Supporting Irregularity in Throughput-Oriented Computing by SIMT-SIMD Integration

Daniel Thuerck
NEC Laboratories Europe

***Abstract*—The last two decades have seen continued exponential performance increases in HPC systems, well after the predicted end of Moore's Law for CPUs, largely due to the widespread adoption of throughput-oriented compute accelerators such as GPUs. When faced with irregular yet throughput-oriented applications, their simple, grid-based computing model turns into a serious limitation. Instead of repeatedly tackling the issues of irregularity on the application layer, we argue that a generalization of the CUDA model to irregular grids can be supported through minor modifications to already established throughput-oriented architectures. To that end, we propose a unifying approach that combines techniques from both SIMD and MIMD approaches, but adheres to the SIMT principles – based on an unlikely ally: a wide-SIMD vector architecture. We extend CUDA's familiar programming model and implement SIMT-inspired strategies for dealing with data and control flow irregularities. Our approach requires only minimal hardware changes and an additional compiler phase. Using a model-based software simulation, we demonstrate that the proposed system can be a first step towards native support for irregularity on throughput-oriented processors while greatly simplifying the development of irregular applications.**

I. Introduction

The last decade has seen a massive increase in raw compute power due to the inclusion of GPUs in HPC systems. GPUs augment the previously dominant multi-core CPU setups by *throughput oriented computing* (TOC). Garland and Kirk [11] mention three important design principles for such systems: an abundance of rather simple processing units, SIMD-style execution and hardware multithreading. In this spirit, the scheduling hardware on GPUs is kept simple in favor of more compute, leading them to perform poorly in case of *control flow irregularity* (e.g. branch divergence, fine-grained synchronization) or *data irregularity* (e.g. differing resource requirements between work items). Modern multi-core CPUs, on the other hand, are optimized for *latency* and can use multithreading (SMT) to swap between executing applications. CPU cores operate independently and request resources as needed. Even though the concepts of latency and throughput-oriented architectures are diametrical opposites, researchers have tried to transplant features between them to speed up the execution of programs in their native programming models.

Classical domains where TOC proves effective are now facing the increasing use of irregular applications: sparse matrices, graph neural networks and sum-product networks frequently appear in e.g. the machine learning community. In order to extend the success and simplicity of CUDA's programming model to these applications, we propose software and hardware modifications that are quickly realizable and add native support for data and control flow irregularity.

Our contributions are as follows:

- We extend CUDA's PTX ISA to support an irregular compute model via an mapping to metadata registers in wide-SIMD processors (handling *data irregularity*).
- We describe a method to use register renaming as a tool for SMT-like processing on a simple vector core, avoiding costly context switches (handling *control flow irregularity*).
- Lastly, we integrate both ideas into a modified wise-SIMD architecture and validate the effectiveness of our ideas using a model-based software simulation.

This paper focuses on ideas and their integration; realizing a full system is an objective for future work. We back our claims and support the viability of our ideas using a model-based simulation (cf. below).

II. Related Work

The architectural continuum between throughput-oriented SIMT/SIMD designs and latency-based (e.g. SMT) designs has been explored to great depths. Multiple extensions to SIMT designs have been proposed: Scalar co-processors for GPU cores [3], [19], [22] that avoid repeated scalar computations; central schedulers share the GPU between host threads [14] or dynamic re-grouping of threads from a warp or block into convergent subgroups [9], [10]. Similar to SMT context switches, Frey et al. [8] propose a model for oversubscription of tasks to SIMT cores and a method for faster context switches using the cores' L1 cache. Similarly, work-stealing between warps has been explored [13].

On the other end of the spectrum, multiple works have investigated layering a SIMT scheduler on top of arrays of in-order CPU cores [2]. These arrays switch between MIMD mode (each processor operates independently) and SIMT mode (control logic is shared by all processors) to save power. Subsequent works extend this idea into a form of "hardware auto-vectorization" [5], [20] of scalar code. In order to group similar instructions on cores of the array, expensive crossbars are required.

Liquid SIMD [4] and Vapour SIMD [17] on the software side and ARM's SVE hardware extensions [1] improve SIMD systems for irregular applications: they offer a convenient way to set the SIMD vector length at runtime, adapting to tasks with varying resource requirements.

III. Programming Model

We now describe our proposed generalization of CUDA's grid-based compute model to irregular workloads. Traditionally, CUDA kernels are parameterized over a *grid* of *blocks*, e.g.

```
kernel<<<m, n>>>(...);
```

which launches `m` blocks of `n` threads. Each block is scheduled onto a streaming multiprocessor (SM) which executes *warps* of 32 threads. All threads within a warp operate in lockstep[1] and branches or conditionals are implemented by *predication* In recent GPU architectures, all threads in a warp share access to the SM's register file and communicate through it with low latency. Blocks of variable sizes $\mathtt{m}_i$ have to be emulated by setting the block size to $\mathtt{m} = \max\{\mathtt{m}_i\}_i$ and masking out threads in each block at runtime.

We build our proposed programming model with this issue in mind: First, we get rid of the block abstraction and directly expose warps to users. In order to handle data irregularity, we make the individual warps' sizes a runtime parameter. Instead of passing parameters `m`, `n` to the kernel, this requires passing an explicit list of warp sizes:

```
const int w_list[] = {4, 11, 3, 2, 8, 3};
kernel<<<m, w_list>>>(...);
```

As on GPUs, warps are assigned statically to SMs at initialization. Implementing kernels follows the same principle as for warp-centric [12] models: all threads in a warp execute in a bulk-synchronous manner and threads communicate through *shuffle* instructions. To distinguish code for our model from traditional CUDA code, we use the keywords `tid` for a thread's index in a warp and `wid`, `ntids` for a warp's index and size, respectively. As a poster child example, we use the SpMV-kernel in Fig. 1 (left), where each warp handles one row of a sparse CSR matrix (arrays `csr_row`, `csr_col`, `csr_val`). Therein, each thread handles one nonzero entry of the row and the results are accumulated by a warp-wide logarithmic reduction (lines 8 through 12). This simple kernel exhibits both data and light control flow irregularity: First, each warp uses a varying amount of threads and thus the share of the SM's register file depends on a runtime parameter where the classical CUDA execution model requires the register count at compiler time. Second, the number of reduction steps depends on the warp size, leading to different execution paths.

IV. Implementation

A little unconventional, we propose executing programs in the programming model from above on traditional SIMD hardware. Specifically, we consider wide-SIMD (i.e. vector) hardware due to their ability to *set a vector size at runtime*. The proposed system introduces additions to CUDA's C-to-PTX compiler and modifications to vector instruction buffers and register renaming units in hardware. Such buffers and renaming units are frequently found in SIMD microarchitectures (e.g. Intel CPUs with AVX). The fundamental ideas of our system are to translate SIMT code (with SIMD-friendly additions) into SIMD code *at runtime* (as opposed to e.g. binary translation [7]) and use register renaming tables to simultaneously execute multiple warps.

[1]With thread-independent scheduling as in the Pascal microarchitecture, the lockstep model has been somewhat relaxed.

A. Front-End

Executing SIMT code efficiently requires hardware support for predicated execution and branch- as well as reconvergence handling. In SIMT models, each thread inside a warp executes the same (scalar) code, but is parameterized by its index inside the warp (CUDA: `lane_id`). SIMD code, on the other hand, operates only on whole vectors at once. Thus, we propose modifications to CUDA's virtual PTX code in order to make it more SIMD-friendly, simplifying processing in the back-end. Changes are visualized using the SpMV example in Fig. 1.

Metadata registers. We follow the SIMT-on-SIMD paradigm of ISPC [18] by mapping the corresponding registers of threads in a warp to lanes in SIMD registers. We find that it is possible to emulate SIMT execution by explicitly associating keywords such as `$tid` in metadata registers to each lane (*parameterized SIMD execution*). Appearances of those keywords in the code are then translated to registers with suitable execution, as marked by (1) in Fig. 1. Unless there is branching involved, SIMT code translates 1:1 into SIMD code; predicated instruction are realized through masked SIMD registers. One more data register holds a candidate PC per thread.

Scalar branch control. Without such per-lane program counters (PCs), SIMD hardware is unable to track execution paths of different threads. Instead, all threads must follow the same path of execution, inactive threads' lanes are masked out. In order to handle divergence and lane masks, we use the technique by Lorie and Strong [15] that inserts JOIN nodes into the code at which the activity of all lanes is tested; if (through e.g. branching) no active threads remain, the PC of an inactive thread is picked up as the warps' sole PC. In the following, we refer to PTX code with these two additions as *vector-ready PTX* (vrPTX).

B. Back-End

With parameterized SIMD execution, differently-sized warps all execute the same vrPTX code. Nevertheless, executing each warp on its own SIMD core would often underutilize the hardware and prevent us from hiding latencies, one of the bedrocks of TOC. Therefore, we propose two hardware modifications in SIMD systems:

Partitioning. Following our execution models, executing a single warp of size less than the number of SIMD lanes would leave many SIMD lanes in wide-SIMD systems unoccupied. Since a lane's execution only depends on its metadata, we use this fact to *pack* multiple warps' data into the same SIMD registers, increasing vector length as necessary and refer to the packed warps as one *partition*. Since warps in a partition

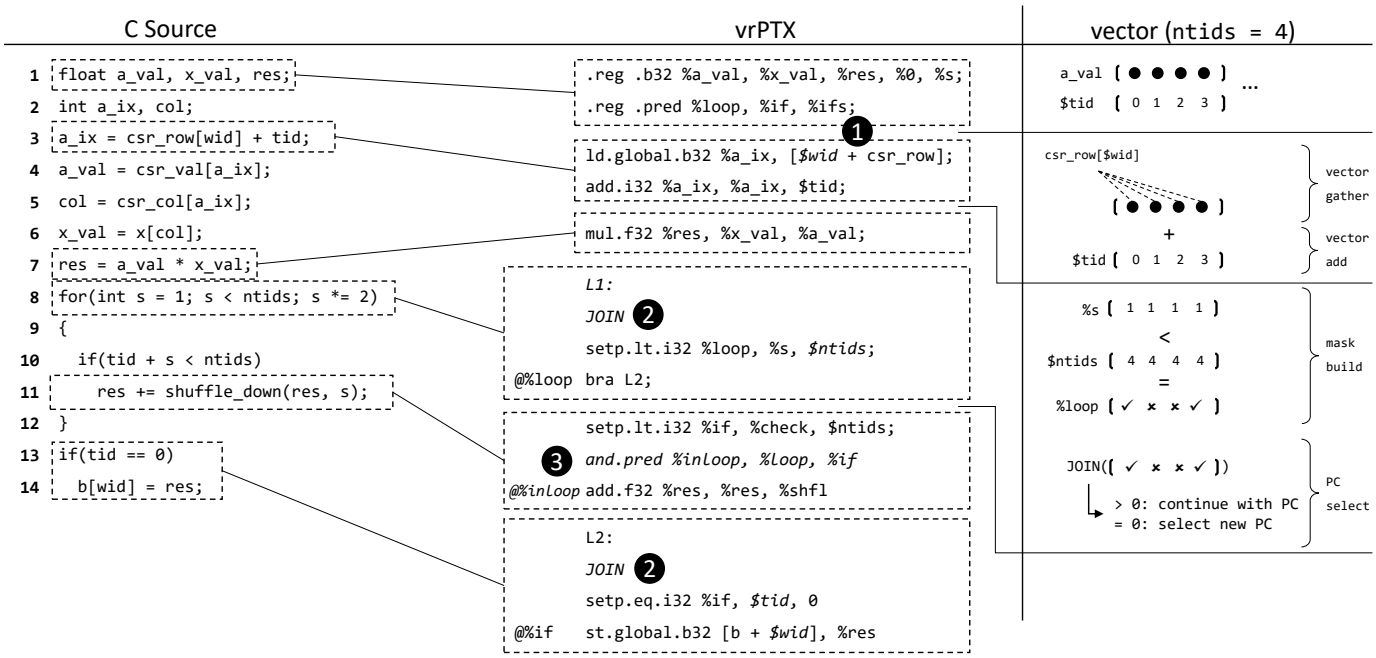

Figure 1: Our solution integrates a modified compilation pass from CUDA to *vector-ready PTX*, that enables execution of SIMT code on SIMD architectures by concatenating threads' registers into SIMD registers. There, we ❶ add special registers for warp and thread indices, ❷ insert synchronization point for PC selection after thread divergence ❸ turn predicated into masked instructions.

may diverge at branches, we propose the following: in the compile phase, we unroll the vrPTX source for multiple values of `ntids` and count the resulting number of instructions. We then group the possible values of `ntids` into *buckets* according to the difference in their number of instructions; warps that fall into the same bucket may be put into the same partition, hoping that they would behave similarly. We may repeat that experiment for random outcomes of branch instructions to improve our estimate.

Vector code issue and multiplexing. Even with packing, we still need a method to handle warps of drastically different sizes and diverging control flow. Wrapping warps into SMT threads is not an option: with larger SIMD registers, context switches become prohibitively expensive. Instead, we propose a static *partition multiplex* scheme that uses a register renaming unit to execute multiple partitions at once. We visualize our idea in Fig. 2b: As long as `vl` is less than the number of SIMD lanes, all partitions require the same number of SIMD registers. Hence, we can proceed analogous to SIMT processors and divide the register file according to the partitions. In our example in Fig. 2b, partition 0 (packing warps 0 and 2) uses physical SIMD registers `v0` through `v4`. Using this *partitioned* register table (PRT), the incoming vrPTX instructions can be mapped conflict-free to physical SIMD registers. After the mapping, a lookup table performs the 1:1 translation from vrPTX to SIMD vector instructions and sets the runtime `vl` accordingly. After renaming, there are no conflicts between streams from different partitions, so all are multiplexed into the same instruction buffer. Through a vector instruction scheduler, this method results automatically hides latencies.

C. Integration into SX-Aurora

As a practical example, we consider the modification of a vector processor design that is already on the market: NEC's SX-Aurora TSUBASA [21]. Aurora's PCIe cards offers up to 10 cores at 1.6 GHz with up to 3.07 TFLOPs in double precision mode. All cores share 16 MB last-level cache (LLC) and 48 GB HBM2 memory with a peak bandwidth of 1.53 TB/s. Each Aurora core includes a scalar processor (SPU) and a vector processing unit (VPU) (with several vector pipeline processors (VPP)) as in Fig. 2a. The VPU uses register renaming to execute vector instructions out-of-order by dispatching them to vector data and mask registers as well as FMA/ALU execution units. In our design, we focus on the VPU exclusively and use only the instruction fetching capabilities of the SPU. Fig. 3 depicts our modifications: Compared to the original VPU, we pull the register renaming unit before the vector instruction buffer, since we do not use out of order execution *within* partitions. Instead, vrPTX instructions are loaded for multiple partitions and multiplexed into a single vector instruction stream. Hence, we offer a comparatively cheap way to leverage existing IP for efficient irregular processing.

V. Simulation Results

Due to a lack of details regarding NEC's SX-Aurora, we used the available ISA documentation in order to build a

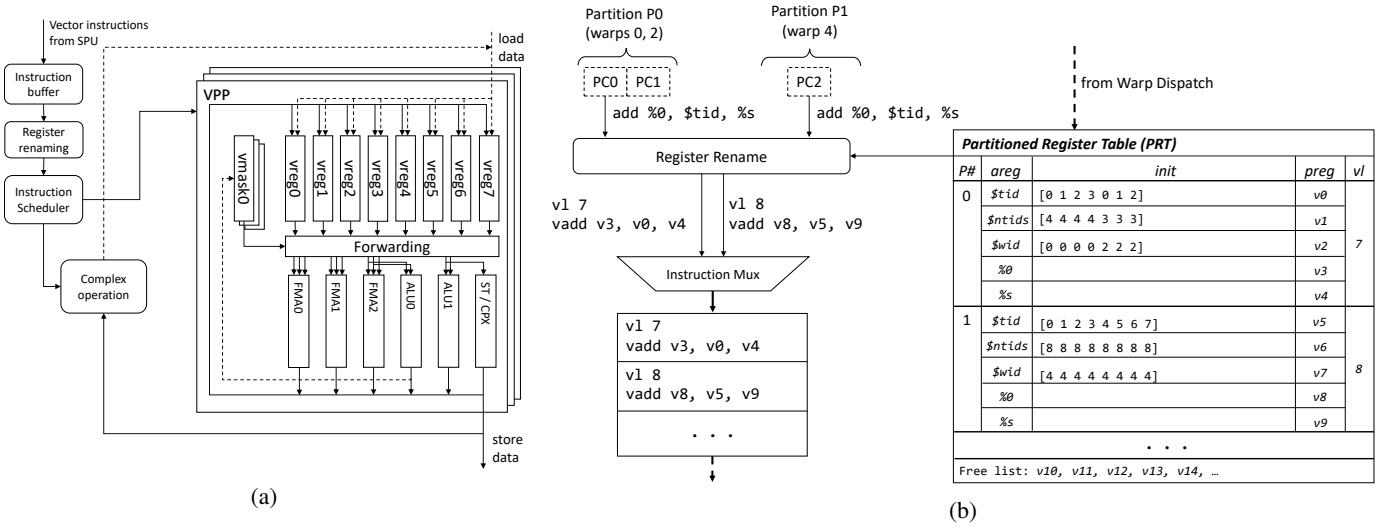

Figure 2: **(a)** SX-Aurora's vector unit offers 8 VPPs and an out-of-order scheduler that resolves hazards through register renaming. **(b)** We propose to re-purpose the register renaming unit to multiplex vrPTX instruction streams from multiple warps into one single stream of vector instructions.

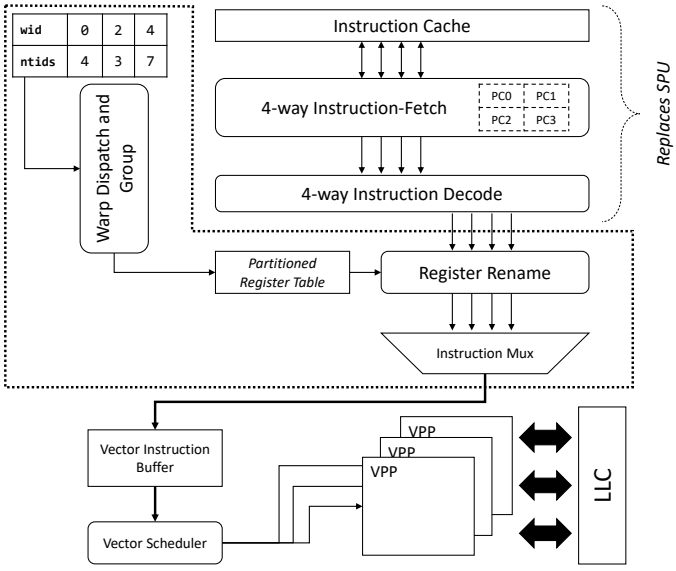

Figure 3: Integration of our proposed warp multiplexer (Figure 2a) into SX-Aurora's vector core. We move the instruction buffer behind it and remove the SPU, except its instruction fetch unit.

model following Fig. 2a and simulate it using the SimPy framework [16]. We express all latencies in terms of multiples of a simple arithmetic vector operation that takes 1 cycle, any complex resp. store operation's latency is the active vector length. We simulate one core of the vector processor with the same number of memory controllers and ports as Aurora. Our simulation consumes the generated PTX from our SPMV example code (see Figure 1). We input multiple matrices from the SuiteSparse Matrix Collection's [6] *linear programming* category, since these matrices often suffer from data irregularity (i.e. different row lengths). This test is meant to showcase two things: First, allowing more partitions (t – also the number of required instruction fetch units) per core results in a larger vector instruction buffer which leads to better utilization of the execution units and thus less empty cycles. Second, packing can save instructions by batching warps – again, we expect less time to termination.

Figs. 4 presents simulation results for two matrices that are representative for the test set: lp22 ($2,958 \times 16,392; 68,512$ nz – first row) and mycielskian11 ($1,535 \times 1,535; 134,710$ nz – second row). Their row length distributions, and thus warp size distributions, are visualized in Figs. 4a. Figs. 4b support our first hypothesis: Independent from the packing setup, more slots result in consistently less cycles being used. More slots lead to more and potentially different simultaneous instructions in the instruction buffer which in turn may be executed in parallel (pending execution unit availability). Furthermore, a looser threshold for packing (permitting higher warp size variations inside a partition) further reduces the total number of cycles spent. In Figs. 4c, we visualize the execution unit utilization in the same experiments: Again, both more partitions as well as looser packing thresholds increase the utilization until reaching a plateau.

Lastly, we point to the error bars for $t = 4$ (Figs. 4b, 4c): for each parameter setting, we ran the simulation 100 times, every time with a random order of the input matrix' rows, and plot the resulting error bars in both cycle and utilization plot. We point out that although the variation is relatively large, at times negating the benefit of packing entirely, the average line (black) tends strongly towards the better region (lower cycles, higher utilization). This indicates that a smaller number of outliers is responsible for such failures. Since we currently do not support work stealing or dynamic allocation, these outliers directly correspond to certain row orders. We leave a closer

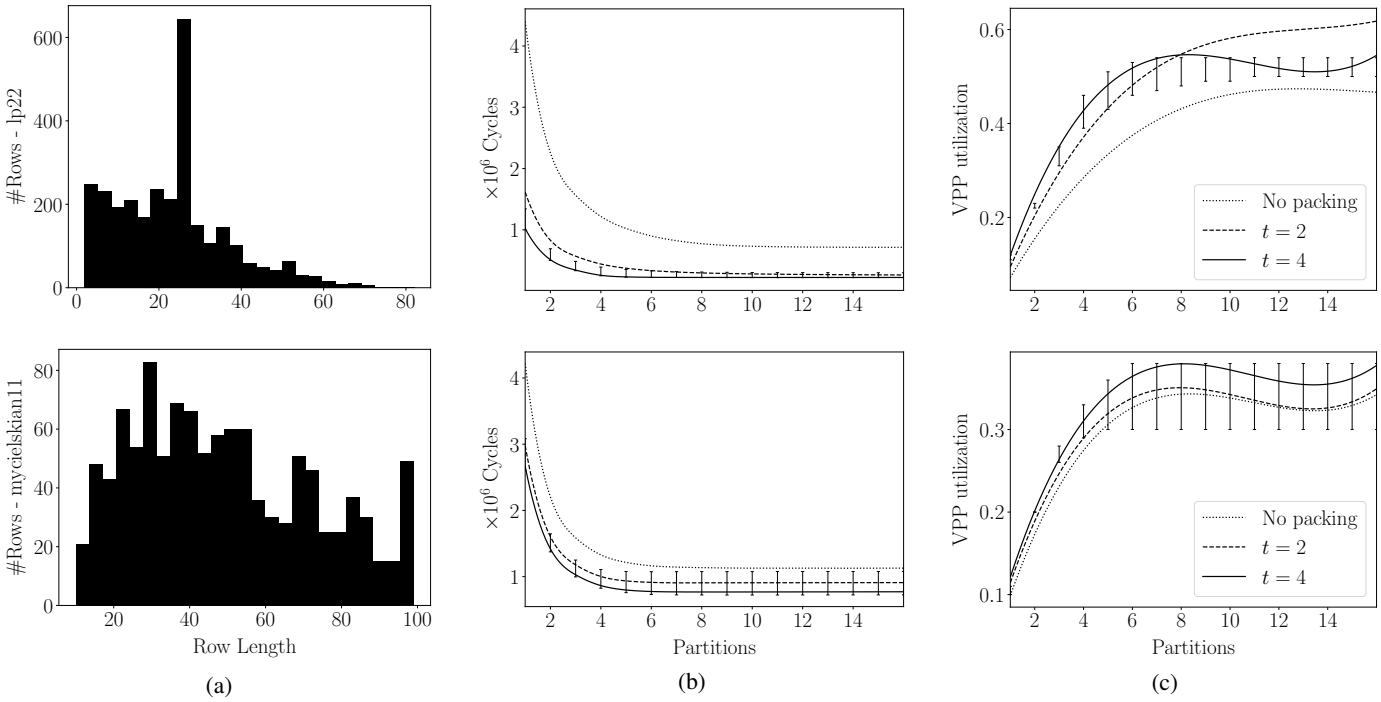

Figure 4: Results from a model-driven simulation of our setup running an SpMV kernel for two sparse matrices (upper row: lp22, lower row: mycielskian11) with one order of magnitude irregularity. The results show that our architecture modifications help to make efficient use of the execution units and hide latencies.

investigation for future work.

VI. Conclusion

In this paper, we briefly discussed how existing vector accelerators could potentially be improved: Embedded in a full stack of programming model, compiler and architecture changes, they can mark a first step towards throughput oriented computing with full support for irregular workloads while retaining the familiar CUDA programming model. Having so far only validated our ideas using a model-based simulation, we plan to follow that up with a cycle-accurate simulation of the proposed system.

References

[1] A. Armejach, H. Caminal, J. M. Cebrian, R. González-Alberquilla, C. Adeniyi-Jones, M. Valero, M. Casas, and M. Moretó. Stencil codes on a vector length agnostic architecture. In *PACT'18*.

[2] K.-C. Chen and C.-H. Chen. Enabling SIMT Execution Model on Homogeneous Multi-Core System. *ACM TACO*, 15(1):1–26, 2018.

[3] Z. Chen and D. Kaeli. Balancing Scalar and Vector Execution on GPU Architectures. In *IPDPS'16*.

[4] N. Clark, A. Hormati, S. Yehia, S. Mahlke, and K. Flautner. Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping. In *HPCA'07*.

[5] S. Collange. Simty: Generalized SIMT execution on RISC-V. 2017.

[6] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM TOMS*, 38:1–25, 2011.

[7] G. Diamos, A. Kerr, and M. Kesavan. Translating GPU Binaries to Tiered SIMD Architectures with Ocelot. Technical Report GIT-CERCS-09-01.

[8] S. Frey, G. Reina, and T. Ertl. SIMT Microscheduling: Reducing Thread Stalling in Divergent Iterative Algorithms. In *PDP'20*.

[9] W. W. L. Fung and T. M. Aamodt. Thread block compaction for efficient SIMT control flow. In *HPCA'11*.

[10] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware. *ACM TACO*, 6(2):1–37, 2009.

[11] M. Garland and D. B. Kirk. Understanding throughput-oriented architectures. *Communications of the ACM*, 53(11):58–66, 2010.

[12] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA graph algorithms at maximum warp. *ACM SIGPLAN Notices*, 46(8):267–276, 2011.

[13] M. Huzaifa, J. Alsop, A. Mahmoud, G. Salvador, M. D. Sinclair, and S. V. Adve. Inter-kernel Reuse-aware Thread Block Scheduling. *ACM TACO*, 17(3):1–27, 2020.

[14] J. Kim, J. Cha, J. J. K. Park, D. Jeon, and Y. Park. Improving GPU Multitasking Efficiency Using Dynamic Resource Sharing. *IEEE Computer Architecture Letters*, 18(1):1–5, 2019.

[15] R. A. Lorie and H. R. Strong Jr. Method for conditional branch execution in SIMD vector processors, U.S. Patent US4435758A, Mar. 1984.

[16] K. G. Müller and T. Vignaux. Simpy. https://github.com/cristiklein/simpy.

[17] D. Nuzman, S. Dyshel, E. Rohou, I. Rosen, K. Williams, D. Yuste, A. Cohen, and A. Zaks. Vapor SIMD: Auto-vectorize once, run everywhere. In *CGO'11*.

[18] M. Pharr and W. R. Mark. Ispc: A SPMD compiler for high-performance CPU programming. In *Inpar'12*.

[19] M. Stanic, O. Palomar, T. Hayes, I. Ratkovic, A. Cristal, O. Unsal, and M. Valero. An Integrated Vector-Scalar Design on an In-Order ARM Core. *ACM TACO*, 14(2):1–26, 2017.

[20] A. Tino, C. Collange, and A. Seznec. SIMT-X: Extending Single-Instruction Multi-Threading to Out-of-Order Cores. *ACM TACO*, 17(2):15:1–15:23, 2020.

[21] Y. Yamada and S. Momose. Vector engine processor of NEC's brand-new supercomputer SX-Aurora TSUBASA. In *HotChips'18*.

[22] Y. Yang, P. Xiang, M. Mantor, N. Rubin, L. Hsu, Q. Dong, and H. Zhou. A Case for a Flexible Scalar Unit in SIMT Architecture. In *IPDPS'14*.