all other WIs.

In the current implementation of our compiler borG, we use only a subset of LLVM instructions: arithmetic operators (`fadd`, `fsub`, ...) and the comparison operator `fcmp` including integral forms; FMA-intrinsics; `load` and `store` and gather/scatter intrinsics where applicable; `select` for masked computations, branching instructions and `phi` nodes. Pointers are dereferenced with `getelementptr`. By following the principle of appending InfReg metadata nodes in LLVM IR, the remaining parts of LLVM IR may be added when needed. Alternatively, the modified LLVM IR may be implemented as a dialect of MLIR [31], a multilevel framework for defining and transforming IRs. We refer to this modified LLVM IR as "Infreg-IR".

4 BORG

Developing even rather short and simple kernel codes can require tremendous effort when multiple target architectures and programming models have to be considered. In this section, we introduce borG, a source-to-source compiler that automatizes the mapping from batched kernels written OpenCL C to multiple accelerator backends via PIRCH IR. borG (where 'G' stands for Generator and the name is reminiscent of a civilization in a popular science fiction franchise that emphasizes its *collective* – as in CoRe) can generate code for CUDA, SX-Aurora and AVX512 as well as emit LLVM-IR. Using borG extends device support for systems relying on batched kernels immediately while saving developers a considerable amount of time.

In order to map to actual hardware, implementing the PIRCH model with its infinite registers is not possible and assuming a maximum batch size is undesirable. However, this is where the independence and thus, concurrence, of batch items comes into play: They may be processed in arbitrary order. borG exploits this to generate code that deals with a limited set of batch items and then uses platform capabilities, e.g., OpenMP loops, to repeatedly call the generated

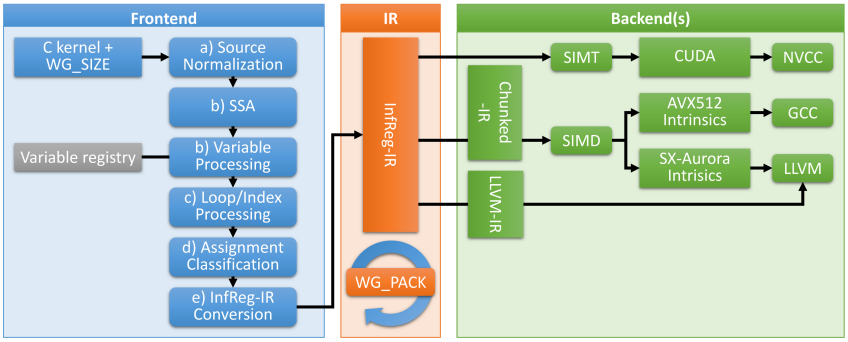


Fig. 3. System overview of the borG source-to-source compiler: the system consumes a CUDA-like kernel written following the warp register cache idiom (Section 2.2) and maps it onto InfReg-IR, based on the virtual architecture PIRCH. From there, either C + intrinsics or LLVM-IR may be generated, effectively generating code for three popular accelerator architectures.

Grammar 1 Grammar of the supported C subset by borG's frontend. The start symbol is f and we use the following descriptive sets: $\mathcal{T}$ for basic types, $\mathcal{B}$ for built-in functions and $\mathcal{O}$ for operators in the C standard (adapted from [32]).

s	::=		*Statements:*
		;	skip
	|	$\{\hat{s}\}$	scope
	|	e;	expression statement
	|	t `id`;	declaration
	|	`__local` t `id`;	local variable declaration
	|	$e_1 = e_2$;	assignment
	|	e_1 `op=` e_2; $\mathtt{op} \in \mathcal{O}$	(shorthand)
	|	`if` s_1 `else` s_2	if-else
	|	`while`(e) s	while
	|	`for`$(s_1; s_2; s_3)$ s_4	for
f	::=		*Function:*
		ϵ	empty
	|	f_1 f_2	sequence
	|	`__kernel` t `id`$(\overline{t\ \mathtt{id}})$ $\{\hat{s}\}$	function
e	::=		*Expressions:*
		c_α	constant of type $\alpha \in \mathcal{A}$
	|	`id`	variable
	|	`id`$[e]$	array indexing
	|	`p`$[e]$	pointer dereferencing
	|	e_1 `op` e_2; $\mathtt{op} \in \mathcal{O}$	binary operation
	|	$b(\overline{e}) \in \mathcal{B}$	call to build-in function
	|	`shuffle`$(\overline{e})$	WG shuffle
$\hat{s}$	::=		*Scoped Statements:*
		s	statement
	|	$\hat{s}$ s	sequence
t	::=		*Type:*
		$\alpha \in \mathcal{T}$	base

kernel with changing metadata. borG's frontend consumes a subset of OpenCL C as defined by Grammar 1. We currently support all standard primitive C types and the OpenCL C functions $\mathcal{B} = \{$`get_local_id`, `get_group_id`, `get_local_size`, `sqrt`, `max`, `min`, `abs`, `rcp`$\}$.

Arrays are passed as pointers into borG kernels. These pointers can be dereferenced and indexed by constant or runtime values, but further pointer arithmetic has not been implemented yet. Moreover, borG currently does not support structs. These are not limitations of the concept. Rather, we often see that warp-cache implementations for GPUs favor the "Structure of Arrays" memory layout for performance reasons [54] which is covered by our chosen subset. As our evaluation shows, many interesting cases are covered by this subset of OpenCL C. However, adding support for both features would be possible. Structs may be either packed into the same InfReg with its members in adjacent lanes or distributed over multiple InfRegs with one InfReg per struct member. Pointers can be added as a datatype in InfRegs. On SIMD platforms, dereferencing these pointers without further analysis then leads to gather and scatter instructions.

Packing kernels. PIRCH offers the option to steer multiple WG's executions from a single PC and stack when the expected divergence is low. In the finite setting of borG, this translates to kernels that process multiple WGs in one call. This feature is especially useful for small batch items, e.g., 3×3 matrices, that would otherwise not fill the (SIMD) registers. In the remainder of this paper, we use the factor `WG_PACK` to express how many WGs are processed in one kernel call.

Example 2 Example of code transformations undergone in borG's frontend up to the InfReg-IR conversion. For details, please refer to Section 4.1.

(a)

```
float a[4], b[4], c[4];
for(int ix = 0; ix < 4; ++ix)
{
    a[ix] = a[ix] + 2 * (b[4 - ix - 1] + shuffle(c[ix], s));
    if(get_local_id() > s)
        b[ix / 2] += c[ix / 2];
}
```

(b)

```
float a[4], b[4], c[4];
for(int ix = 0; ix < 4; ++ix)
{
    a_0[ix] = b[4 - ix - 1]
    a_1[ix] = shuffle(c[ix], s);
    a_2[ix] = a_0[ix] + a_1[ix]
    a[ix] = __fma(2, a_2[ix], a[ix]);
    if_0[ix / 2] = static_resolve(get_local_id() > s);
    b[ix / 2] = if_0[ix / 2] ? (b[ix / 2] + c[ix / 2]) : b[ix / 2];
}
```

(c)

```
float a[4], b[4], c[4];

/* ix = 0 */
a_0[0] = b[3]
a_1[0] = shuffle(c[0], s);
a_2[0] = a_0[0] + a_1[0]
a[0] = __fma(2, a_2[0], a[0]);
if_0[0] = static_resolve(get_local_id() > s);
b[0] = if_0[0] ? (b[0] + c[0]) : b[0];

/* ix = 1, 2 */
for(int ix = 2; ix < 3; ++ix)
{
    a_0[ix] = b[4 - ix - 1]
    a_1[ix] = shuffle(c[ix], s);
    a_2[ix] = a_0[ix] + a_1[ix]
    a[ix] = __fma(2, a_2[ix], a[ix]);
    if_0[ix / 2] = static_resolve(
        get_local_id() > s);
    b[ix / 2] = if_0[ix / 2] ? (b[ix / 2] +
        c[ix / 2]) : b[ix / 2];
}
/* ix = 3 */
[...]
```

Furthermore, we refer to these WGs as one "pack". Within the pack, a WG's global index is then the sum of the index of the pack plus the WG's local offset within the pack. `WG_PACK` is exposed to the user as a tunable parameter, compromising between register pressure within a kernel and overhead of launching kernels.

Beyond batched kernels. PIRCH and borG were designed with batched kernels in mind, assuming independence of all batch items. In addition to that, our current implementation strictly follows the PIRCH model, not offering any means of communication between different WGs. A generalization to general-purpose code would require two extensions: First, it would mandate adding capabilities for communication and synchronization between WGs. This could be done by e.g., adding support for atomic instructions, mutexes or other forms of resource sharing in borG. Second, borG's language should be extended to support structs and functions to cover a larger subset of OpenCL C and enable recursion. The required changes are conceptually minor, but may complicate the mapping from variables in the front-end to InfRegs and make the mapping opaque for developers. The current restrictions on borG thus are not conceptual, but should enable users to maintain a high level of control on their code while still covering a large-enough set of important batched kernels.

4.1 Frontend

borG's frontend consumes the kernel code using PyCParser [11] and a WG configuration (`WG_SIZE`, `WG_PACK`) and generates a kernel description in InfReg-IR. Given an OpenCL C-file, the frontend parses all functions with a __kernel annotation. As depicted in Fig. 3, borG's frontend executes a

Example 3 Static pivoting LDU-Kernel implementation for borG.

```
__kernel
void ldus_generic(double * mat_a) {
    double t_mat_a[get_local_size()];
    double t_piv_row[get_local_size()];

    /* load input matrix A */
    for(int ix = 0; ix < get_local_size(); ix += 1)
        t_mat_a[ix] = mat_a[get_group_id() * (get_local_size() * get_local_size()) +
            get_local_id() * get_local_size() + ix];

    /* decompose matrix (with fused Schur complement!) */
    for(int s = 0; s < get_local_size(); s += 1) {
        /* copy pivot row to PE-local storage */
        for(int ix = s; ix < get_local_size(); ix += 1)
            t_piv_row[ix] = shuffle(t_mat_a[ix], s);

            /* scale pivot column */
            if(get_local_id() > s)
                t_mat_a[s] = t_mat_a[s] / t_piv_row[s];

            /* scale pivot row and perform Schur downdate */
            for(int ix = s + 1; ix < get_local_size(); ix += 1)
                if(get_local_id() >= s)
                    t_mat_a[ix] = (get_local_id() == s ? (t_mat_a[ix] / t_piv_row[s]) : (t_mat_a[ix] -
                        t_mat_a[s] * t_piv_row[ix]));
    }

    /* save decomposed matrix A = LDU */
    for(int ix = 0; ix < get_local_size(); ix += 1)
        mat_a[get_group_id() * (get_local_size() * get_local_size()) + get_local_id() *
            get_local_size() + ix] = t_mat_a[ix];
}
```

sequence of code transformations on the *abstract syntax tree* (AST). Ex. 2 shows the different steps of the code transformations where listing (a) represents the input code.

Source Normalization. In order to facilitate source code processing downstream, we promote all scalar variables to array variables of size 1 and add an index `[0]` to all accesses. Furthermore, we expand all combined assignments (e.g., `+=`) into their full expressions and move the conditions in ternary operations into if-blocks. Lastly, we collect all variables with their respective (scalar) types and dimensions and classify them as *explicit* variables.

SSA decomposition. Similar to most compilers, we perform a SSA decomposition on each assignment in the code, leaving only elementary operations on the right-hand side; all resulting assignments are placed in a single code block. For each additional assignment, we instance a copy of the variable on the left hand side of the original assignment that is local to the generated block. We mark those local variables as *implicit*. Due to PIRCH's requirements on cross-lane operations and the restriction to compile-time array indices, we extend the SSA decomposition: Array indices for the variables on the left and right hand side are compared symbolically using SymPy [27]. If we encounter symbolically different indices, this results in an additional assignment for variable reordering being generated. Similarly, any calls to `shuffle`, another type of cross-lane operations in the InfReg down the road, imply an additional assignment. For each `if`- and `else` block we create boolean array-valued variables that are used as masks. All masks – one instance per compatible IR-format of variables that appear in the conditional's body – are then propagated to assignments as conditionals in ternary operations. Lastly, whenever we encounter a matching pattern in the AST, we insert FMA-type operations automatically. After this step, the code of our running example has been transformed into listing (b) in Ex. 2.

Variable Processing. We extract all implicit and explicit variables and assign unique global IDs. Then, we gather all variables that interact with each other, i.e. appear in the same assignment, into

a matrix. By interpreting this matrix as a graph and extracting connected components, we find interacting sets of variables. Two variables from different sets may be packed into the same lanes in an InfReg. Variables inside the same set must be packed into different InfRegs, but aligned to the same lanes. We pack all variables into InfRegs in a greedy manner, packing the independent sets horizontally into each InfReg. Lastly, we annotate all lanes with their respective WI ids.

Loop and Index Processing. Kernels following CoRe can contain WI-local arrays, e.g. `t_mat_a` in Ex. 3. We map each lane of an array to registers, hence their indices must be resolved at compile time. When arrays are accesses in the body of a for-loop, we first classify that loop into one of three categories: unroll, vectorize and runtime. If the loop index depends on a runtime variable, all instructions inside the loop's body are predicated with a mask that deactivates all lanes of WIs that have already left the loop. At runtime, the loop is executed until all WIs evaluate the loop condition to false. In case of nested for-loops, all outer loops are unrolled. Otherwise, we evaluate their indices in borG and propagate the values to array accesses in the loop body. Here, we iterate over the loop's counter variable sequentially and determine index sets that could be processed within a single instruction ("vectorize"). As an example, compare lines 6 and 8 in Ex. 2's listing b): for `ix = [0, 1, 2, 3]`, line 6 leads to indices `[0, 1, 2, 3]` on the left hand side, so all lanes used in this loops may be processed in a single instruction. Line 8 leads to `[0, 0, 1, 1]`. To avoid write conflicts, we split the loop progressively into index sets `[[0], [1, 2], [3]]`. As long as there are no dependencies on earlier iterations of this loop, the split index set can be optimed by reordering iterations, leading to `[[0, 2], [1, 3]]`. Without this type of analysis, SIMD units would be dramatically underutilized. The resulting code after splitting is given in listing (c) of Ex. 2. In doubt, e.g., when an array index depends on the loop variable of a runtime-for, we execute a loop sequentially, i.e. with one array member per WI processed per iteration instead of the whole array in a single iteration. When array indices cannot be determined at compile time, the whole array is temporarily stored in memory and accesses for the current instruction are mapped to gather/scatter instructions, leading to drastic performance losses.

In an optional frontend pass, we analyze index expressions for memory accesses using SymPy [27]. We try to express the expression as a polynomial on a work item's local and global id. Based on that analysis, we determine whether an access is contiguous and extract an scalar offset for contiguous loads and stores.

Additionally, we offer a *streaming* option to limit register pressure. With this enabled, data is always loaded from memory upon access and written back after each update. For that, we store store the results of the last symbolic memory access analysis and turn it into an InfReg $\leftrightarrow$ memory mapping.

InfReg-IR Conversion. Remaining loops in the transformed C code are unrolled into separate IR nodes at this stage, resulting in a doubly-linked list of nodes, that are then converted into InfReg-IR. As last step, we pack multiple WG instances into the InfRegs as described before, which then automatically expands all IR instructions to processing multiple WGs. The resulting IR is then handed off to one of our backends.

4.2 Backends

4.2.1 SIMT Backend. The SIMT backend generates CUDA-C++. To match the hardware warp size, we restrict `WG_SIZE` to ≤ 32 and leverage the GPU's hardware shuffle functions in lockstep execution. The crucial point in this backend is the decomposition of InfReg-IR computations into *execution groups*, i.e. groups of CUDA threads inside a warp that execute the same code . We partition the WIs into active sets according to the InfReg lanes that they access or, if specified, mask. Each groups' code is predicated using a condition such as `((1 « wi_ix) & 0x02)` where `wi_ix` corresponds to the WI's index in InfReg metadata. A groups' code is serialized in case of diverging accesses, as in

the case for `b` in `a[0] = a[0] / b[wi_ix]`. Here, `b`'s index varies per WI, thus every WI is its own execution group. Consequently, all WIs are serialized, resulting in code resembling the following extract (for `WG_SIZE` = 2):

```
if(wi_ix == 0) a_0 = a_0 / b_0;
if(wi_ix == 1) a_0 = a_0 / b_1;
```

If `WG_PACK` is set to a value > 1, the whole WG code is replicated and references to the work group index are updated by the pack's local WG offsets. Reordering, shuffles and ternary operations all map natively to the scalar CUDA language. The shuffle and reordering patterns are inferred from the InfReg-IR by analyzing the lane permutation vectors. While WI-local variables are mapped to registers, WG-wide shared variables are put into the GPU's local memory. Furthermore, we use the cooperative groups-API offered by NVIDIA to handle `WG_SIZES` ≤ 32. We launch CUDA blocks of size 32 potentially processing multiple WGs with packing. On NVIDIA GPUs, Streaming Multiprocessors (SM) can only hold a limited number of both blocks and warps in their warp schedulers to perform hardware multithreading. Due to our 1:1 mapping between blocks and warps, our code is limited by the minimum of both constraints, where in practice the number of held blocks per SM is the lower of both. However, since we map InfReg lanes to registers, register pressure turns out to be the limiting factor for occupancy, not the limit on schedulable warps. This was confirmed through experiments where larger, multi-warp CUDA blocks, did not exhibit a noticeable performance benefit.

Memory accesses and optimizations. In Ex. 3, line 8, the WI-private arrays `t_mat_a` contain a matrix in row-wise format. Iterating over the WI's ids and then `ix` leads to contiguous indices into the InfReg for SIMD accesses. On GPUs, this causes all work items to load one member of its `t_mat_a` per instruction, leading to non-coalesced loads with a stride of `WG_SIZE` doubles. Ideal register-cache kernels have a high ratio of data reuse, hence fetching the data once in a suboptimal fashion has negligible impact. In fact, e.g., loading a matrix in col-major format when it is held as row-major in memory leads to strided resp. interleaved memory accesses. Optimizing these accesses for CPUs with SIMD units [1, 3] or DSPs with indirect vector register files that allows on-the-fly reordering of vector registers [40] is covered by related work. Thus, one strategy could be to optimize the CoRe kernel code for the GPU's memory access layout and minimize the resulting strided memory accesses on the CPU in order to avoid changing data layouts of the application. We leave this integration for future work.

Except for dead code elimination and memory analysis, borG's backend does currently not perform any optimizations on the IR or native code level. Since these optimizations can be very architecture-dependent and we only compile to the source (or IR) level, we rely on the optimizations performed in the native compilers.

A note on thread independent scheduling. With the Volta generation of GPUs, NVIDIA relaxed the lockstep execution within warps. Every thread now has its own state, represented by a PC and stack, and instructions from divergent threads can be interleaved. This means that e.g., if two threads execute the same `_shfl_sync` command, they may do that at different points in time. Hence, NVIDIA's warp-level primitives now include bitmasks that determine all threads which block at the respective `_shfl_sync`, giving users finer-grained control over synchronization on the sub-warp level. PIRCH, in order to maintain compatibility with SIMD execution, only uses one state per WG. Therefore, borG's SIMT backend emulates the pre-Volta mode of lockstep execution by inserting warp synchronization primitives and appropriate masks where applicable to ensure correctness. As a consequence, architectures supporting thread-independent scheduling are not fully exploited by borG, potentially resulting in a performance penalty. As borG's main focus is its portability, an extension of borG to accommodate this model would require representing the

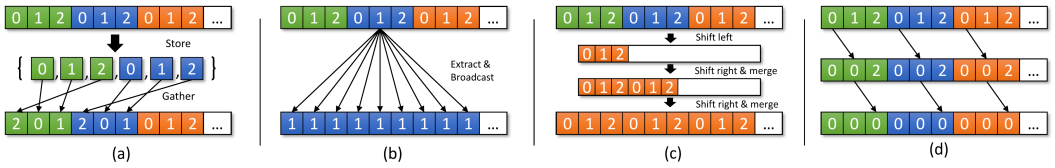


Fig. 4. Normal store/gather and our three cross-lane shuffle/reorder implementations for SIMD architectures to speed-up borG's kernels (from left to right): **Store/gather** for arbitrary permutations, **Scalar Broadcast** as well as **Vector Broadcast** and **Segmented Scalar Broadcast**, the latter being implemented by repeated logarithmic lane shifting and merging.

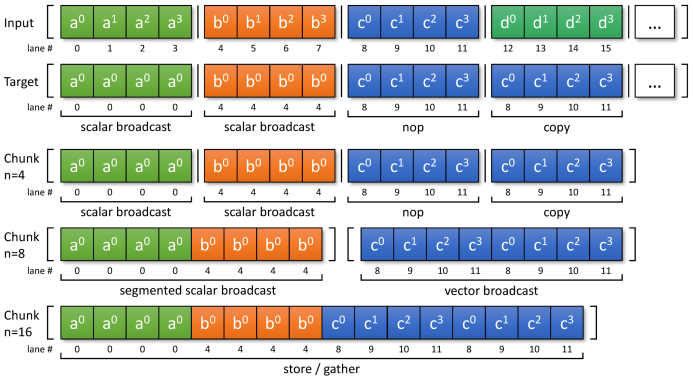


Fig. 5. When an InfReg is split into pieces of native SIMD width (*chunks*), the length of each of those chunks can change the selected implementation for reorderings. Within a general permutation vector, there may be chunk-aligned sub-sequences that fit one of our patterns.

WI's state in InfRegs, adding masks to the `shuffle` functions as well as modifications to the SIMD backend. We leave this extension for future work.

4.2.2 SIMD Backends. The initial mapping from InfReg-IR to SIMD-based ISAs proceeds by splitting arbitrary-length IR vectors into pieces in the native vector length, which we call *chunks*. This is possible due to the packing constraints enforced by the frontend after SSA'ing the input code: all interacting data in InfRegs are aligned to the same lanes; reorderings have been processed in an earlier assignment and the resulting implicit variable is aligned in its InfReg as well. We implemented a common "SIMD" backend for AVX512 and SX-Aurora and specialized only the configuration of scalar type-specific native vector lengths and primitive mappings from IR operations to vendor-specific C intrinsics. Memory accesses use the analysis from the frontend part and are either mapped to contiguous load and store calls or gather/scatter instructions.

Specialized reorderings for cross-lane operations. CUDA kernels following the warp register cache idiom often use an approach where each thread stores only parts of the problem's input data in its registers and uses shuffles to access other threads' data. Mapped to InfRegs, this approach leads to a high number of undesirable cross-lane operations. By default, such operation default to issuing a store and a gather call to permute the lanes contents. An analysis of the AVX512 and SX-Aurora instruction sets has resulted in three techniques mirroring useful shuffle patterns in CUDA (see Fig. 4):

- Scalar broadcast: `a[ix] = shuffle(a[0], 2)`,
- Vector broadcast: `a[ix] = shuffle(a[ix], 2)` and
- Segmented scalar broadcast: `a[ix] = a[0]`.

Each of these methods works within one native register. Interestingly, the length of chunks covering an InfReg can make a striking difference regarding which method is selected, as evident from Fig. 5. In practice, shorter vectors see more scalar broadcasts, longer vectors more vector or segmented scalar broadcasts. Additionally, when the same permutation vector appears multiple times, we just the register in question.

4.2.3 LLVM backend. The simplest among the backends is the LLVM backend. Given InfReg-IR, it outputs LLVM-IR that may then be compiled for all architectures supported by LLVM, such as x86, ARM or AVX512. Converting from InfReg-IR to LLVM-IR only drops the `!borg` metadata. With the exception of masked scatter and gather, we avoid LLVM intrinsics (as they are not supported across all architectures) and use sequential emulations. In some cases, the LLVM backends recognize these and generate vectorized code. Chunking and implementation selection is left to LLVM's backends.

5 EVALUATION

We evaluate borG on 7 batched kernels from linear algebra (1-4), machine learning (5) and numerical optimization (6-7). With $n =$ `WG_SIZE` and uniformly random matrices $A \in \mathbb{N}^{n\times n}, B \in \mathbb{N}^{n\times n}, C \in \mathbb{N}^{n\times n}$, these are:

(1) **GEMA**: matrix-addition $C = A + B$;
(2) **GEMM**: matrix-matrix multiplication $C = AB$ in outer product formulation;
(3) **TRSV**: 2 subsequent $n \times n$ triangular solves of an n-vector b;
(4) **LDUS**: $n \times n$ LDU factorization $A = LDU$ with diagonal D and **s**tatic pivoting;;
(5) **PADE**: backward pass for the learnable Padé Activation Unit [37];
(6) **ENUM**: enumeration of small integer programs: $\max c^\top x$ s.t. $Dx \leq b$, $x \in \{0,1\}^{10}$, $b \in \mathbb{R}^8$;
(7) **FRCG**: Fletcher-Reeves CG [20] optimization on the Rosenbrook family of functions $f(x, y) = (a - x)^2 + b(y - x^2)^2$ with minimum line search and a fixed number of steps.

Most of the kernel choices were application driven. For example, one of the pillars of scientific simulations in the peta- resp. exascale eras are communication-avoiding algorithms that scale well over thousands of nodes. One prominent case is the block-Jacobi preconditioner [2] that has recently been augmented with mixed precision [4]. Computing and using this preconditioner requires kernels (2-4). As one more expensive alternative to Jacobi preconditioning, incomplete block-matrix factorizations [50] require kernel (1) to perform Schur downdates. For very large sparse matrices beyond a few million rows, this quickly leads to batch sizes of 100,000. ENUM is used in a research project based on [6, 13] in which we repeatedly enumerate the whole solution space of truncated integer programs of k variables – there, the batch size is 2^k. Additionally, we selected the kernels such that they cover different reordering patterns and functionalities: GEMM uses both segmented scalar and vector broadcast, TRSV relies on implicit reorderings to convert between IR formats, LDUS uses masking and ENUM type conversions. Our set of benchmarks covers both kernels (1, 2, 4) represented in vendor libraries as well as custom codes.

We group our kernels into two groups by their scheme for data-parallelism: "group-wise" kernels (1-4) process one batch item per WG, hence the maximum InfReg width is $\texttt{WG\_SIZE}^2 \times \texttt{WG\_PACK}$. "Item-wise" kernels process one batch item per WI, using `WG_SIZE` as a tunable parameter.

Since borG's purpose is supporting rapid cross-architecture development of batched general-purpose computing kernels, we compare against hand-written, hand-tuned **references** for each accelerator. All reference kernels were written targeting the architectures' native programming models and run through their native compilers. For CUDA, we use NVIDIA's **NVCC** and used `WG_SIZE`, as a template parameter. All data is stored in private arrays and as all array indices can be resolved at compile time, NVCC can cache array accesses in registers. Whenever possible, we also used (sub-)warp shuffles using NVIDIA's cooperative groups API. For SX-Aurora, we used NEC's

Kernel	CUDA	CUDA/TC	AVX512	AVX512/LLVM	VE
GEMA	0.71×	0.59×	1.00×	1.02×	0.98×
GEMM	1.30×	0.96×	2.26×	2.92×	0.22×
TRSV	2.00×	-	1.70×	1.60×	1.87×
LDUS	3.31×	-	2.38×	2.86×	1.72×
PADE	0.96×	-	0.91×	-	3.82×
ENUM	0.91×	-	1.57×	-	1.29×
FRCG	0.82×	-	1.26×	2.63×	0.28×
geom. Mean	1.23×	0.75×	1.49×	2.05×	1.00×

Table 2. Geometric mean speedups for borG vs. the reference kernels, computed over all `WG_SIZE`s.

auto-vectorizing **NCC** compiler v3.0.28. We embed scalar kernels into OpenMP loops over all batch items, allowing the compiler to vectorize beyond the boundaries of single problems (similar to WG packing). AVX512 code is handled by **ISPC**, which has proven to generate code that often surpasses hand-written intrinsics. We include the source code of all kernels as supplementary material in the interest of transparency.

In addition to our references, we compare linear algebra kernels to vendors' BLAS libraries. These libraries and kernels are, in order GEMA/GEMM/LDUS: NLC BLAS (`{s,d}axpy_`/`{s,d}gemm_`/-) for Aurora, MKL (`cblas_{s,d}axpy`/`{S,D}GEMM_BATCH`/`{S,D}GETRF`) for AVX512 and cuBLAS (`cublas{S,D}axpy`/`cublas{S,D}gemmStridedBatched`/`cublas{S,D}getrfBatched`) for CUDA. Where necessary, we have added parallelizing for-loops around BLAS to support batched workloads. Since BLAS only covers standard kernels, we also compare to Tensor Comprehensions (TC) [52], an optimizing generator for custom kernels for NVIDIA GPUs. TC infers iteration variables and their ranges automatically from the source code. Partial iterations, like in line 22 in Ex. 3 must be unrolled into separate kernels and reduced in a separate kernel at the end. Since the resulting performance is far from the range that borG's kernels lie in, we only include GEMA and GEMM which are implemented as a single TC function each.

borG's generated code is compiled by GCC 8 (AVX512), the LLVM-VE v1.13 (Aurora) and NVCC (CUDA). We run both AVX512 and SX-Aurora experiments on a server featuring an Intel Xeon 6126 Gold processor, 96 GB RAM and 2 NEC SX-Aurora TSUBASA 10B cards running on CentOS 7. CUDA experiments are executed on a PC with an Intel i7-9700K CPU, 32 GB RAM and a NVIDIA TITAN V GPU. We use CUDA 10.1 with driver version 430.50. All compilers use the flags `-O3 -march=native` for optimization. We use a batch size of 100, 000 and report the minimum execution time of 10 benchmark rounds after 2 warmup rounds in order to compensate for initialization and cache effects. We measure execution time using C++11's chrono API. The correctness of all kernels was checked by automatically generated tests, comparing the results against a CPU reference implementation.

5.1 Comparison with reference kernels.

We build all kernels for all three architectures with parameters `WG_PACK` $\in \{1, 2, 4\}$ and `WG_SIZE` $\in \{4, 8, 12, 16, 20, 24, 28, 32\}$ for "group-wise"-kernels and up to the native vector width for "item-wise" kernels. Figs. 6 and 7 plot the best `WG_PACK` configurations' runtimes per `WG_SIZE`. A quantitative result on the speedups over native kernels is presented in Tab. 2. Averaged over all architectures, borG's generated code achieves speedups over the references of 1.23× for CUDA, 2.05× for AVX512 (via LLVM) and comes to a draw for the SX-Aurora. We found that, (near-) optimal configurations may be determined by making sure that `data per WI` × `WG_SIZE` × `WG_PACK` is less or equal to 2 times the native vector length on SIMD systems.

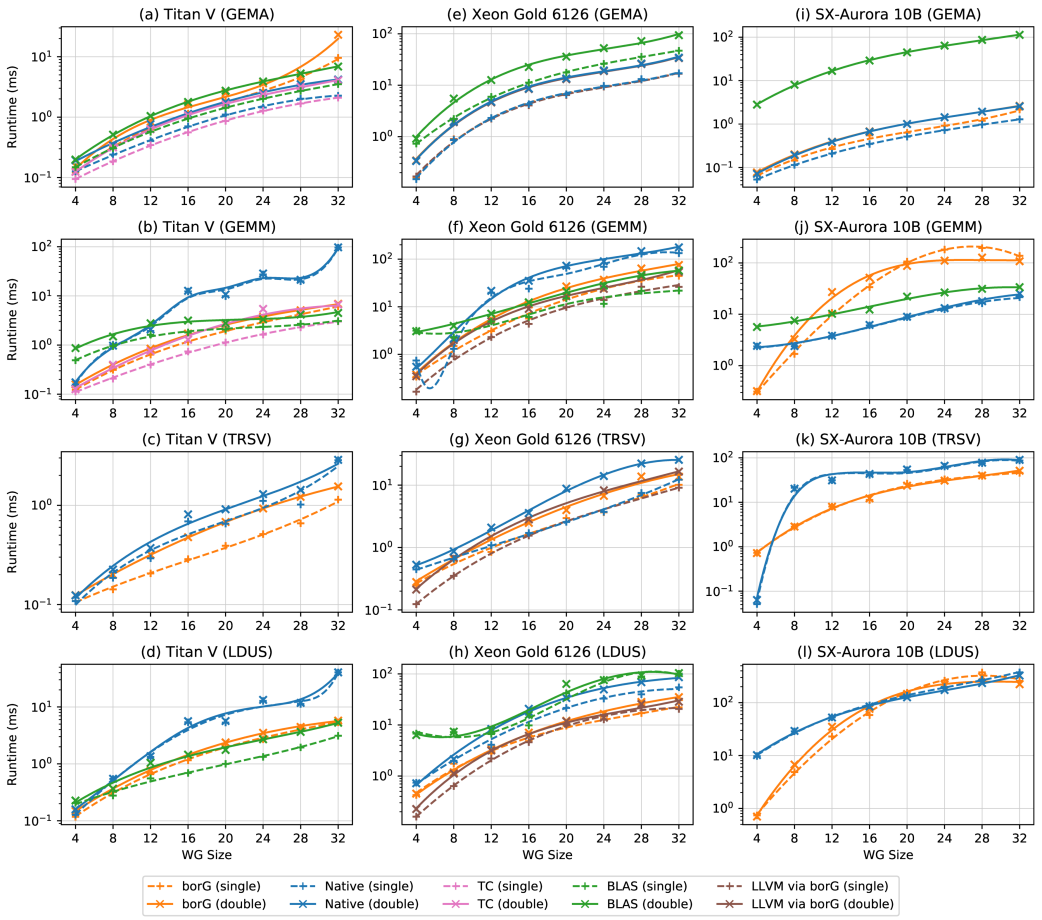


Fig. 6. Runtimes (mind the **logarithmic scale**) of kernels (1-4), with one kernel per row and one device per column. We plot both single and double precision separately and only plot the configuration with the best `WG_PACK` for each `WG_SIZE`. These kernels process matrices of sizes `WG_SIZE` × `WG_SIZE` resp. vectors of dimension `WG_SIZE`.

Group-wise kernels. Kernels (1-4) process (multiple) matrices of size `WG_SIZE` × `WG_SIZE` in each batch item. Across all measured `WG_SIZE`s, borG's GEMA kernel on CUDA is 39% slower than its reference. Since there is no data reuse in GEMA, the extra effort for caching both input matrices does not pay off here. Instead, borG's register caching scheme limits occupancy and thus bandwidth. This is in stark contrast to the LDUS kernel on CUDA: With a high degree of data reuse directly from registers, borG achieves a speedup of 3.31×. Particularly for double precision, registers (holding 3 matrices) are the limiting factors for `WG_SIZE`s close to 32. TC generates kernels from a higher-level description and tunes them using a genetic algorithm for CUDA. Hence, it has full control over data layouts as well as the order of instructions. For GEMA, where memory transfers outweighs computation, TC shines in comparison to borG by generating coalesced memory accesses. However, in the computation-bound GEMM kernel, borG is only 4% slower on average than TC. Interestingly, the performance gap is higher for single precision than for double precision. The reason here is still not clear to us.

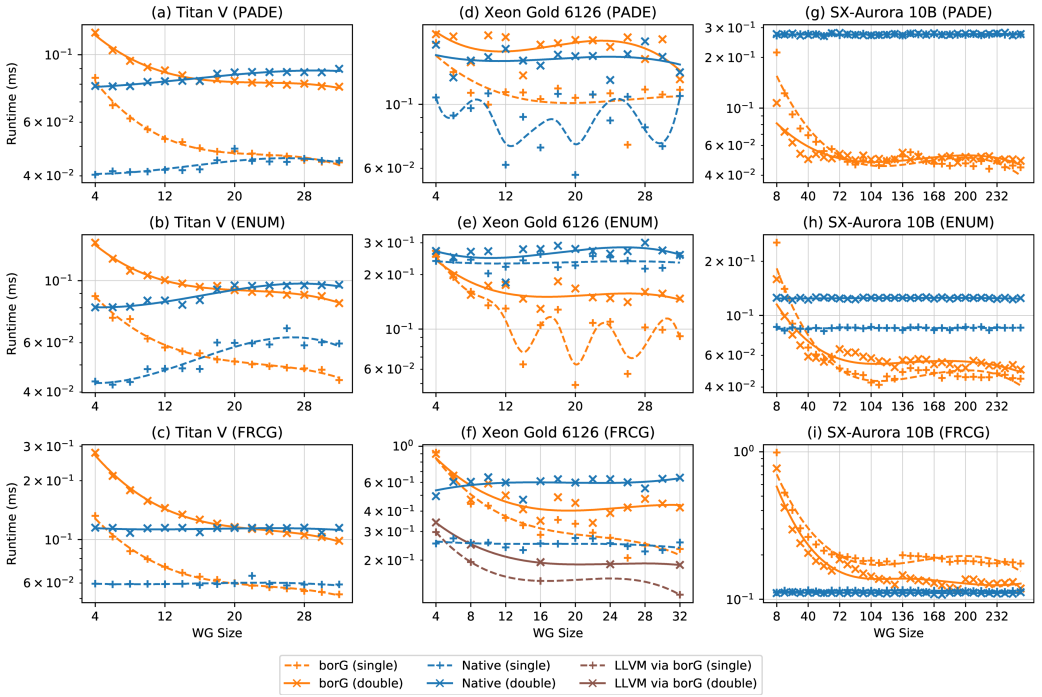


Fig. 7. Runtimes (mind the **logarithmic scale**) of the 3 "item-wise" benchmark kernels, following the same methodology as Fig. 6. Here, each work item processed ont batch item, i.e., the `WG_SIZE` determines how many batch items are processed per WG. borG's kernels can keep up with CUDA and outperform ISPC, even though both are designated for this type of parallelization.

On AVX512, both of our own backends and the LLVM-IR backend outperform ISPC on almost all kernels. ISPCs SPMD-on-SIMD approach, putting each kernel instance into individual register lanes versus borG's PIRCH-based approach leads to a significantly higher amount of spills, slowing down its kernels' execution time. LLVM's own backend for AVX512 outperforms our own chunked AVX512 backend; the heavy use of intrinsics in our code prevents the compiler from optimizations. Moreover, LLVM's AVX512 backend is a mature compiler, able to access the whole range of the AVX512 ISA, whereas our research compiler only offers a limited selection of intrinsics.

On SX-Aurora, GEMM is the worst case for borG. Due to the wide SIMD vectors, most reordering operations fall back to the store/gather approach. Our kernel's outer product approach to GEMM roughly uses one FMA per reordering, i.e. gather operation, creating a bandwidth-bound kernel. As Aurora's ISA offers packed float operations only for a subset of instructions, in all other cases, we resort to splitting a float vector into double vectors and processing both parts individually, which effectively more than doubling the number of compute instructions for single-precision kernels. Moreover, the immature LLVM-VE compiler stack was unable to compile some kernel instances due to the lack of register spilling slots available during register allocation. This varied over LLVM-VE versions – hence, the results for VE are to be taken with a grain of salt.

For GEMM, GEMA and LDUS, we include runtimes for batched kernels from the vendors' BLAS libraries as an absolute reference point. While we have the lead in the GEMA kernels (where we use DAXPY/SAXPY and a memcopy operation to express $C = A + B$), borG's code does compare poorly on GEMM on larger matrices - a code that is classically ill-suited for the register cache approach since it needs to cache 3 full matrices, leading to massive register spills. However, due to its massive

register file, borG stands its ground for GEMM on CUDA. For kernels that benefit from a register cache, borG is able to keep up, as evidenced by LDUS. At least the AVX512 BLAS comparisons, however, should be taken with a grain of salt: lacking some batched kernels in the MKL [53], we used OpenMP to parallelize BLAS kernels over the batch size. Additionally, BLAS kernels are more flexible, accepting matrices of arbitrary size, while borG specializes for one fixed size per compile run, saving a lot of overhead in the process, including cache-aware tiling of input matrices. On the other hand, these kernels have often been hand-tuned by vendors on the assembly level.

Item-wise kernels. These kernels (results in Fig. 7) rarely use cross-lane operations, making them prime examples for ISPC and CUDA. Given the kernels' short runtimes, results for AVX512 benchmarks are noisy due to OS and scheduler influences. However, we still see an edge for borG – it comes close or exceeds ISPC's performance. On CUDA, we are second to NVCC on smaller `WG_SIZE`s, but consistently have an small advantage for larger `WG_SIZE`s. This trend continues on SX-Aurora – where borG catches up once the wide vectors are filled to a sufficiently degree. NCC is able to vectorize over all batch items directly, explaining its consistent performance. We point out that NCC seemingly operates with a reduced precision even with optimization level `-O3`; both in FRCG and LDUS, it lacks precision compared to all the other backends and compilers. As Fig. 9 (f) shows, this can make a considerable difference on SX-Aurora.

5.2 Ablation studies.

We close with ablation studies on the performance impact of the crucial `WG_PACK` parameter and the optimizations in borG's SIMD backends as discussed in Section 4.2.2 using the double precision LDUS kernel. Fig. 8 visualizes the the effects of `WG_PACK`. For CUDA, higher packing leads to higher register pressure and thus limits the SM's ability to hide latencies. The SIMD backends show a different characteristic: As long as there are empty lanes in vector registers, packing improves performance. On AVX512, due to its shorter vector length, this tipping point comes much earlier than for SX-Aurora. Besides the higher vector length, each core of the SX-Aurora has a higher number of 96 vector registers vs. 32 for AVX512. We consider `WG_PACK` to be the crucial parameter to scale borG's kernels across both short and long SIMD systems.

Next, we quantify the impact of specialized reordering in the SIMD backends. Starting with configuration (a), where all cross-lane operations gathers, we enable the following optimizations: (b) reuse a previous chunk, (c) scalar broadcast, (d) vector broadcast and (e) segmented scalar broadcast. Additionally, we extend (d) to use an approximation for floating point division (RCP14) as configuration (f). Furthermore, we evaluate the streaming option from Sec. 4.1. The labels in Fig. 9's bars are the configurations' runtimes relative to "baseline" configuration (a). In general, these results confirm that specialized reordering significantly improves performance over store/gather across `WG_SIZE`s. As illustrated in Fig. 5, depending on the chunk size and WG configuration, the same reordering pattern may be decomposed into different specialized reorderings. AVX512 has no instruction that would enable any of the logarithmic shuffle patterns. However, due to its shorter SIMD length, borG can often use scalar broadcasts instead or reuse (i.e. copy) other chunks – therefore, those two optimizations improve performance the most. For SX-Aurora, the longer SIMD width shifts the balance towards vector and segmented scalar broadcasts. While the latter's move pattern can result in a speed-up of 20%, using logarithmic shifts for segmented scalar broadcasts results in a heavy performance penalty. The documentation of the SX-Aurora's architecture does not provide an explanation for this behavior. Streaming turns out to not be of any use; dealing with register oversubscription is best left to the compiler itself.

Lastly, we investigate the effects of using runtime-bound ("loop") instead of compile-time unrolled ("unroll") loops. For this experiment, we wrap the GEMA-kernel's addition in a for-loop, repeating it 5 times, i.e., $C = 5 \cdot (A + B)$. Fig. 10 plots the slowdown factor for all three architectures over

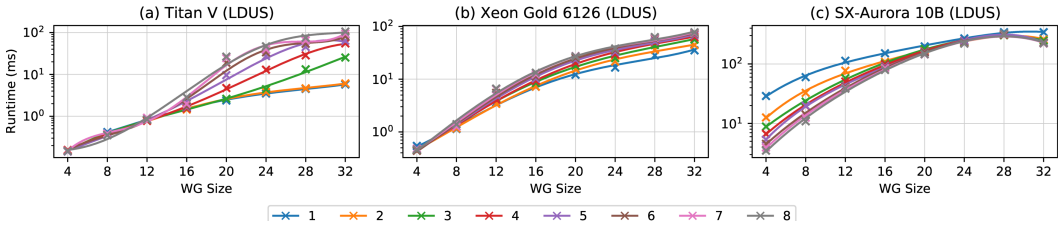


Fig. 8. Parameter studies for the influence of the WG packing factor: while packing has an adverse effect in CUDA, the parameter proves to be crucial to fill vectors of wide SIMD systems such as SX Aurora.

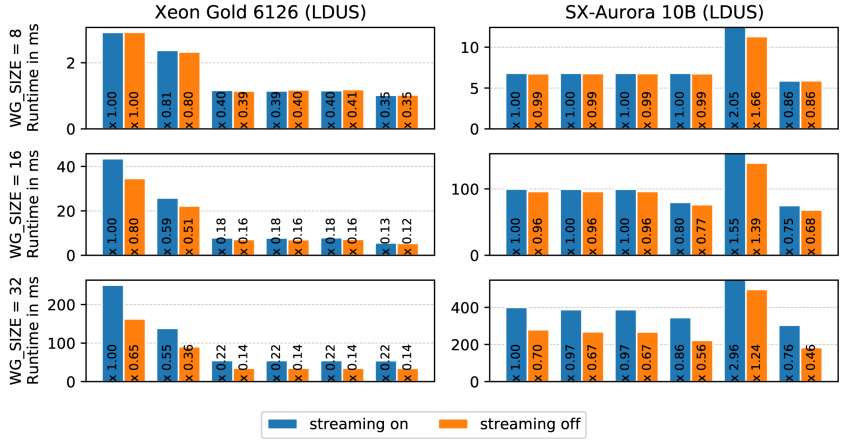


Fig. 9. Ablation study for specialized reordering functions on the LDUS kernel. As a baseline, we use store/-gather (a) for each reordering and then continue to enable more specialized functions ((b) reuse, (c) scalar broadcast, (d) vector broadcasts, (e) segmented scalar broadcasts and (f) RCP approximation for division). Except for (f), all specialized implementations improve runtimes over a store/gather approach when discovered by borG.

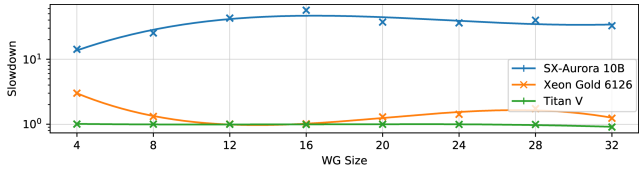


Fig. 10. Ablation study for the effect of masking due to runtime-bounded loops. We loop over a GEMA kernel 5 times and plot the slowdown factor when we use runtime loops vs. compiler-time, unrolled loops. Note that all WIs are still executing the same code all the time, i.e., there is no divergence.

`WG_SIZES`; we use `WG_PACK` = 1 for all builds. Runtime-bound loops are implemented through masked instructions in the body and a `popcount()` to vote on the stopping criteria over the whole WG. Here, creating masks involves implicit cross-lane communication and each mask application lead to register duplication on the device, hitting Aurora's performance hard (up to 30x loss). On AVX512, most chunks of the reordering resolve to simple assignments and cross-lane communication is much cheaper, especially when the `WG_SIZE` matches multiples of the hardware SIMD width. As a result, we see a less staggering drop in performance (up to 2.5x). On GPUs, there is almost no noticeable slowdown: all cross-lane communication maps to fast `shuffle` calls and branches as well as predication are supported by the hardware. We conclude that one should prefer compile-time constant loops and minimize mask usage in order to maintain performance portability.

6 RELATED WORK

The warp register cache idiom has become mainstream in the CUDA community [5, 16, 48, 50], having been generalized into "task-parallel programming for warps" [8].

Architectures and Programming models. SIMD and SIMT are not the end of the architectural developments; researchers are exploring extensions to both. Tino et al. [51] design and simulate a SIMT micro-architecture that includes speculative execution and out-of-order execution with register renaming. Likewise, Fung et al. [21] investigate the dynamic re-building of warps on the fly in case of branching code, which is beyond the current-generation SIMT model On the software side, several warp-explicit programming models have been proposed in the past. Warp consolidation [35] makes the implicit warp-centric programming explicit by a series of (manual) code transformations which launch only one warp per CUDA block. Chen et al. [15] draw inspiration from hardware systolic arrays and map them onto CUDA warps, which is very effective for kernels with completely static control flow that can be expressed as a sequence of unary transformations and warp reductions. Similarly, warps are used as the smallest executable units that offer the benefit of being able to alternate between SIMD and SISD phases in [35]. All these models share the trait of mapping their computation to warps, minimizing global communication. However, none of these works comes with a compiler or code generator; they are merely guidelines how to implement a kernel.

DSLs and code generation. While borG is tailored towards a programming *idiom*, *domain-specific languages* (DSLs) evolved as a means to rapidly implement *applications* in a limited area. DSLs can either generate high-level code in a more general language or directly go to an IR level such as LLVM-IR. For batched Cholesky factorization and Kalman filters, Lemaitre et al. [34] propose a template system. Rodrigues et al. [44] specify a small DSL for static tensor multiplications – even parallelizing error correction in 5G base stations [14] warrants a DSL. Likewise, there is a DSL for stencil operations, prime examples for memory-bound kernels and the importance of minimizing memory transformations through registers, in CUDA [56]. Code generation is not limited to DSLs, though: several approaches transform scalar and sequential into parallel code, either through an IR [9] and different backends or directly through a reinforcement learning approach, that learns when and how to parallelize and chunk for-loops [25]. Ben-Nun et al. [10] propose a dataflow multi-graph as IR that allows to split domain science and performance engineering in the development process and generates code for different platforms. On the CPU side, Bertolacci et al. [12] identify loop chains as a code construct that defines data sharing and parallel schedules that allow automatic OpenMP code generation. A recent system specializing on tensor kernels for deep learning is Tensor Comprehensions [52] which integrates polyhedral code generation, auto-tuning and a DSL for tensor expressions in Einstein notation.

Vectorizing compilers. The most general approach for accelerator programming are vectorizing and parallelizing compilers. Such compilers mainly work by analyzing execution patterns within (possibly nested) loops, including function calls. [45] detects computations in a program that may be performed simultaneously in SIMD lanes, and optimizes for vector register usage in order to minimize memory traffic. The Region Vectorizer [28] is the first open source vectorizer that works on the whole-function level. After converting a whole function to an SSA based representation, execution flows are traced, masks generated and loops reordered. Its latest extension, TensorRV [39] extends its capabilities to nested, multidimensional loops It is also used as a part of a larger system PACXXv2p [24], which – driven by a single coherent programming model resembling the data-parallel and memory abstractions of CUDA – can build native kernels for CPU and GPU. Scalar computation is wrapped into three levels of for-loops which are then vectorized treated by RV. The classical approach to vectorization is the polyhedral model for code generation [22]. One

could say that auto-vectorizing compilers offer a common representation for SIMD and SIMT as well: C code. From the same C code, they generate programs for SIMD-CPUs and GPUs. However, we argue that SIMD auto-vectorization is somewhat opaque, in contrast to PIRCH and borG, where a well-defined execution model allows the developer to stay close to the actual hardware – the transformation is very transparent.

The closest competitor to borG is ISPC [43]. Inspired by graphic shaders, ISPC works with a SPMD-on-SIMD model, which is similar in concept to our SIMT-on-SIMD model. There are two main differences: first, ISPC maps every kernel to one SIMD lane; e.g., a batched, 32×32 GEMM requires $3 \times 32 \times 32 = 3096$ SIMD registers, far exceeding the capacities of any SIMD processor - but only 12 vector registers on the SX-Aurora using borG. Thus, SPMD-on-SIMD is not well-suited for CoRe based kernels. Second, ISPC arranges threads in *gangs*, similar to our WGs, but with a fixed size that is dictated by hardware. On the other side, its longer development history and support by Intel makes it a much more general and production-oriented system that goes beyond borG's capabilities when it comes to match data structures to computation. The co-developed Sierra [33] is a C/C++ extension shares some ideas such as automatic blocking and conversion of data structures.

A lot of compilers for high-level languages are based on the compiler infrastructure LLVM [30] with its IR in SSA-form that is both hardware and language-independent, but permits the inclusion of platform-specific intrinsics. LLVM IR offers native support for fixed-length vectors of simple datatypes but, so far, lacks the support for vector features such as an active vector length (e.g. SX-Aurora) or masked instructions. Furthermore, LLVM IR does not offer a mapping from SIMD lanes to SIMT threads for e.g., compilation for GPUs. Instead, the internal GPU support works on scalar code that is run on each thread on the SIMT system. An extension with SIMT support, especially regarding the semantics of thread divergence, was proposed [23]. Over the last decade, LLVM IR has become sort of the "lingua franca" for the compiler community; it is, however, relatively low-level and lacks support for custom extensions. In order to overcome these weaknesses, [31] propose MLIR, a framework for the definition of customized SSA-based *IR dialects*. MLIR allows developers to implement generalized compiler passes on the control flow graph, enabling a progressive lowering of such dialects to vanilla LLVM IR. Another IR that is used mainly in the graphics area is Khronos' SPIR [29]. SPIR-V, its latest version, is a portable binary format that describes compute kernels and graphics shaders for GPUs and is used as an IR in popular graphics APIS OpenGL, Vulkan but also offers support for OpenCL 2.1.

Portability. Vectorizing compilers convert scalar code into code for vectorized computation. Since there are many platforms that can deal with vectors, a mapping from vectorized code to the platform is the next step to a final binary. Thus, another category of works target *program portability*, i.e., the ability to use a program on different platforms without re-implementing it. For SIMD platforms, this entails the ability to work with different hardware SIMD widths, ideally without recompiling. Vapour SIMD [41] solves this problem by explicitly embedding the vector idioms that a compilers auto-vectorization pass detects in scalar code in a custom bytecode format. A recent proposal for an extension to LLVM [38] also lowers arbitrary-length vectors in LLVM IR to the underlying SIMD platform at compiler time. At runtime, a Just-In-Time compiler is used to lower the bytecode to the SIMD instruction set in use, often comparable in performance to native vectorization. Liquid SIMD [17] proposes a lower-level approach: mapping a baseline scalar instruction set to vector instructions at runtime *in hardware*. By encoding induction variables for scalar loops, a deterministic finite automaton may be synthesized for this task.

A stricter variant of program portability is *performance portability*, which was the goal of the OpenCL [47] standard. Du et al. [19] evaluated its effectiveness on a GEMM kernel, concluding that it is vital to tune parameters, such as blocking or unrolling, per platform. A similar evaluation has been done for the competing OpenACC standard [36] with similar results. Building on this

portability, Steuwer et al. [46] propose an IR for higher-level languages that encodes some of OpenCL's concepts, allowing tuning and optimization on a lower level and outputting transformed OpenCL code. The performance and versatility of OpenCL depends on the quality of drivers and compilers implemented on each platform. For a SIMT-to-SIMD style transformation, CPU-based OpenCL drivers apply ISPC's approach, sharing weaknesses and advantages. While borG inevitably shares some concepts with the referenced works, its hybrid approach, integrating a code generator for flexible WG parameterization, an abstract IR for both SIMD and SIMT representation and a cross-architecture source-to-source compiler makes it a unique approach.

7 CONCLUSION AND FUTURE WORK

We observed that programs following the *collective register cache* idiom are good candidates for performance-portable, cross-architecture computational kernels. We therefore defined a generalization of the idiom (CoRe) and a virtual architecture (PIRCH) to abstract SIMD and SIMT architectures. We presented the borG compiler, to automatically map the programs to PIRCH, and generate optimized compute kernels for three different architectures.

The provided kernels provide similar performance to hand-tuned implementation on all three evaluated architectures, and in some cases, borG even produced code that is comparable to vendor-tuned libraries, significantly reducing the burden for programmers in need of custom kernels.

Besides future work previously mentioned, we plan to extend abstractions to add more dynamic control flow and runtime-indexed shuffles. Furthermore, we would like to couple borG with stencil optimizers to address the generation of custom deep learning kernels. We already proposed a hardware extension [49] that realizes a variant of "runtime"-PIRCH on a slightly modified vector architecture.

REFERENCES

[1] Farhana Aleen, Vyacheslav P Zakharin, Rakesh Krishnaiyer, Garima Gupta, David Kreitzer, and Chang-Sun Lin. 2018. Automated compiler optimization of multiple vector loads/stores. *International Journal of Parallel Programming* 46, 2 (2018), 471–503.

[2] Yussuf Ali, Naoyuki Onodera, Yasuhiro Idomura, Takuya Ina, and Toshiyuki Imamura. 2019. GPU acceleration of communication avoiding Chebyshev basis conjugate gradient solver for multiphase CFD simulations. In *2019 IEEE/ACM 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. IEEE, 1–8.

[3] Andrew Anderson, Avinash Malik, and David Gregg. 2015. Automatic vectorization of interleaved data revisited. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 4 (2015), 1–25.

[4] Hartwig Anzt, Jack Dongarra, Goran Flegar, Nicholas J. Higham, and Enrique S. Quintana-Ortí. 2019. Adaptive Precision in Block-Jacobi Preconditioning for Iterative Sparse Linear System Solvers. *Concurrency and Computation: Practice and Experience* 31, 6 (2019), e4460.

[5] Hartwig Anzt, Jack Dongarra, Goran Flegar, and Enrique S. Quintana-Orti. 2017. Variable-Size Batched LU for Small Matrices and Its Integration into Block-Jacobi Preconditioning. In *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, Bristol, United Kingdom, 91–100.

[6] David Applegate, Robert Bixby, Vašek Chvátal, and William Cook. 2001. TSP Cuts Which Do Not Conform to the Template Paradigm. In *Computational Combinatorial Optimization*. Springer, 261–303.

[7] Paul Barham and Michael Isard. 2019. Machine Learning Systems Are Stuck in a Rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. ACM, Bertinoro Italy, 177–183.

[8] Michael Bauer, Sean Treichler, and Alex Aiken. 2014. Singe: Leveraging Warp Specialization for High Performance on GPUs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 119–130.

[9] David A. Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J. Kunen, Olga Pearce, Peter Robinson, Brian S. Ryujin, and Thomas RW Scogland. 2019. RAJA: Portable Performance for Large-Scale Scientific Applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 71–81.

[10] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N. Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful Dataflow Multigraphs: A Data-Centric Model for Performance Portability on Heterogeneous Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[11] Eli Bendersky. 2010. PyCParser. https://github.com/eliben/pycparser
[12] Ian J. Bertolacci, Michelle Mills Strout, Stephen Guzik, Jordan Riley, and Catherine Olschanowsky. 2016. Identifying and Scheduling Loop Chains Using Directives. In *2016 Third Workshop on Accelerator Programming Using Directives (WACCPD)*. IEEE, 57–67.
[13] Christoph Buchheim, Frauke Liers, and Marcus Oswald. 2008. Local Cuts Revisited. *Operations Research Letters* 36, 4 (2008), 430–433.
[14] Adrien Cassagne, Olivier Aumage, Denis Barthou, Camille Leroux, and Christophe Jégo. 2018. MIPP: A Portable C++ SIMD Wrapper and Its Use for Error Correction Coding in 5G Standard. In *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*. 1–8.
[15] Peng Chen, Mohamed Wahib, Shinichiro Takizawa, Ryousei Takano, and Satoshi Matsuoka. 2019. A Versatile Software Systolic Execution Model for GPU Memory-Bound Kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–81.
[16] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. Cudnn: Efficient Primitives for Deep Learning. *arXiv preprint arXiv:1410.0759* (2014). arXiv:1410.0759
[17] Nathan Clark, Amir Hormati, Sami Yehia, Scott Mahlke, and Krisztian Flautner. 2007. Liquid SIMD: Abstracting SIMD Hardware Using Lightweight Dynamic Mapping. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE, 216–227.
[18] Javier Diaz, Camelia Munoz-Caro, and Alfonso Nino. 2012. A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era. *IEEE Transactions on parallel and distributed systems* 23, 8 (2012), 1369–1386.
[19] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. 2012. From CUDA to OpenCL: Towards a Performance-Portable Solution for Multi-Platform GPU Programming. *Parallel Comput.* 38, 8 (2012), 391–407.
[20] Reeves Fletcher and Colin M. Reeves. 1964. Function Minimization by Conjugate Gradients. *The computer journal* 7, 2 (1964), 149–154.
[21] Wilson WL Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. 2009. Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware. *ACM Transactions on Architecture and Code Optimization (TACO)* 6, 2 (2009), 1–37.
[22] Martin Griebl, Christian Lengauer, and Sabine Wetzel. 1998. Code Generation in the Polytope Model. In *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. 98EX192)*. IEEE, 106–111.
[23] Nicolai Hähnle. 2019. D68994 [RFC] Redefine 'convergent' in Terms of Dynamic Instances. https://reviews.llvm.org/D68994
[24] Michael Haidl, Simon Moll, Lars Klein, Huihui Sun, Sebastian Hack, and Sergei Gorlatch. 2017. Pacxxv2+ RV: An LLVM-Based Portable High-Performance Programming Model. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*. 1–12.
[25] Ameer Haj-Ali, Nesreen K. Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. 2020. NeuroVectorizer: End-to-End Vectorization with Deep Reinforcement Learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 242–255.
[26] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* 62, 2 (Jan. 2019), 48–60.
[27] David Joyner, Ondřej Čertík, Aaron Meurer, and Brian E. Granger. 2012. Open Source Computer Algebra Systems: SymPy. *ACM Communications in Computer Algebra* 45, 3/4 (Jan. 2012), 225–234.
[28] Ralf Karrenberg. 2015. Whole-Function Vectorization. In *Automatic SIMD Vectorization of SSA-Based Control Flow Graphs*. Springer, 85–125.
[29] John Kessenich, Boaz Ouriel, and Raun Krisch. 2018. SPIR-V Specification. *Khronos Group* 3 (2018).
[30] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.
[31] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. *arXiv:2002.11054 [cs]* (Feb. 2020). arXiv:2002.11054 [cs]
[32] Roland Leißa, Sebastian Hack, and Ingo Wald. 2012. Extending a C-like Language for Portable SIMD Programming. *ACM SIGPLAN Notices* 47, 8 (2012), 65–74.
[33] Roland Leißa, Immanuel Haffner, and Sebastian Hack. 2014. Sierra: A SIMD Extension for C++. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*. 17–24.
[34] Florian Lemaitre, Benjamin Couturier, and Lionel Lacassagne. 2018. Small SIMD Matrices for CERN High Throughput Computing. In *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*. 1–8.
[35] Ang Li, Weifeng Liu, Linnan Wang, Kevin Barker, and Shuaiwen Leon Song. 2018. Warp-Consolidation: A Novel Execution Model for Gpus. In *Proceedings of the 2018 International Conference on Supercomputing*. 53–64.

[36] M. Graham Lopez, Verónica Vergara Larrea, Wayne Joubert, Oscar Hernandez, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2016. Towards Achieving Performance Portability Using Directives for Accelerators. In *2016 Third Workshop on Accelerator Programming Using Directives (WACCPD)*. IEEE, 13–24.

[37] Alejandro Molina, Patrick Schramowski, and Kristian Kersting. 2019. Padé Activation Units: End-to-End Learning of Flexible Activation Functions in Deep Networks. *arXiv preprint arXiv:1907.06732* (2019). arXiv:1907.06732

[38] Simon Moll. 2019. D57504 [RFC]: Prototype & Roadmap for Vector Predication in LLVM. https://reviews.llvm.org/D57504

[39] Simon Moll, Shrey Sharma, Matthias Kurtenacker, and Sebastian Hack. 2019. Multi-Dimensional Vectorization in LLVM. In *Proceedings of the 5th Workshop on Programming Models for SIMD/Vector Processing*. 1–8.

[40] Dorit Naishlos, Marina Biberstein, Shay Ben-David, and Ayal Zaks. 2003. Vectorizing for a SIMdD DSP architecture. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*. 2–11.

[41] Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, and Ayal Zaks. 2011. Vapor SIMD: Auto-Vectorize Once, Run Everywhere. In *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, 151–160.

[42] NVIDIA. 2020. CUDA C++ Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[43] Matt Pharr and William R. Mark. 2012. Ispc: A SPMD Compiler for High-Performance CPU Programming. In *2012 Innovative Parallel Computing (InPar)*. IEEE, 1–13.

[44] Christopher Rodrigues, Amarin Phaosawasdi, and Peng Wu. 2018. Simdization of Small Tensor Multiplication Kernels for Wide SIMD Vector Processors. In *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*. 1–8.

[45] Jaewook Shin, J. Chame, and M.W. Hall. 2002. Compiler-Controlled Caching in Superword Register Files for Multimedia Extension Architectures. In *Proceedings.International Conference on Parallel Architectures and Compilation Techniques*. 45–55.

[46] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: A Functional Data-Parallel IR for High-Performance GPU Code Generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 74–85.

[47] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in science & engineering* 12, 3 (2010), 66–73.

[48] Daniel Thuerck. 2019. Stretching Jacobi: Two-Stage Pivoting in Block-Based Factorization. In *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 51–58.

[49] Daniel Thuerck. 2020. Supporting Irregularity in Throughput-Oriented Computing by SIMT-SIMD Integration. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3) (Short Paper)*. IEEE.

[50] Daniel Thuerck, Maxim Naumov, Michael Garland, and Michael Goesele. 2018. A Block-Oriented, Parallel and Collective Approach to Sparse Indefinite Preconditioning on GPUs. In *2018 IEEE/ACM 8th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 1–10.

[51] Anita Tino, Caroline Collange, and André Seznec. 2020. SIMT-X: Extending Single-Instruction Multi-Threading to Out-of-Order Cores. *ACM Transactions on Architecture and Code Optimization* 17, 2 (May 2020), 15:1–15:23.

[52] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2019. The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 4 (2019), 1–26.

[53] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel Math Kernel Library. In *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 167–188.

[54] Nicolas Weber and Michael Goesele. 2017. MATOG: Array Layout Auto-Tuning for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 3 (2017), 1–26.

[55] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, and Robert Hundt. 2016. Gpucc: An Open-Source GPGPU Compiler. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. 105–116.

[56] Tuowen Zhao, Protonu Basu, Samuel Williams, Mary Hall, and Hans Johansen. 2019. Exploiting Reuse and Vectorization in Blocked Stencil Computations on CPUs and GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–44.