RE-VETORIZAÇÃO DE CHAMADAS DE FUNÇÃO

RUBENS EMILIO ALVES MOREIRA

RE-VETORIZAÇÃO DE CHAMADAS DE FUNÇÃO

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Fernando Magno Quinto Pereira

Belo Horizonte

Setembro de 2016

RUBENS EMILIO ALVES MOREIRA

FUNCTION CALL RE-VECTORIZATION

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Fernando Magno Quinto Pereira

Belo Horizonte

September 2016

Emilio Alves Moreira, Rubens

Function Call Re-Vectorization / Rubens Emilio Alves Moreira. — Belo Horizonte, 2016
xxviii, 71 f. : il. ; 29cm

Dissertação (mestrado) — Federal University of Minas Gerais
Orientador: Fernando Magno Quinto Pereira

1. Computação — Teses. 2. Compiladores — Teses.
I. Fernando Magno Quintão Pereira. II. Re-Vetorização de Chamadas de Função.

[Folha de Aprovação]

Quando a secretaria do Curso fornecer esta folha,
ela deve ser digitalizada e armazenada no disco em formato gráfico.

Se você estiver usando o `pdflatex`,
armazene o arquivo preferencialmente em formato PNG
(o formato JPEG é pior neste caso).

Se você estiver usando o `latex` (não o `pdflatex`),
terá que converter o arquivo gráfico para o formato EPS.

Em seguida, acrescente a opção `approval={`*nome do arquivo*`}`
ao comando `\ppgccufmg`.

Se a imagem da folha de aprovação precisar ser ajustada, use:
`approval=[`*ajuste*`][`*escala*`]{`*nome do arquivo*`}`
onde *ajuste* uma distância para deslocar a imagem para baixo
e *escala* é um fator de escala para a imagem. Por exemplo:
`approval=[-2cm][0.9]{`*nome do arquivo*`}`
desloca a imagem 2cm para cima e a escala em 90%.

I dedicate this work to my mom, family, friends, to every single person that symbiotically keeps UFMG's Department of Computer Science the great institution it is, and specially to my dear professor Fernando Magno Quintão Pereira.

I am more than conversant with how hard I have tried to get to this point.

I am more than convinced that your guidance set the stage for this success.

I am more than aware of my otherwise unfortunate failure.

Acknowledgments

Agradeço à minha mãe por cada segundo do meu sempre que dedica cuidando de mim. E por cada segundo que já teve que estar no trabalho, seja na escolinha, no hospital João XXIII ou na casa de paciente – em plantões que, do que guardo comigo, já chegaram a mais 72 horas seguidas! Tudo para garantir a excelente qualidade de vida que sempre tivemos, além de me permitir tempo para minhas coisas de escola. A meu pai, por ter sido bom com tantas pessoas; pelos saudosos momentos de presença; pela insistente permanência desgastante; por ter sido tudo que teve de ser – sem acasos.

A meus pais pelo carinho imensurável. E eterno.

À minha família, por tudo que me deram; por tudo que me fizeram aprender.

To the magnificent persistence of my dearest professor Fernando Magno Quintão Pereira, to whom I owe each opportunity I have been blessed with in the past two years – and, take note, perhaps even less! For the multiple times you did not let me completely lose myself, Thank you, sir!

To my professors from UFMG's Department of Computer Science and the whole team that works together to keep this institution the great environment for learning it is. Special thanks to my dear professor Wagner Meira Jr.; to Sônia Borges (mother of all students), Sheilla, Stella, Maristela, and everyone from the PPGCC team. To my dear professors and friends from UFMG's Music school, that always had the doors opened to receive me whenever I needed the good ol' bits of fun.

To my beloved friends – not at all excluding the aforementioned ones –, your presence summed up to who I am, indeed some more than others, but every single one of you is entitled a share of my achievements. Special thanks to those I have spent the last few years with and that at times took better care of me than I myself, Péricles Oliveira, Larissa Leijôto, Mívian Marques, Marcos Felipe, Max Lima, Júlio Albinati, Camila Araújo, Gabriel Poesia, Rodrigo Caetano, Fernando Mussel, Elverton Fazzion, Osvaldo Morais, Vinícius Dias, Denise Britto, Samuel Evangelista, Paulo Bicalho, Luciana Lourdes, Marcus Rodrigues, Pedro Ramos, Marcos Almeida, Gleison Souza, Vitor Paisante, Michael Frank, Florian Brendner, Paschalis Mpeis, Douglas Teixeira,

Victor Campos, Henrique Nazaré, Demontiê Júnior, Carolina Medeiros, Gina Nogueira, Filipe Arcanjo, Bruno Coutinho, Diogo Rennó, Izabela Karennina, Guilherme Sad, Lucas Moreira, Hudson de Almeida, Edson Júnior, Vinícius Garcia, Pedro Dias, Solange Oliveira, Humberto Fialho, Caio Ianelis, Henrique Lourenço, Murilo Barbosa, Maurício Veloso, Fernando Endo, Imane AlSaghira, Aswin Sridharan, Jake Enstrom, Antoine Froger, Spurrya Jaggi, Leo Souza, Adrian Martinu, Alex Meyer, David Sena, Guilherme Ferrari, Talita Ribeiro, Bruna Ziviani, Amanda Abreu, Mara Campelo, Lucas Matos, Rodrigo Chaves, Thales Linke, Débora Campos, Sandro Pessutti, Gabriel Ferraz, to all my friends from my undergrad course in computer science, from UFMG's Compilers Lab, LBS, Speed, from Colégio Nossa Senhora das Dores, and from the Orchestre Universitaire de Rennes.

Para mí caro amigo Miguel Angel Alfredo Pérez Rueda, con quien aprendí tantas cosas de esa lengua genial que es el español. Estoy seguro de que aún nos encontraremos para platicarmos bastante y para que yo no más te grite sin razón.

My sincere apologies to those I have conceded unfortunate moments of arrogance – regardless of how worded, irresponsibly deceiving, or mischievously silent the means of utterance.

A thankful nod to my bleeding hearts, that get me up in the morning, put me to bed, and save my life every now and then, Billie Holiday, Cartola, Prokofiev, Nina Simone, David Bowie, Liszt, Elza Soares, Miles Davis, Bill Evans, Villa Lobos, Tchaikovsky, Amy Winehouse, Charles Mingus, Shostakovich, Coleman Hawkings, Leo Ornstein, Poulenc, Stravinksy, Cezar Franck, Mozart.

Last, but not least, this little poem I wrote for you all, this mystique amalgam of wondrous people that continuously flies in and out of my life, filling out the gaps with sheer joy and genuine living. It is entitled *Borboleta*. To my non-portuguese speaker friends, I hope one day I will know your language well enough to be able to pick the proper words to translate what I meant with this poem.

Borboleta

Inspirado em *Borboleta* (*Sommerfugl*), opus 43 número 1, de Edvard Grieg.

Borboleta criou em mim asas pela mão.
E pousou.

Alazão na travessia do rio fez meandro com o peso do passo.
Pôs-se a galope, penseroso. E esqueceu.

Levantou em mim alma pelas asas.
Rodopiou com alegria. Alegrou todo canto. Fez esquecer todo o rio.
E levitou.

Cria leito na esperança pelo repouso da borboleta.
Conveceu-se da eternidade! Demagogo... E amoleceu.

Abriu em mim as mãos pela alma.
Tomou tempo do futuro.

Demorou a pousar.

E planou.

Cada fibra do corpo soluçava latejo.
Prumou-se à ideia, penoso.
– Alazão, é medo!

O vôo seguinte foi alto.

Borboleta deu de bruços com a sorte:
– O vento é quem me sopra pro norte, mas a razão da alegria é você.

Põe-se então o rio a encher de pressa.

Alazão perde sorte à outra peça:
– Voltarás pra me ver?

Borboleta criou em mim alma alada.
De encanto, levou todo o rio.

Deu de asas com o vento.

“ stars, hide your fires; life trickles in everywhere
risca o chão com vara de marmelo; seria destino
upon flights across the ocean; over pools of Those memories
no more free steps to heaven
locuta es de triuio

this should be part none! what difference does it make
keep gawking; tip; take a nap; fall and snap
culpa rubet uultus meus
time and space do not exist
we should have proceeded no further in those business

amici, amici... lesser, greater; not so happy, yet much happier
neither this, nor that; neither that, –
aut tace aut loquare meliora silentio
the bleeding hearts –, and the artists –, make their stand
and there it goes again, spinning, weaving new patterns

my poor heart is sentimental, not made of wood
wake duncan with thy knocking; would not we thou couldst
triuium... triuium...
e o azul: mar alto de por-dizeres... virado de por-fazeres... à deriva
never treats me... sweet and gentle...

semper oracula mentiantur... ontem, dor eterna; hoje, esquecimento total
lies, lies... a tissue of lies; all of it
lux aeterna luceat eis domine... donna eis requiem...
i've got it bad... and that ain't good...
not an answer; not even a laugh, Don't talk like that –

it is hard to make meaning out of oneself – amid some many changes
ah, corra e olha o céu...
it is hard not to talk about that which lies constantly in mind
revolução... tem é que ajudar; se envolver; mudança vem é de dentro
it's so hard for us to really be

e, de repente, sertão: saudade de tudo ”
()

Resumo

Linguagens SPMD para arquiteturas SIMD, como C para CUDA, OpenCL e ISPC contribuíram para melhorar a programabilidade de aceleradores SIMD e placas de processamento gráfico. No entanto, linguagens SPMD ainda não disponibilizam ao programador toda a flexibilidade que se pode obter a partir de programação SIMD explícita. A fim de contornar esta falha de expressividade, preservando a abstração SPMD, introduzimos a noção de Call Re-Vectorization (CREV). CREV permite que o programador altere a dimensão da vetorização durante a execução de um kernel SPMD, e o faz por meio de uma chamada aninhada de kernel. CREV provê uma abstração similar àquela oferecida pelo conceito de paralelismo dinâmico: é possível invocar um kernel dentro de outro kernel. Nossa abordagem reduz os custos associados a esse processo. Neste trabalho, apresentamos as definições formais de CREV, além de sua implementação no compilador ISPC. Para validar nossa abordagem, implementamos uma série de algoritmos clássicos explorando o conceito de Call Re-Vectorization. Tais algoritmos incluem casamento de padrão, busca em profundidade e Bellman-Ford, e foram implementados com CREV sem muito esforço. Uma vez compilados usando ISPC para gerar instruções vetoriais de máquinas Intel, nossas implementações são tão eficientes quanto soluções de estado-da-arte, sendo, em geral, mais simples de se programar. Por exemplo, nossa implementação simples de casamento de padrão atinge speedup de 12% sobre o algoritmo Knuth-Morris-Pratt.

Palavras-chave: Vetorização, Compiladores, SIMD.

Abstract

SPMD programming languages for SIMD hardware such as C for CUDA, OpenCL, or ISPC have contributed to increase the programmability of SIMD accelerators and graphics processing units. However, SPMD languages still lack the flexibility offered by low-level SIMD programming on explicit vectors. To close this expressiveness gap while preserving the SPMD abstraction, this dissertation introduces the notion of Call Re-Vectorization (CREV). CREV allows changing the dimension of vectorization during the execution of an SPMD kernel, and exposes it as a nested parallel kernel call. CREV affords a programmability close to dynamic parallelism, a feature that allows the invocation of kernels from inside kernels, but at much lower cost. In this work, we present a formal semantics of CREV, and an implementation of it on the ISPC compiler. To validate our idea, we have used CREV to implement some classic algorithms, including string matching, depth first search and Bellman-Ford, with minimum effort. These algorithms, once compiled by ISPC to Intel-based vector instructions, are as fast as state-of-the-art implementations, yet much simpler. As an example, our straightforward implementation of string matching beats the Knuth-Morris-Pratt algorithm by 12%.

Keywords: Vectorization, compilers, SIMD.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

New hardware asks for new programming idioms. As an example, the appearance of general purpose Graphics Processing Units (GPUs) has led to a revolution in programming: C for CUDA [Sanders and Kandrot, 2010], OpenCL [Munshi et al., 2011], ISPC [Pharr and Mark, 2012], and PyCuda [Garland and Kirk, 2010, Nickolls and Dally, 2010] are among languages designed to unscramble high-end programming for state-of-the-art accelerators. These Multi-Threaded (MT) languages let programmers express computations as single kernels executed by many threads. They target architectures that combine SIMD and multi-threaded execution, like GPUs and multi-core CPUs with vector instructions. Developers are thus spurred into exploiting massive computational power from a clear coding perspective – milder at hardware specifics awareness. While abstracting away low-level primitives springs simpler code, it also gives rise to flexibility and composability constraints. For instance, given a matrix of threads, one may either pack such processing units (threads) into context-dependent rows or columns. This packaging, or parallel dimension, is not unoften fixed throughout the execution of a kernel[1], in such wise sifting functions that may be invoked by the fixed group of threads. Moreover, threads may also be suspended and resumed due to thread-dependent control flow, which makes up for yet another filter on functions that do not comply with divergence.

GPUs and so-called vector machines – including CPUs with vector processing capabilities – fall into a special classification of computer architectures and parallel programming: Flynn's taxonomy. Proposed in 1966 by Michael J. Flynn [Flynn, 1972], it groups architectures by their level of parallelism: single-thread, multi-thread, and multi-program; and by their memory organization, i.e., having processors with a single

[1] Kernels are functions executed at the device-side (GPUs), as opposed to regular functions, carried on at the host-side.

or multiple data streams. For discussing our solution, it is ideal to bear in mind the following two settings: SIMD and SIMT. An **SIMD** architecture consists of a *Single Instruction* stream read, in lock-step, by a vector of n units, each using a different data source from the *Multiple Data* streams. The **SIMT** organization is a *Single Instruction* source executed by *Multiple Threads*, say n, and each thread holding an SIMD lane of length m, i.e., every instruction runs $n \times m$ times. By times we shall refer to this latter model as either SIMT or multi-threaded (MT) programming.

Most languages that target hardware accelerators fix the parallel dimension along which threads are packed into SIMD vectors, or GPU warps[2], for the whole duration of a kernel call. This affects device-side library functions, which cannot assume any particular organization of parallelism nor thread activity. To further grasp how suitable it is to make such an assumption, pay heed to the fact that many hand-tuned libraries [Catanzaro, 2012, NVIDIA, 2016] provide functions that rely on having all threads enabled within a warp: (i) to use such functions, one must assure warp-wise control flow uniformity; (ii) using high-end code is profitable in terms of correctness, efficiency and to avoid replicate work. Developers circumvent the constraints of pure SIMT programming in two ways: via *warp-synchronous programming*, or via *dynamic parallelism*. In the first case, the programmer profits from the fact that threads in an accelerator are grouped into warps to achieve direct thread communication without synchronization or memory sharing. Yet, warp-synchronous coding is not easily composable with classic multi-thread (MT) programming: developers must ensure that every thread within a warp participates in each collective operation; e.g., the CUDA `__shfl` function has undefined behavior when reading data from an inactive thread. Such task is otherwise unsubstantial in plain MT programming, for thread divergence control is put out of the hands of the programmer. Consequently, MT code with control flow divergences may not call warp-synchronous functions from within its divergent regions.

Regarding the second approach, CUDA's *dynamic parallelism* along with OpenCL's device-side *enqueue* consists of the ability to create a new group of threads from within threads already in flight [Yang and Zhou, 2014], thus conferring developers the opportunity to implement strikingly elegant algorithms [Merrill and Grimshaw, 2011]. However, this construct is too hefty for our simpler purpose of re-activating threads within a warp. For instance, invoking new threads from within a thread in CUDA involves the global scheduling of a new grid of threads [Jones, 2014], a very expensive event. In short, currently, either we have

[2]Following the NVIDIA jargon, we shall call groups of threads that execute in lock-step a *warp*.

the programmability and elegance of the multi-threaded model, or the efficiency of warp-synchronous programming, but not both.

Our goal is to allow the composability of SIMD and SIMT through a programming construct syntactically similar to dynamic parallelism. To this end, we introduce the notion of *Call Re-Vectorization* (CREV), and show how to implement it efficiently in a state-of-the-art vector compiler. CREV is a programming idiom that modifies function calls. Functions marked with the **crev** tag, henceforth called *r-functions*, are executed by all the threads available in an SIMD unit. This implies a context switch: to run an r-function, the runtime must change the state of all the threads, including those inactive due to previously divergent control flows. Upon completion, the runtime returns those workers back to their previous state, in the same way a function call is handled. Thus, we achieve a new level of recursion, in which threads can spawn new threads in a stack-based fashion. However, contrary to traditional dynamic parallelism, CREV uses only the accelerator's local memory (registers and call stack) to save thread states; hence, it is cheaper.

To validate our ideas, we have implemented them in ISPC[3] [Brodman et al., 2014, Pharr and Mark, 2012]. ISPC is a programming language, plus its companion compiler. This compiler produces industrial quality code for SIMD units such as Intel Streaming SIMD Extensions (SSE), Intel Advanced Vector Extensions (AVX) including AVX-512 for Xeon Phi accelerators [Sodani et al., 2016], or ARM NEON. We chose to implement CREV in ISPC because this framework provides the only modern SIMT-to-SIMD translator that, to the best of our knowledge, supports the notion of *unmasked* or *everywhere* blocks [Pharr and Mark, 2012]: the ability to activate – in a new context – threads that are idle due to divergences. This feature is a requirement of CREV. We have re-written some benchmarks available in ISPC to use CREV, as well as incorporated some of our own to compose a suite of CREV tests. We show that these implementations are as efficient as warp-synchronous versions of them, and as clear and elegant as if the had been implemented using dynamic parallelism. And this extra efficiency does not imply a loss of programmability. On the contrary, CREV often leads to more concise programs. In addition to the ISPC implementation of CREV, we have also built a small interpreter for an SIMD-like programming language, which better demonstrates the semantics of our new construct.

Summary of our Contributions. The key contribution of this thesis is the notion of function call re-vectorization, which comes out of the observation that it is possible to capitalize on divergent threads to help speed up the work of active threads. We

[3]The Intel SPMD Program Compiler (ISPC) is available at `https://github.com/ispc/`

explain the concept of CREV through examples, a formal semantics, and an industrial quality implementation:

- **Examples:** Section 2 shows examples of algorithms that benefit from our notion of Call Re-Vectorization. Further examples are discussed in Section 5.
- **Semantics:** Section 3.2 formalizes the semantics of μ-SIMD, a low-level instruction set sufficient to implement CREV. We have written a Prolog interpreter to validate the semantics. This interpreter made it easy to prototype different implementations of CREV, until we had a design we could graft into a state-of-the-art compiler.
- **Translation:** Section 3.3 describes the translation of the high-level "**crev**" keyword into the low-level representation. Core properties of the final, low-level code, as produced by the translator, are listed in Section 3.4.
- **Evaluation:** Section 5 provides an empirical evaluation of our implementation. To perform this evaluation, we have implemented some algorithms, which are faster and cleaner than their original versions without CREV.

Published Papers We now present the works published throughout this Master's course. The initial work, related to Return Oriented Programming attack prevention, was crucial to gaining experience with one of nowadays' main crowd-source compilation framework, the LLVM infrastructure; it was as well a means of better understanding compilation techniques and optimizations. We have thus heavily applied such learnings in this project.

Return Oriented Programming This project targeted a well–known vulnerability exploit named ROP-attack. Return-oriented programming is a technique attackers employ to take control of the execution of a program, and eventually of the entire host machine. The exploit begins in the identification of flaws in the program, such as buffer overflows, and thus input extraneous data into the read-only section of memory. By chaining a series of indirect jumps interleaved with instructions that have little to no side-effects, the attacker forces the foreign data placed in the read-only memory to be processed as if executable memory. To bestow such exploit in a program is rather an artsy process: the attacker must analyze many corner-cases to find the ones useful for conveying the exploit.

Being challenging does not mean being impossible to do, and it is known that even government agents have used such technique to acquire top secret information [Kushner, 2013]. Whenever an attack takes place, the frequency of indirect branches seen at the processor increases significantly [Tymburibá et al., 2016]. Our

solution lies in deriving tight frequency thresholds for applications: we statically traverse the control flow graph (CFG) of programs, and search for the path of up to a fixed number of instructions with the highest density of indirect branches. Despite the NP-hardness of determining a maximum path, our static analysis operates in a feasible runtime, as we limit the path's maximum length.

As outcome of this project, we had a paper accepted at the Brazilian Symposium on Information and Computational Systems Security (SBSEG'15) [Moreira et al., 2015]; a dynamic detector for ROP attacks, namely Rip-Rop Deducer [Tymburibá et al., 2015]; and we have put up a website [Moreira and Tymburibá, 2016] with a static analysis to infer frequency thresholds for indirect branches in applications. Finally, we had another work published at the International Symposium of Code Generation and Optimization (CGO'16) [Tymburibá et al., 2016], and were awarded the Golden Medal (1st prize) at CGO's Student Research Competition (CGO-SRC'16) [Spink, 2016]. I thank my friends Mateus Tymburibá and Fernando Magno for being so patient and supportive throughout this work.

Function Call Re-Vectorization In this work we aimed at the capabilities of warp-synchronous programming in conjunction with the simplicity of dynamic parallelism. Warp-synchronous programming is known to give programmers a high-level interface with SIMD native instructions from vector processing machines. Besides the possibility of fine-tuning applications, warp-synchronous code can easily become a nightmare, even for seasoned developers [Moreira et al., 2017]. We implement, on top of an industrial SIMT compiler, the idiom **crev**, allowing programmers to call warp-synchronous procedures even within divergent regions. We further detail this project in the remaining sections of this thesis.

We published an initial work at Brazilian Symposium on Programming Languages (SBLP'16) [Moreira et al., 2016], in which we define the semantics of *everywhere* blocks in the SIMD world. The concept of everywhere blocks is key to developing CREV, as it allows one to temporarily re-enable threads within a warp. We later publish our contribution, including the **crev** implementation on top of Intel's SPMD compiler, ISPC. Our paper, entitled Function Call Re-Vectorization, was published at the Symposium on Principles and Practice of Parallel Programming (PPoPP'17) [Moreira et al., 2017]. Je remerci bien mes orientateurs Fernando et Sylvain pour l'opportunité de travailler avec eux sur ce project, et aussi de faire trois mois de stage à la France. Là-bas j'ai fait la connaissance de beaucoup de personnes qu'ont fait mon séjours vraiment speciale.

Un grand remercie à mon ami Fernando Akira Endo, et à tous mes amis au centre de recherche INRIA Rennes-Bretagne Atlantique.

Twidd: Twig over RDDs Originally "Twig: An Adaptable and Scalable Distributed FPGrowth" was a work developed during the last two years of my undergraduation and first semester of my Master's course. We proposed a distributed FPGrowth algorithm with dynamic policy for load distribution among computing nodes, with low replication overhead. Our approach partitioned the input database using the FPTree structure from the FPGrowth algorithm. FPTree is a prefix-tree having each node to represent an element from a list, given a transactional database. The interesting point of the FPTree is that it keeps the most frequent elements closer to the root node, thus reducing data replication within its structure. The work, available online[4], was not accepted at the International Parallel and Distributed Programming Symposium of 2015.

During the first semester of my Master's course, the algorithm, initially implemented in C++, was reimplemented in Spark/Scala, by my friend and Computer Science Master's student Vinícius Victor Santos Dias. The implementation, namely Twidd, exploited the benefits of Resilient Distributed Datasets (RDDs) to scalably allow for fault tolerance. Vinícius also implemented a distributed version of the Eclat frequent itemset algorithm, which was used, along with Twidd, to study performance issues in massively parallel applications. He later published the work "Diagnosing Performance Bottlenecks in Massive Data Parallel Programs" at the International Symposium on Cluster, Cloud and Grid Computing (CCGRID'16) [Dias et al., 2016], and granted me the co-authorship on his work. Besides having my entire C++ formation upon this work, I have also gained a lot of technical experience throughout this project, both due to my efforts in implementing Twig, and from the many people either voluntarily involved or dragged into this project. I here take this opportunity to publicly thank all of them for their contributions, and Vinícius for his kind gesture.

[4]Twig: An Adaptable and Scalable Distributed FPGrowth. Work unfortunately not accepted by the committee of IPDPS'15. `https://github.com/rubenseam/dfptree/blob/master/ipdps15.pdf`.

Chapter 2

Overview

The goal of this section is to explain *Warp-Synchronous Programming*, *Dynamic Parallelism* (DP), and our notion of *Call Re-Vectorization* (CREV). To this end, we shall use Algorithm 1 as an example. This program receives a book b_i, plus a pattern p. It then copies out all the lines $l \in b_i$ that match p. Pattern matching is performed by `memcmp`, and memory copying is done by `memcpy`. The book is represented as a matrix of characters; thus, each of its lines, and also the pattern p, is a vector of up to N characters. Algorithm 1 runs in parallel: t_{id} is a thread identifier. Hence, each thread is in charge of matching a line l in b_i against p. In case the match is positive, this thread must copy l to an output matrix b_o. For clarity, we assume a single warp in this example, although the techniques here described can be applied independently to multiple warps. The number of threads that run simultaneously in Algorithm 1 is W, the warp width.

Algorithm 1: SIMD Book Filter and `memcpy` function

```
1 W ← warp size; t_id ← thread index;
2 Function bFilter(mtx b_i, mtx b_o, vec p, int N)
3     for k ← t_id to num_lines(b_i) - 1 step W do
4         l ← b_i[k];
5         if memcmp(l, p, N) == 0 then
6             memcpy(l, b_o[k], N);
7 Function memcpy(str l_src, str l_dest, int N)
8     for i ← 0 to N - 1 do  l_dest[i] ← l_src[i] ;
```

The naive multi-thread implementation of `memcpy` iterates sequentially over the arrays within each thread (Algorithm 1, line 8). This implementation is highly inefficient due to branch divertion, for only threads that step into the `memcpy` function will

be active and working on their own memory copy. Divergence may take place upon branch evaluation, or due to unrelated memory accesses. Branch divergence occurs if the number of iterations N differs across threads. Threads with few iterations would finish the loop earlier and wait for threads with more iterations in order to restore convergence at the end of the loop. Memory divergence also happens as threads within a warp access data in unrelated locations. Such accesses, referred to as *uncoalesced* in the CUDA literature or as *gather/scatter* on SIMD platforms, are bandwidth-inefficient compared to accesses to consecutive elements.

2.1 Warp Synchronous Programming

It is possible to write function `memcpy` in a way that distributes operations on contiguous elements across consecutive threads. Algorithm 2 does it. Function `memcpy_shfl` is aware of the SIMD nature of a warp. Variables are stored as vectors, having each position belonging to a specific thread. Instruction $\texttt{shfl}(v, i)$ allows thread t_{id} to read the value stored in variable v, but in the register space of thread i. This implementation give us an efficient way to copy data between arrays, as copies are distributed evenly between threads, removing most of the branch divergence. Memory divergence is also eliminated as threads of a warp access consecutive elements at each iteration of the loop on line 7.

Nevertheless, this function has an important limitation: it requires all threads in the warp to be active. It cannot safely be called from a point that has potential branch divergence. Indeed, the loop on line 7 would skip elements if some threads were inactive. To support calls to `memcpy_shfl` within divergent regions, we need a way to re-activate threads and put them to work on the copy loop. In addition, the warp-synchronous programming construct is more complex and error-prone than the naive implementation.

Algorithm 2: Warp synchronous `memcpy`

```
1 W ← warp size; t_id ← thread index;
2 Function memcpy_shfl(vec s, vec d, int N)
3     for j ← 0 to W − 1 do
4         d_my ← shfl(d, j);
5         s_my ← shfl(s, j);
6         N_my ← shfl(N, j);
7         for i ← t_id to N_my − 1 step W do
8             d_my[i] ← s_my[i];
```

2.2 Dynamic Parallelism in CUDA

In NVIDIA's CUDA and OpenCL 2.0, dynamic parallelism (DP) is the ability to invoke a new kernel K_2 from within a kernel K_1 [Wang and Yalamanchili, 2014]. In this case, programmers may request a large number of threads, i.e., multiple new warps in multiple thread blocks. As the inner K_2 is a new kernel, all its threads are active upon entry, regardless of branch divergence in K_1. Algorithm 3 shows an implementation of `memcpy` that we could invoke from Algorithm 1 using dynamic parallelism. This algorithm splits, among all the threads in a warp, the work of copying vector s to vector d. Its main advantage is simplicity; its disadvantage is efficiency.

Algorithm 3: Implementation of `memcpy` that could be invoked dynamically from Algorithm 1.

```
1 W ← warp size; t_id ← thread index;
2 Function memcpy_dp(vec s, vec d, int N)
3     for k ← t_id to N − 1 step W do
4         d[k] ← s[k];
```

Wang *et al.* demonstrate that the overhead of a new kernel launch can be as high as one millisecond [Wang and Yalamanchili, 2014]. The new kernel must be scheduled and wait until there are resources available for its execution. Then, the requested number of warps and memory blocks must be allocated before execution starts. For large workloads, the overhead of launching a nested kernel is paid off by the massive data parallelism available in the GPU [DiMarco and Taufer, 2013]. However, for small tasks, this extra cost might degrade performance.

2.3 Call Re-Vectorization

Introducing an inner dimension of parallelism is desirable to implement irregular algorithms such as graph traversal and recursive sorting. Unfortunately, current abstractions based on warp-synchronous programming or Dynamic Parallelism either compromise efficiency or programmability. To solve this conundrum, we introduce Call Re-Vectorization (CREV), a new programming idiom. Syntactically, CREV is akin to CUDA's dynamic parallelism. Semantically, it avoids the cost of scheduling new kernels.

CREV revisits the concept of **everywhere** (also known as **all** or **unmasked**) blocks to temporarily re-enable inactive threads within divergent regions. Such construction was available in programming languages for SIMD machines, such as

Algorithm 4: SIMD Book Filter using CREV

```
1 W ← warp size; t_id ← thread index;
2 Function bFilter(mtx b_i, mtx b_o, vec p, int N)
3     for k ← t_id to num_lines(b_i)-1 step W do
4         l ← b_i[k];
5         if memcmp(l, p, N) = 0 then
6             crev memcpy_crev(l, b_o[k], N);
7 Function memcpy_crev(vec s, vec d, int N)
8     for k ← t_id to N − 1 step W do
9         d[k] ← s[k];
```

C* [Rose and Steele, 1987], MPL (*MasPar Programming Language*) [MasPar, 1992] or POMPC [Hoogvorst et al., 1991] in the late 1980s and early 1990s, and has made a recent comeback in ISPC [Pharr and Mark, 2012]. In these languages, an **everywhere** block is executed by every processing element, regardless of its divergent state. At the end of that block, threads are sent back to their original state.

The **everywhere** block is a low-level construct we employ in the implementation of CREV; however, programmers do not deal with it directly – this is the task of the code generator. Algorithm 4 shows how Algorithm 1 looks like once implemented using CREV. Programmers use the **crev** keyword at line 6 to re-vectorize functions. CREV maintains a stack of thread states to track execution contexts, thus supporting nested calls of r-functions. In terms of performance, a call to a function using the **crev** directive is equivalent to a regular function call – unlike the implementation of dynamic parallelism in CUDA, for instance. Thus, we favour the use of CREV for fine grain nested parallelism. Example 2.3.1 arms the reader with some intuition on how CREV works, yet we explain the nitty-gritties behind the CREV directive in Section 3.

Example 2.3.1 *A function is called with the* **`crev`** *prefix to indicate that every thread, whether enabled or disabled, should execute the function. We address as r-functions the procedures targeted by our* **`crev`** *directive. Every thread should execute the r-function multiple times if multiple enabled threads in the warp call it. For instance, if the warp size is 32 and 7 threads are enabled when the program flow hits line 6 in Algorithm 4, all 32 threads execute* **`memcmp_crev`** *7 times. In each case, the 32 threads temporarily take on the local state of the active thread that they are helping. Once done, these workers all get their local state restored.*

```
void simd_reset(uniform int data[], uniform int length) {
    for (varying int i = programIndex; i < length; i += programCount) {
        data[i] = 0;
    }
}
export void toy(uniform int * uniform data[], uniform int length) {
    if (programIndex % 2 == 0) return;
    for (uniform int i = 0; i < length; i += 2) simd_reset(data[i], length);
}
```

Figure 2.1. Toy example of CREV application. We use this snippet to highlight the main programming issues we try to tackle with CREV. The first issue is to *bridge SIMT-SIMD*, in the sense we must be able to call SIMD-based functions within divergent regions.

2.4 Why CREV?

As shall be presented in the forecoming sections and remaining of this work, CREV is designed as an idiom for SIMT languanges, in order to provide more flexibility for programmers in search for speedups. We endeavor code simplicity in the resulting language extension, so developers can better capitalize on target-hardware computational supply without abdicating their time to understanding yet another concept or struggling to fix new bugs. Our approach relies on the straightforwardness of *dynamic parallelism* whilst its underlying implementation is founded on the rather user-unfriendly *warp synchronous* programming. Let us now go quickly over the building blocks for composing CREV, and then discuss why CREV is useful and on which context it stands as a better solution than works previously proposed in the literature. To assist us in grasping each of the challenges counterposed by our solution, we built a toy example of application for **crev**, depicted in Figure 2.1.

Everywhere Blocks. One of the concerns that arises from vector-based programming is the usage of SIMD within pontentially divergent regions, which is often the case in multithreaded programs. SIMD kernels require all threads within a warp to be enabled, in other words, the control flow must be uniform. Since divergence is one of the main characteristics of a system orchestrating a set of control flow independent threads, there must be a way of cutting through a divergent region and guaranteeing, at least temporarily, the whole warp to be active. Analyzing function `toy` from Figure 2.1, the conditional establish a control divergence by halving the threads into odd- and even-indexed sets – the latter being deactivated upon calling *return*. The subsequent lines of code must all be in compliance with such missing-threads, i.e., they may not

call SIMD functions. If we then try to execute the call to *simd_ reset*, unless originally the programmer's intention, the result would be wrong, as such function depends on having all threads active to *reset* a vector to 0 valued entries.

In our toy example, we simply create a mock condition to enforce divergence, but in the latter sections we show a series of examples in which such dillema may appear. Furthermore, there are also publicly available cases in which one may get entangled amid divergent threads but also need control flow uniformity to call SIMD procedures [NVIDIA, 2016]. The problem of expressing nested SIMD loops in multi-thread style is not new. Some data-parallel programming languages for SIMD computers in the 80's and 90's allow to re-enable temporarily dormant threads. The C* language [Rose and Steele, 1987], the Maspar Programming Language [MasPar, 1992], and the POMPC language [Hoogvorst et al., 1991] incorporate a control flow construct named either everywhere or all to this end. We have re-used these instructions to implement CREV. However, these are low-level primitives: they are not programmer-friendly, nor have any interface with function calls. Using **everywhere** directly is difficult, as this abstraction has no knowledge nor control over the state of dormant threads. CREV, on the contrary, is as easy to use as dynamic parallelism. It manages register saves and restores automatically, relieving the programmer from this task.

Warp-level convergence guarantees. Previous work enforce guarantees on where threads converge after control divergences to make warp-synchronous programming safer. For instance, Pharr *et al.* have proposed the *maximal convergence* guarantee [Pharr and Mark, 2012], and Gaster has proposed a divergence-aware execution model for OpenCL [Gaster, 2014]. CREV goes further by actually *enforcing* convergence at arbitrary program points, allowing warp-synchronous functions to be called from divergent sections. To the best of our knowledge, this is the first attempt to provide developers with such possibility. Considering our toy example from figure 2.1, we simply need to replace the loop with the invocation of `simd_reset` by a call to this function using the **crev** directive. Whenever a variable is tagged *varying*, it holds in fact a vector of values, one per thread; whereas a *uniform* variable holds a single value, equal for all threads. Therefore, notice that the index variable to the matrix `data`, which was the induction variable i, has now been replaced by the `programIndex`. The result is a *varying* pointer, i.e., there is one pointer value from `data` per thread within warp – potentially distinct addresses. To simplify, let us assume the outer dimension of the matrix to equal the number of threads from the warp; but, to ensure the procedure genericity, all that is needed is to make a loop with increment step equal to the warp size.

```
export void toy(uniform int * uniform data[], uniform int length) {
    if (programIndex % 2 == 0) return;
    crev simd_reset(data[programIndex], length);
}
```

Figure 2.2. Application of the **crev** directive within a divergent region.

Grid-level Dynamic Parallelism. Much effort has been spent to reduce the overhead of dynamic parallelism. Alternatives to CUDA Dynamic Parallelism such as DTBL [Wang et al., 2015], Free Launch [Chen and Shen, 2015] and LaPerm [Wang et al., 2016] reduce sub-kernel launch overhead or improve cache locality. By relying on global schedulers, they allow load-balancing between GPU stream multiprocessors. We are not competing with these efforts, because CREV is not an alternative to dynamic parallelism. CREV is a static code transformation with no dynamic scheduling; hence, it does not create extra parallelism. In other words, we move work to threads that are already in flight, instead of spawning new threads. The main benefit of CREV, when compared to these previous work comes in terms of programmability and efficiency: by supporting composability of multi-thread and SIMD code, we give developers the chance to benefit from efficient warp-synchronous idioms without neither having to deal with primitives like shuffle, vote and population count, nor having to worry about saving the context of threads. Again, considering our example, the main goal of our approach is to provide an easy-to-write solution for mixing SIMD and SIMT, while also conveying good performance. Given our sample application of **crev**, depicted by Figure 2.2, we seem to achieve simplicity; in Section 5, we show we get not too far behind w.r.t. performance.

Thread-level divergence aware optimizations. Compilers may reorder computations across loop iterations within each thread to mitigate branch divergence [Coutinho et al., 2011, Han and Abdelrahman, 2013, Novak, 2015, Khorasani et al., 2015]. However, each thread performs the same set of tasks as in the original version, so divergences induced by load unbalance between threads of a warp remains an issue. CREV is a way to deal with irregular programs whose performance divergences hurt. However, CREV is not an optimization implemented by the compiler: programmers must adapt algorithms to use this construct. CREV deals well with divergences because it lets developers balance workload between threads in flight. In other words, it changes the loop structure by distributing iterations across different threads.

Conclusion. This chapter has introduced the basic notions around function re-vectorization via an example. This example gave us the opportunity to provide some insights on the syntax and the semantics of the new abstraction that we propose. In particular, it made clear the difficulty of combining programmability and efficiency in the SIMD world. In the next chapter we start to explain the details around the implementation of CREV. In particular, we define its semantics and syntax more formally.

Chapter 3

Semantics of CREV

This chapter presents the semantics of Call Re-Vectorization. First, in Section 3.1, we informally state key features of CREV. In Section 3.2, we introduce μ-SIMD, a low-level programming language with a set of primitives that lets us implement CREV. In Section 3.3, we show how to implement the **crev** high-level construct using the building blocks available in μ-SIMD. Finally, in Section 3.4, we use our semantics to state some properties of CREV. Before we dive into these details, we suggest the reader to revisit Example 2.3.1 and thus recapitulate the overall control flow behaviour of a CREV call.

3.1 The Cornerstones of CREV

CREV is defined as follows: for each active thread that reaches a call tagged with **crev**, we execute the target function once, forwarding global parameters (scalars) and extracting private ones per active thread (vectors). This principle of Call Re-Vectorization lays on three pillars: *thread re-activation*, *SIMD function call* and *data distribution*.

Thread Re-Activation. CREV does not lead to the creation of new threads. A function invoked via a CREV call is executed by every thread of the running warp, regardless of the state of such threads (active or not). As mentioned in Section 2, a thread might be inactive due to divergences. There is, however, a means of temporarily re-enabling threads, which is the functionaly conveyed by an `everywhere` block. We capitalize on this construct in order to implement our **crev** extension, thus re-activating dormant threads to perform work.

Given a warp, its former state, i.e., the former state of its threads, is saved into the context stack, used for divergence management. The context stack basically has

to store bit masks for keeping track of the state of each thread within its associated warp. On software-based context stack implementations, such as used on AMD GPUs and Intel AVX-512 platforms, this operation is performed entirely in software. For platforms with hardware-based context stack implementations, like NVIDIA GPUs, it requires a new machine instruction to allow software manipulation of the context stack.

SIMD Function Call. Multi-thread (MT) and SIMD languages have different definitions of function calls. In an MT setting, a function call is only performed by active threads, and upon processing instructions, only register lanes that correspond to active threads are saved. The other threads are guaranteed to stay inactive during execution of the function, thus requiring no context save. Although each thread conceptually has its own private call stack, the call stacks of a warp are typically synchronized, both for performance reasons and to allow the sharing of a single scalar stack pointer for a warp. Implementations of SIMD languages, on the other hand, save whole vector registers on function calls, keeping one stack pointer per warp.

Unlike regular MT functions, procedures invoked with the **crev** directive (r-functions) follow an SIMD application binary interface. This ensures that all registers in-use are saved before being overwritten inside the function, including lanes of threads that were inactive. Because **crev** does not create new threads, the cost associated to our construct is equivalent to that of a context switch resultant of invoking a new function.

Data Distribution. Upon entrance at **crev** call, the execution flow issues a series of light-weight data serializations, one per formerly active thread within the running warp. The serialization consists in loading data, once local to each thread, into global variables, accessible to all threads within the warp. The goal here is to allow the target r-function to receive its input arguments, and also ensure all threads to have access to such data.

Once each formerly active warp thread is serialized to have a full warp operate on its data, the r-function can be correctly invoked. Revisiting our warp-synchronous `memcpy` example (Algorithm 2), the local data serialization requires extracting and broadcasting each thread's register lane. More specifically, all variables that were local to each thread, but that are used as input argument to the r-function, must be serialized and broadcast. Data distribution will be later covered with more detail in Algorithm 5.

branch if zero	bz v, l
unconditional branch	jmp l
branch if thread previously active	jmp_mask t_{id}, l
write to shared memory	$\uparrow v_x = v$
read from shared memory	$v = \downarrow v_x$
binary operations	$v_1 = o_1 \oplus o_2$
copy	$v = o$
shuffle data between lanes	shfl(v, v_{lane})
synchronization barrier	sync
halt the machine	stop
begin everywhere block	everywhere
end everywhere block	end_everywhere

Figure 3.1. μ-SIMD instruction set. Operands (o) can be either variables or integer constants.

Labels (L)	$::= l \in \mathbb{N}$
Constants (C)	$::= c \in \mathbb{N}$
Variables (V)	$::= T_{id} \cup \{v_1, v_2, \dots\}$
Instructions (I)	::= Figure 3.1
Active Threads	$\Theta \subset \mathbb{N}$
Local Memory	$\sigma \subset V \mapsto \mathbb{Z}$
Local Memory Bank	$\beta \subset T_{id} \mapsto \sigma$
Shared Memory	$\Sigma \subset \mathbb{N} \mapsto \mathbb{Z}$
Synch Stack	$\Pi \subset (L \times \Theta \times L \times \Theta \times \Pi)$
Context Stack	$\Lambda \subset (\Theta \times \Pi \times \Lambda)$
Program	$P \subset L \mapsto I$
Program Counter	$pc \in \mathbb{N}$

Figure 3.2. The state of μ-SIMD machine is a septuple $M(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc)$. Θ is the set of active threads. A thread $t \in \Theta$ has a local memory σ, accessible through a memory bank β. Threads communicate through shared memory Σ. The stack Π tracks control flow divergences. A key component of Call Re-Vectorization is the thread stack Λ. The program counter, pc, keeps track of the next instruction $\iota \in P$ to be executed. The program P is a linear sequence of instructions. Although it never changes, we include it as state for convenience.

3.2 Low-Level Semantics

We formalize the notion of Call Re-Vectorization on top of a core language, μ-SIMD. This language provides the low-level constructs necessary to implement **crev** and thus invoke r-functions. Most of the syntax of μ-SIMD comes from Sampaio *et al.* [Sampaio et al., 2013], who, in turn, have reused ideas from Bougé *et al.* [Bougé and Levaire, 1992] and Farrell *et al.* [Farrell and Kieronska, 1996]. A μ-SIMD program is a sequence of instructions indexed by a pc. Figure 3.1 shows μ-

$\mathbf{split}(\Theta, \beta, v) = (\Theta_0, \Theta_n)$ **where**
$\Theta_0 = \{t \mid t \in \Theta \textbf{ and } \beta[t] = \sigma_t \textbf{ and } \sigma_t[v] = 0\}$
$\Theta_n = \{t \mid t \in \Theta \textbf{ and } \beta[t] = \sigma_t \textbf{ and } \sigma_t[v] \neq 0\}$

$\mathbf{push}([], \Theta_n, pc, l) = [(pc, [], l, \Theta_n)]$
$\mathbf{push}((pc', [], l', \Theta'_n) : \Pi, \Theta_n, pc, l) = \Pi'$ **if** $pc \neq pc'$
where $\Pi' = (pc, [], l, \Theta_n) : (pc', [], l', \Theta'_n) : \Pi$
$\mathbf{push}((pc, [], l, \Theta'_n) : \Pi, \Theta_n, pc, l) = (pc, [], l, \Theta_n \cup \Theta'_n) : \Pi$

Figure 3.3. Auxiliar functions used to define μ-SIMD. split is a filter, dividing *threads* into two divergent sets (Θ_0 and Θ_n). Auxiliary function push updates the synchronization stack Π due to control flow divergences.

SIMD's syntax.

Operational Semantics. The state M of a program is a tuple $(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc)$, as described in Figure 3.2. Threads are uniquely identified by a natural t_{id}, having a local memory $\beta[t_{id}]$, and sharing a global memory Σ. Memory is vectorized, thus, a local address v denotes a vector of variables $v \in \beta[t_{id}]$; hence, each thread sees its private version of v.

To formalize the semantics of μ-SIMD, we use the auxiliary functions shown in Figure 3.3. The semantics of μ-SIMD is given by Figures 3.4 and 3.5. The former shows the behavior of instructions that change the program's control flow; the latter shows the behavior of logic and arithmetic instructions. The result of executing a control flow instruction is a triple (Θ, β, Σ). The interface between Figure 3.4 and Figure 3.5 is performed by Rules IT and TL. The result of executing an arithmetic or logic instruction is a pair (β, Σ), i.e., they only update the program memory.

The semantics of control flow divergences. To simulate the effect of divergences, μ-SIMD has a stack Π. Each element in Π is a tuple $(l_{id}, \Theta_{done}, l_{next}, \Theta_{todo})$, which indicates the point where divergent threads must re-converge. A new tuple is pushed onto Π due to a conditional branch, located at l_{id}, that has caused a divergence, as described by Rules BT, BF and BD, in Figure 3.4. Θ_{done} is the set of threads that have reached the synchronization point. Θ_{todo} is the set of threads waiting to execute. These threads, once active, will resume execution at label l_{next}. The stack is popped by instructions sync, whose behavior is given by Rules SS and SP.

The Thread Stack. To implement CREV, we have added a *thread stack* Λ to μ-SIMD. This stack is fundamental to the implementation of **everywhere** blocks. Λ

$$\text{(SP)}\quad \frac{P[pc] = \mathit{stop}}{(\Theta, \beta, \Sigma, \emptyset, \Lambda, P, pc) \to (\Theta, \beta, \Sigma)}$$

$$\text{(JP)}\quad \frac{P[pc] = \mathtt{jmp}\ l \qquad (\Theta, \beta, \Sigma, \Pi, \Lambda, P, l) \to (\Theta', \beta', \Sigma')}{(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \to (\Theta', \beta', \Sigma')}$$

$$\text{(BT)}\quad \frac{P[pc] = \mathtt{bz}\ v, l \qquad \mathbf{split}(\Theta, \beta, v) = (\Theta, \emptyset) \qquad \mathbf{push}(\Pi, \emptyset, pc, l) = \Pi' \qquad (\Theta, \beta, \Sigma, \Pi', \Lambda, P, l) \to (\Theta', \beta', \Sigma')}{(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \to (\Theta', \beta', \Sigma')}$$

$$\text{(BF)}\quad \frac{P[pc] = \mathtt{bz}\ v, l \qquad \mathbf{split}(\Theta, \beta, v) = (\emptyset, \Theta) \qquad \mathbf{push}(\Pi, \emptyset, pc, l) = \Pi' \qquad (\Theta, \beta, \Sigma, \Pi', \Lambda, P, pc + 1) \to (\Theta', \beta', \Sigma')}{(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \to (\Theta', \beta', \Sigma')}$$

$$\text{(BD)}\quad \frac{P[pc] = \mathtt{bz}\ v, l \qquad \mathbf{split}(\Theta, \beta, v) = (\Theta_0, \Theta_n) \qquad pc' = pc + 1 \qquad \mathbf{push}(\Pi, \Theta_n, pc, l) = \Pi' \qquad (\Theta_0, \beta, \Sigma, \Pi', \Lambda, P, pc') \to (\Theta', \beta', \Sigma')}{(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \to (\Theta', \beta', \Sigma')}$$

$$\text{(BA)}\quad \frac{P[pc] = \mathtt{jmp\_mask}\ T_{id}, l \qquad T_{id} \in \Theta' \qquad (\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, l) \to (\Theta'', \beta', \Sigma')}{(\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, pc) \to (\Theta'', \beta', \Sigma')}$$

$$\text{(BI)}\quad \frac{P[pc] = \mathtt{jmp\_mask}\ T_{id}, l \qquad T_{id} \notin \Theta' \qquad (\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, pc + 1) \to (\Theta'', \beta', \Sigma')}{(\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, pc) \to (\Theta'', \beta', \Sigma')}$$

$$\text{(SS)}\quad \frac{P[pc] = \mathtt{sync} \qquad \Theta_n \neq \emptyset \qquad (\Theta_n, \beta, \Sigma, (pc', \Theta_0, l, \emptyset) : \Pi, \Lambda, P, l) \to (\Theta', \beta', \Sigma')}{(\Theta, \beta, \Sigma, (pc', \emptyset, l, \Theta_n) : \Pi, \Lambda, P, pc) \to (\Theta', \beta', \Sigma')}$$

$$\text{(SI)}\quad \frac{P[pc] = \mathtt{sync} \qquad pc' = pc + 1 \qquad (\Theta_n, \beta, \Sigma, (\_, \emptyset, \_, \Theta_0) : \Pi, \Lambda, P, pc') \to (\Theta', \beta', \Sigma')}{(\Theta_0 \cup \Theta_n, \beta, \Sigma, \Pi, \Lambda, P, pc) \to (\Theta', \beta', \Sigma')}$$

$$\text{(EB)}\quad \frac{P[pc] = \mathtt{everywhere} \qquad (\Theta_{all}, \beta, \Sigma, \emptyset, (\Theta, \Pi) : \Lambda, P, pc + 1) \to (\Theta', \beta', \Sigma')}{(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \to (\Theta', \beta', \Sigma')}$$

$$\text{(EE)}\quad \frac{P[pc] = \mathtt{end\_everywhere} \qquad (\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc + 1) \to (\Theta', \beta', \Sigma')}{(\_, \beta, \Sigma, \emptyset, (\Theta, \Pi) : \Lambda, P, pc) \to (\Theta', \beta', \Sigma')}$$

$$\text{(IT)}\quad \frac{P[pc] = \iota \qquad \iota \neq \text{Control Flow Instruction} \qquad (\Theta, \beta, \Sigma, \Theta_{mask}, \iota) \to (\beta', \Sigma') \qquad pc' = pc + 1 \qquad (\Theta, \beta', \Sigma', \Pi, (\Theta_{mask}, \Pi') : \Lambda, pc') \to (\Theta', \beta'', \Sigma'')}{(\Theta, \beta, \Sigma, \Pi, (\Theta_{mask}, \Pi') : \Lambda, P, pc) \to (\Theta', \beta'', \Sigma'')}$$

Figure 3.4. Semantics of μ-SIMD's control flow instructions.

holds pairs (Θ, Π). Figure 3.4 shows that instructions `everywhere` (Rule EB) push elements onto Λ, and instructions `end_everywhere` (Rule EE) pop it. The first element in this tuple is the set of threads active immediately before the execution of an

$$(\textsc{Mm})\quad \frac{\Sigma(v) = c}{\Sigma \vdash v = c} \qquad (\textsc{Mt})\quad t, \beta \vdash t_{id} = t \qquad (\textsc{Mv})\quad \frac{\beta[t] = \sigma_t \qquad \sigma_t(v) = c}{t, \beta \vdash v = c}$$

$$(\textsc{Tl})\quad \frac{(t, \beta, \Sigma, \Theta_{mask}, \iota) \to (\sigma_t, \Sigma') \qquad (\Theta, \beta \setminus [\beta[t] \mapsto \sigma_t], \Sigma', \Theta_{mask}, \iota) \to (\beta", \Sigma")}{(\{t\} \cup \Theta, \beta, \Sigma, \Theta_{mask}, \iota) \to (\beta", \Sigma")}$$

$$(\textsc{Bp})\quad \frac{t, \beta \vdash v_2 = c_2 \qquad t, \beta \vdash v_3 = c_3 \qquad \beta[t] = \sigma_t \qquad c_1 = c_2 \oplus c_3}{(t, \beta, \Sigma, \_, v_1 = v_2 \oplus v_3) \to (\sigma_t \setminus [v_1 \mapsto c_1], \Sigma)}$$

$$(\textsc{Si})\quad \frac{t, \beta \vdash v_1 = c_1 \qquad t, \beta \vdash v_{lane} = c_{lane} \qquad \beta[t] = \sigma_t \qquad c_{lane} \notin \Theta_{mask}}{(t, \beta, \Sigma, \Theta_{mask}, \mathtt{shfl}(v_1, v_{lane})) \to (\sigma_t \setminus [v_1 \mapsto \_], \Sigma)}$$

$$(\textsc{Sv})\quad \frac{\begin{array}{c} t, \beta \vdash v_1 = c_1 \\ t, \beta \vdash v_{lane} = c_{lane} \qquad \beta[t] = \sigma_t \qquad c_{lane} \in \Theta_{mask} \qquad \beta[c_{lane}] = \sigma_{lane} \qquad \sigma_{lane}(v_1) = c_2 \end{array}}{(t, \beta, \Sigma, \Theta_{mask}, \mathtt{shfl}(v_1, v_{lane})) \to (\sigma_t \setminus [v_1 \mapsto c_2], \Sigma)}$$

$$(\textsc{Ct})\quad \frac{\beta[t] = \sigma_t}{(t, \beta, \Sigma, \_, v = c) \to (\sigma_t \setminus [v \mapsto c], \Sigma)}$$

$$(\textsc{Ld})\quad \frac{t, \beta \vdash v_x = c_x \qquad \beta[t] = \sigma_t \qquad \Sigma \vdash c_x = c}{(t, \beta, \Sigma, \_, v =\downarrow v_x) \to (\sigma_t \setminus [v \mapsto c], \Sigma)}$$

$$(\textsc{As})\quad \frac{t, \beta \vdash v' = c \qquad \beta[t] = \sigma_t}{(t, \beta, \Sigma, \_, v = v') \to (\sigma_t \setminus [v \mapsto c], \Sigma)}$$

$$(\textsc{St})\quad \frac{t, \beta \vdash v_x = c_x \qquad t, \beta \vdash v = c \beta[t] = \sigma_t}{(t, \beta, \Sigma, \_, \uparrow v_x = v) \to (\sigma_t, \Sigma \setminus [c_x \mapsto c])}$$

Figure 3.5. Semantics of arithmetic, logic and data-related instructions. Rule TL loops over every thread $t \in \Theta$, and for each one of them, executes instruction ι. No assumption can be made on the order in which instructions run.

everywhere block. The second element is the divergence stack, also in the state before the execution of the last **everywhere** block traversed by the program flow. In Rule EB (Fig. 3.4), Θ_{all} represents all the threads available in a warp. The thread stack lets us represent an unbounded number of different thread contexts; hence, programs might contain an arbitrary number of nested **everywhere** blocks. After executing the instruction `end_everywhere`, threads previously inactive will go back into sleeping mode. In other words, after `end_everywhere`, the pair (Θ, Π) at the top of Λ is popped, and the diverging configuration Π becomes part of the current state of threads. If necessary to check if a thread is active due to an **everywhere** block, then μ-SIMD provides a conditinal `jmp_mask`. The statement $jmp\_mask(t_{id}, l)$ will divert execution to l if t_{id} is active in the mask at Λ's top. Example 3.2.1 illustrates the behavior of these instructions.

	Instructions
	v0 = ↓tid
	v1 = (v0 == 0)
	bz v1, Done
	v2 = 4 * (tid + 1)
	everywhere
	v8 = 0
Loop	jmp_mask v8, Call
	jmp Next
Call	v3 = shfl(v2, v8)
	v4 = v3 + tid
	v5 = ↓v4
	v6 = v5 + 1
	↑v4 = v6
Next	v8 = v8 + 1
	v7 = (v8 == 4)
	bz v7, Loop
	end_everywhere
	↑tid = 1
Done	sync

	Shared Memory			
Address	0	1	2	3
Contents	0	1	1	0
Address	4	5	6	7
Contents	2	1	3	4
Address	8	9	10	11
Contents	1	5	6	1
Address	12	13	14	15
Contents	2	3	1	7
Address	16	17	18	19
Contents	1	3	4	0

	Private Memory			
Address	v0	v1	v2	v3
Contents	*	*	*	*
Address	v4	v5	v6	v7
Contents	*	*	*	*
Address	v8			
Contents	*			

Figure 3.6. Program written in μ-SIMD, plus its initial state.

Example 3.2.1 *Figure 3.6 shows a program written in* μ-SIMD. *We assume* $|\Theta_{all}| = 4$. *This program increments a* 4×4 *matrix; however, line* i *is incremented only if* $\Sigma[t_{id}] = 0$. *The figure shows the initial state of the shared* (Σ) *and local memory of each thread* (σ). *The initial state of the variables in the local memory is immaterial for this example. Figure 3.7 shows a trace of the execution of the program, given its initial state. Only threads* $t_{id} = 0$ *and* $t_{id} = 3$ *will enter the* ***everywhere*** *section, because* $\Sigma[0] = \Sigma[3] = 0$. *Nevertheless, all the four threads will execute the commands within that block. Instruction* `v3 = shfl(v2, v8)` *lets each thread read into* `v3` *the value of* `v2` *seen by thread* `v8`.

3.3 High-Level Semantics

The μ-SIMD assembly gives us the primitive building blocks to implement CREV in higher-level languages. As a proof of concept, we have implemented CREV onto ISPC, using instructions of ISPC that are equivalent to those seen in μ-SIMD. By focusing on an abstract notation, μ-SIMD, instead of on a concrete language, such as ISPC, we claim generality: CREV can be implemented in any environment that supports our notions of **everywhere** and shuffle. In this section we show how to implement the **crev** modifier, which marks a function call as an r-function. For simplicity, our high-level language provides only syntax to declare and invoke functions. A function declaration consists of a *name* f, plus a list of *formal parameters*, e.g.: $f(T\ p_1, \ldots, T\ p_n)$. We

			Tid					Tid			
	Instructions	**Var**	**0**	**1**	**2**	**3**		**0**	**1**	**2**	**3**
	v0 = ↓tid	v0	0	1	1	0		✓	✓	✓	✓
	v1 = (v0 == 0)	v1	1	0	0	1		✓	✓	✓	✓
	bz v1, Done		F	T	T	F		✓	✓	✓	✓
	v2 = 4 * (tid + 1)	v2	4	*	*	16		✓	•	•	✓
	everywhere							✓	•	•	✓
	v8 = 0	v8	0	0	0	0		✓	✓	✓	✓
Loop	jmp_mask v8, Call		T	T	T	T		✓	✓	✓	✓
	jmp Next		F	F	F	F		✓	✓	✓	✓
Call	v3 = shfl(v2, v8)	v3	4	4	4	4		✓	✓	✓	✓
	v4 = v3 + tid	v4	4	5	6	7		✓	✓	✓	✓
	v5 = ↓v4	v5	2	1	3	4		✓	✓	✓	✓
	v6 = v5 + 1	v6	3	2	4	5		✓	✓	✓	✓
	↑v4 = v6							✓	✓	✓	✓
Next	v8 = v8 + 1	v8	1	1	1	1		✓	✓	✓	✓
	v7 = (v8 == 4)	v7	0	0	0	0		✓	✓	✓	✓
	bz v7, Loop		T	T	T	T		✓	✓	✓	✓
Loop	jmp_mask v8, Call		F	F	F	F		✓	✓	✓	✓
	jmp Next		T	T	T	T		✓	✓	✓	✓
Next	v8 = v8 + 1	v8	2	2	2	2		✓	✓	✓	✓
	v7 = (v8 == 4)	v7	0	0	0	0		✓	✓	✓	✓
	bz v7, Loop		T	T	T	T		✓	✓	✓	✓
Loop	jmp_mask v8, Call		F	F	F	F		✓	✓	✓	✓
	jmp Next		T	T	T	T		✓	✓	✓	✓
Next	v8 = v8 + 1	v8	3	3	3	3		✓	✓	✓	✓
	v7 = (v8 == 4)	v7	0	0	0	0		✓	✓	✓	✓
	bz v7, Loop		T	T	T	T		✓	✓	✓	✓
Loop	jmp_mask v8, Call		T	T	T	T		✓	✓	✓	✓
	jmp Next		F	F	F	F		✓	✓	✓	✓
Call	v3 = shfl(v2, v8)	v3	16	16	16	16		✓	✓	✓	✓
	v4 = v3 + tid	v4	16	17	18	19		✓	✓	✓	✓
	v5 = ↓v4	v5	1	3	4	0		✓	✓	✓	✓
	v6 = v5 + 1	v6	2	4	5	1		✓	✓	✓	✓
	↑v4 = v6							✓	✓	✓	✓
Next	v8 = v8 + 1	v8	4	4	4	4		✓	✓	✓	✓
	v7 = (v8 == 4)	v7	1	1	1	1		✓	✓	✓	✓
	bz v7, Loop		F	F	F	F		✓	✓	✓	✓
	end_everywhere							✓	•	•	✓
	↑tid = 1							✓	•	•	✓
Done	sync							✓	✓	✓	✓

Figure 3.7. Execution trace of the program in Figure 3.6. Column **Var** shows contents of last variable assigned. T indicates branch taken; F indicates otherwise. The symbol • marks inactive threads. For the syntax of instructions, we refer the reader to Fig. 3.1; for their semantics, Figs. 3.4 and 3.5.

let T denote a *type modifier*, which can be either *uniform* or *varying*. Recall from Chapter 2.4, that a *uniform* variable holds a single value, shared across all threads from the warp, whereas a *varying* variable holds a vector of values, one per thread. We have borrowed such notation from ISPC. Other programming languages have different ways to express these modifiers. For instance, in CUDA we have *shared* and *global* allocation filling the role of ISPC's uniform variables.

Figure 3.8 shows the code that we produce for a r-function call $f(a_1, \ldots, a_n)$,

```
1          everywhere                ;;begin CREV
2          i = 0                     ;;Loop counter
3  loop :  jmp_mask i, call
4          jmp next                  ;;Skip idle threads
5  call :  extract(t^n, p^n, a^n, i) ;;Algorithm 5
6          "call" f                  ;;function call
7  next :  i = i + 1
8          bnz(i ≠ W) loop
9          end_everywhere            ;;end CREV
```

Figure 3.8. Low-level code produced to call r-function f.

Algorithm 5: Data distribution

```
1 Function declaration: f(p_1, ..., p_n);
2 Function call: f(t_1 a_1, ..., t_n a_n);
3 Function extract(t^n, p^n, a^n, i)
4     for k ∈ 1...n do
5         if t_k == uniform then
6             p_k = a_k;
7         if t_k == varying then
8             shfl(a_k, i);
```

where each $a_i, 1 \leq i \leq n$ is an actual argument of f. Such an r-function call will trigger up to $|\Theta|$ executions of f, one for each active thread $t \in \Theta$. The test in lines 3 or 8 in Figure 3.8 are used to single out the function invocation performed by each thread. A different call will happen due to each $handle_t$ label. In another dimension of parallelism, each function call will be executed by Θ_{all} threads, due to the **everywhere** block at lines 10 and 13. Thus, we might have up to Θ^2_{all} computations.

Algorithm 5 generates code that implements data distribution. Data distribution determines how actual parameters are bound to formal parameters, given that actual parameters can have one of two types: *uniform* or *varying*. By construction, r-functions have only uniform parameters. The loop in line 4 will go over all the function arguments, comparing formal (p) and actual (a) parameters. We let the type of a_i be t_i. If an actual argument is uniform, then parameter passing is trivially implemented as a copy between variables. Line 5 of Algorithm 5 generates code under such circumstance. If an actual parameter has type varying, then we generate code to perform a broadcast, as seen in line 7 of Algorithm 5.

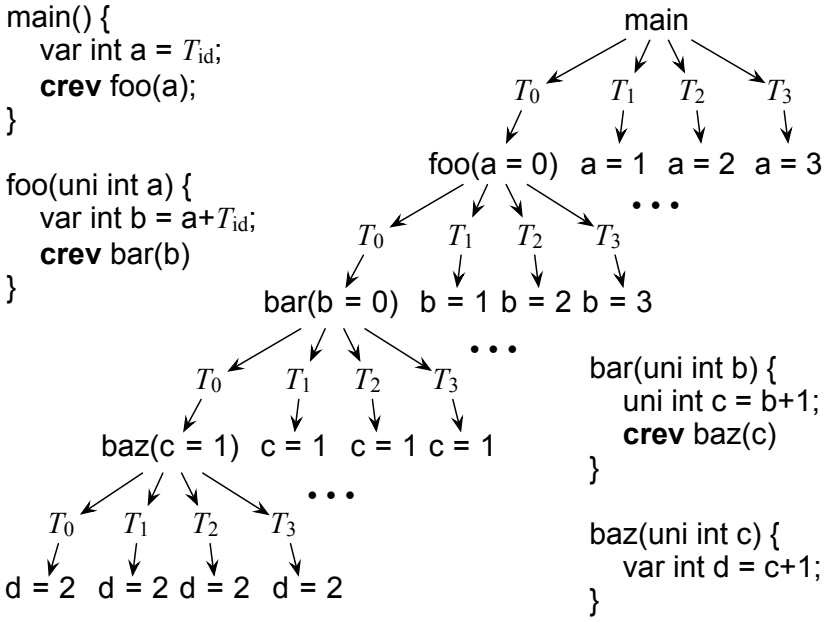

Figure 3.9. A program written in ISPC, and the tree showing function calls for T_0.

Example 3.3.1 *The program in Figure 3.9 shows three functions called via* ***crev****. We are assuming an architecture with four SIMD lanes, i.e.,* $\Theta_{all} = \{T_0, T_1, T_2, T_3\}$*. When* foo *is invoked, the value of* a*,* main*'s local variable, is broadcasted to* foo*'s formal parameter. Thus,* T_0 *sees* foo(0)*,* T_1 *sees* foo(1)*, etc. When* T_0 *calls* bar *from* foo*, the same behavior is observed. However, when* T_0 *calls* baz *from* bar*, all the four threads activated into this context see* baz(2)*, because* baz *receives a uniform argument. The fact that* baz*'s local variable* d *is marked as varying is immaterial in this example, as this variable is initialized with uniform values.*

3.4 Properties of CREV

The semantics of CREV, given by μ-SIMD's primitive building blocks, and the translator seen in Section 3.3, lets us establish a few properties that are true about this programming abstraction. In this section we go over a few of these properties. They are valid under the assumption that programs are *well-formed*. We define well-formed programs below:

Definition 3.4.1 (Well-Formed Program) *A* μ-SIMD *program is well-formed if any occurrence of an* everywhere *instruction at label* l_1 *is matched by an occurrence of an* end_ everywhere *instruction at label* l_2*, and these two labels are* control equivalent.

Definition 3.4.1 borrows the concept of control equivalence from Ferrante *et al.* [Ferrante et al., 1987]. Two points, l_1 and l_2, in a program's control flow graph are said to be control equivalent if l_1 dominates l_2, and l_2 post-dominates l_1. We say that l_1 dominates l_2 if, and only if, any path from the root of the CFG to l_2 must cross l_1. Dually, l_2 post-dominates l_1 if, and only if, any path from l_1 to the end of the CFG must cross l_2. Our translator produces well-formed programs, as long as the program flow cannot leave a function through points other than its return address.

Theorem 3.4.2 (Well-Formed Translation) *The translator of Figure 3.8 produces well-formed programs.*

> **Proof:** This result follows trivially from the fact that an **everywhere** block surrounds only Algorithm 5 and the r-function. Well-formedness holds as long as none of these routines let the program flow escape the enclosing end_everywhere instruction. This implies that the r-function cannot throw exceptions, for instance. □

Composability. CREV allows the nesting of **everywhere** blocks. Composition happens due to nested function calls. The thread stack Λ ensures that the last invoked r-function will be the first to remove pending computation. In what follows, we visit three consequences of this property.

Composition is multiplicative. An **crev** call will put all the warp threads in active mode. By coupling this observation with *composibility*, we have that, in the absence of divergences, a sequence of n nested **crev** calls will create $|\Theta_{all}|^N$ tasks. Notice that CREV produces new tasks, but not new threads: we still have only $|\Theta_{all}|$ threads to solve these tasks.

Commutativity. The translator of Figure 3.8 calls an r-function in a lexicographic order defined by thread identifiers. However, μ-SIMD's primitives do not impose any order on the threads pushed onto Λ. Therefore, the multiple SIMD calls of an r-function can be handled in any order.

Synchronization parity. There is no distinction between the top level of parallelism and the inner level of parallelism with regards to the synchronization primitive. In other words, divergences are handled transparently by the synchronization stack Π, and, from a synchronization standpoint, it is not possible to tell if execution exists within the context of an r-function or not. To ensure this property, μ-SIMD's everywhere instruction pushes onto Λ, together with the set of active threads, the divergent state Π.

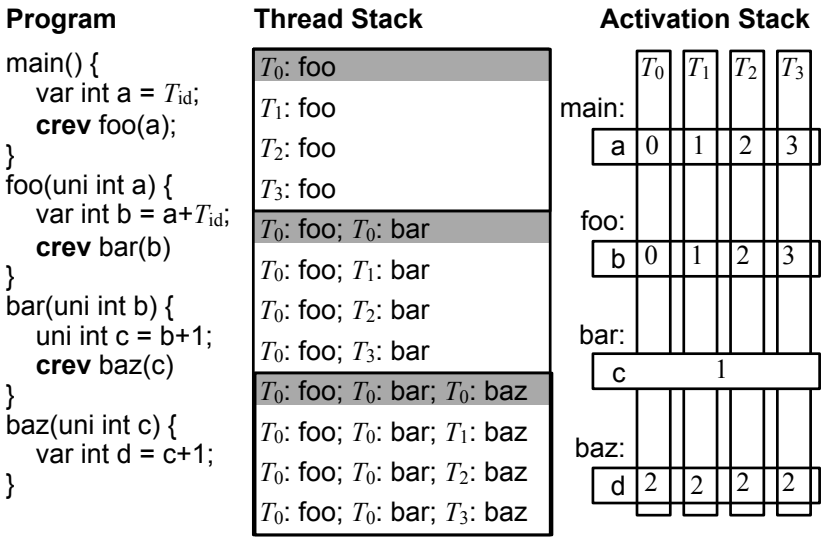

Figure 3.10. Example of three nested calls to r-functions. Calls currently in the activation stack are highlighted.

The interplay between CREV and nested function calls. The implementation of CREV does not interfere with the implementation of function calls. Programming languages that support recursion use a structure known as *activation stack* to manage function calls. Entries in the activation stack are called *activation records*, and they store functions' local variables, return address, arguments, etc. Upon invocation, the activation record of a function is pushed onto the activation stack. For each thread pushed onto the thread stack there will exist one activation record on the activation stack. The multiplicative nature of CREV also implies on a multiplication of activation records. Therefore, n nested r-calls will generate $|\Theta_{all}|^n$ activation records; however, the maximum depth of the activation stack is still $n+1$: activation records owned by different threads will not exist simultaneously.

Example 3.4.3 *Figure 3.10 reuse the program from Example 3.3.1 to illustrate these points. Again, we assume* $|\Theta_{all}| = 4$. *Thus, three non-divergent nested r-calls will create* $4 \times 4 \times 4 \times 4 = 256$ *tasks. At any time, the thread stack will contain at most* $4+4+4+4 = 16$ *tasks waiting for execution. The activation stack will contain, at any given point, at most 4 activation records, corresponding to the activation of functions* main, foo, bar *and* baz.

3.5 Discussion

This chapter has presented the semantics of CREV. The main benefit of having a formal semantics is the possibility to test different approaches when design the set of primitive instructions that constitute CREV. From these primitives, we can then design high-level constructs that give developers the opportunity to user our new abstraction. This observation is so true that we have implemented the semantics of CREV in Prolog. We have used this implementation extensively, before producing the actual x86 implementation of CREV. Thus, our semantics is executable, and extensible, as new constructs typical of the SIMD execution model can be added to it. In the next chapter we show how this semantics can be materialized into a concrete and robust implementation of function re-vectorization in an Intel architecture.

Chapter 4

Implementation

We now present our CREV implementation on top of Intel's SPMD compiler, ISPC. To truly grasp how the idiom works, and thus understand the expected results, it is desireable to not only be conversant with the ISPC compiler and language, but to have in mind the SIMD and SIMT processing models. Before diving into the technicality of our code, let us first get acquainted with ISPC in a higher level. Most of the information here exposed on the ISPC language and compiler we extracted from ISPC's documentation [Intel, 2016].

4.1 Making friends with ISPC

Intel's SPMD Program Compiler, ISPC for short, is an SPMD compiler for CPU applications. In SPMD programs, the abstraction is that of processing a single piece of data at a time, while the underlying hardware and runtime system hand over the same instruction to multiple processing nodes, each using as input its own data. The hardware handles such operations with as many parallel instances as there are resources available. The ISPC compiler is open source, with a BSD license. It uses the LLVM Compiler Infrastructure for back-end code generation and optimization. The compiler supports Windows, Mac, and Linux, with both x86 and x86-64 targets.

ISPC is a variant of the C programming languange and its companion compiler. The languange is meant to deliver good performance for programmers who want to run SPMD programs on CPUs. It provides a thin abstraction layer between the programmer and the hardware, in such a way that it is still possible to trace back, in the generated assembly code, the behaviour described in the source code of ISPC programs. ISPC harnesses the computational power of SIMD vector units without harming the

programmibilty, i.e., it spares programmers the burden of low-productive code writing, as it abstracts away low-level SIMD intrinsics.

4.1.1 Parallel Execution Model

In CUDA terminology, a warp is a group of threads set to execute a kernel. In ISPC, a warp is referred to as gang, and threads as program instances. For consistency only, we stick to CUDA's nomenclature. As ISPC is designed to support CPU SPMD programmers, a warp, or gang, is actually an SIMD vector unit, each lane being a thread, or program instance.

Code organization. An ISPC code is usually structured in two files: the C/C++ source accomodating the `main` function, and the ISPC source itself, containing the SPMD functions. As an abuse of terminology, let us refer to these functions as kernels, as they hold the core of ISPC parallelism and are executed on the target device – i.e., on the CPU vector units. To further clarify the composition of such kernels, bear in mind they are the one functions lowered to the vector-processing instruction set of the CPU.

The first source compiled is the ISPC one. The compiler generates a C header file, creating an interface between C and ISPC. Within the C source, programmers should include the header file generated, and make use of the kernels written on the ISPC source – notice this header also comprises types and structures created along with kernels. In addition to the header, the kernels are compiled to the target architectures selected on the command line. Then, the C source is compiled and linked with the programmer's brand-new SPMD library.

Code writing. ISPC programs are indeed very similar to C programs in terms of syntax. For example, the code from Figure 4.1.1 is valid both in C language and when compiled using the ISPC compiler. However, the semantics of the program changes considerably: unlike the result obtained in C – the floating number $(a + b/2)$ –, ISPC holds multiple values for a and b, one per each thread in a warp, thus generating multiple return values. From the parallel programming realm, ISPC ports the concepts of *global* and *private* values: variables in ISPC are either `uniform`, i.e., global/scalar values, uniform across all threads; or `varying`, i.e., each thread holds a private value, which may vary in comparison to other threads.

If we were to use a C compiler, the code generated for `func` would have a regular non-parallel semantic. Using the ISPC compiler, on the other hand, the assembly

```
float func(float a, float b) {
    return a + b / 2.;
}
float func_divergent(float a, float b) {
    if (programIndex % 2 == 0) return a + b / 2.;
    return 0;
}
```

Figure 4.1. `func` is a regular division function, as present in ISPC's documentation. The similarities between C and ISPC code are notable: this function has valid syntax in both language, but indees carries a different meaning. In C, such function is a regular division of float variables `a` and `b`, whereas such variables are actually vectors of values in ISPC, each value associated with a thread. In the latter, the result is a vector of floats – generally with unique values per thread. `func_divergent` wraps the main operation from `func` with a divergent branch.

produced comprises low-level instructions that exploits the vector processing capabilities of the target CPU. The semantics of the program changes considerably: instead of having two scalar values and a single control flow, we now encounter one unique control flow per thread, and each thread has its own version of variables `a` and `b`, i.e., such variables are now `varying` within the warp. Therefore, the ISPC kernel result in a vector of floats – generally with unique values per program instance.

The multiplicity of variables is statically defined by the length of the SIMD vector of processing units – a warp. The number of threads within a warp usually ranges from 2 to 8 threads, being thus quite small w.r.t. warps of often 32 threads, found in GPUs. Akin to SPMD GPU programs, ISPC's runtime system also grants each thread within a warp its own control flow: instructions are fetched and received in lock-step, and eventual divergence are handled by deactivating divergent threads. Extending our initial example with a conditional branch allows us to perceive this SPMD characteristic in ISPC. Still in Figure 4.1.1, we have function `func_divergent`, which wraps the division with a divergent branch. Whenever a thread of even index enter reach the branch, it will compute the division and return the result. Since the execution is in lock-step, threads with odd indices is put on hold at each instruction executed by active threads. Upon leaving the branch, the remaining threads return 0. Notice that, due to the keyword `programIndex`, function `func_divergent` may only be compiled by the ISPC compiler. `programIndex` is equivalent to CUDA's `threadIdx` in the sense it allows programmers to distinct threads by their indices.

Explicit parallelism. ISPC defines default qualifiers for variables: in plain C, function `func` (Figure 4.1.1) would have scalar variables `a` and `b`, and a single result would be

returned. In ISPC code, the declarations of `a` and `b` default to `varying float a` and `varying float b`, thus rendering such vars, by default, vectorized values. To explicitly inform the compiler the desired behavior, or even if to further document the code for future usage, programmers should add the qualifiers `uniform` or `varying`. It is upon having parallel data regular functions will issue their overloaded parallel versions and vice-versa.

An interesting point concerning uniform variables is the assignment policy. As stated in the quote below, extracted from the ISPC documentation, `uniform` variables cannot receive data from `varying` variables. For instance, if a warp holds an integer global acrosss all threads within the warp, a single thread is not able to assign this integer a varying value, private to this thread's context. It may seem counter-intuitive to attempt doing such an operation, but our implementation depends heavily on an equivalent syntax, as discussed ahead in Section 4.2.

> uniform variables can be modified as the program executes, but only in ways that preserve the property that they have a single value for the entire gang. Thus, it's legal to add two uniform variables together and assign the result to a uniform variable, but assigning a non-uniform (i.e., varying) value to a uniform variable is a compile-time error.
>
> – ISPC Documentation, "uniform" and "varying" Qualifiers.

ISPC also features the keywords `launch` and `sync`, for explicit asynchronous parallel processing. `launch` issues asynchronous parallel tasks and absracts away the many possible implementations of underlying engines for simulating asynchronous processing, which includes:

- Microsoft's Concurrency Runtime
- Apple's Grand Central Dispatch
- bare pthreads
- Cilk Plus
- TBB
- OpenMP

The task system implementation can be selected at compile time, by defining the appropriate preprocessor symbol on the command line. Not all combinations of platform and task system are meaningful. If no task system is requested, a reasonable default is selected for the host platform. The keyword `sync`, similar to a PTHREADS

`join`, puts the execution on hold until all threads have converged to its launch call site. As the task systems are available along with the ISPC open source code, the programmer may extend the set of runtime engines, developing her/his own task system.

Delineating the benefits from using ISPC's task system, we now show a wrapping-up example that uses both ISPC's SIMD capabilities and task system. The example consists of a generic matrix-based procedure: simply put, we create a vector of `varying` values, private to each thread; such values are zero-initialized, then, asynchronously, each entry of the vector (varying values in `data[i]`) receives its corresponding thread index multiplied by -1, being the last operation subject to whether the index is an even number.

Each line of the matrix is processed asynchronously, in parallel, by as many threads as the system supports; and then, each thread uses its associated SIMD resouces to operate, in lock-step, on the input line of `varying` values (function `f`). In this example, the matrix has a single parameterizable dimension: we can set the length of the vector within function `proc_matrix`, but the number of values each position `data[i]` holds is always defined by the length of the warp. In the next section we show, amongst other examples, how to exploit the SIMD capabilities without retaining oneself to a matrix structure.

4.1.2 ISPC Language

The ISPC language is an extended version of the C programming language, and provides a number of new features that make it easy to write high-performance SPMD programs for the CPU. Albeit there is but a handful of syntactic differences between ISPC and C code, the former conveys a fundamentally parallel execution model, and thereby C code cannot simply be compiled by the ISCP compiler to correctly run in parallel. However, starting with working C code and porting it to ISPC is an efficient way to quickly write ISPC programs [Intel, 2016].

We now introduce small examples of programs written in ISPC, as to further acquaint the reader with the language. Details on the syntax and semantics of the ISPC language can and should be looked up online, at ISPC's documentation webpage [Intel, 2016]. We here focus on elements we believe should grant the reader base knowledge for catching up with small ISPC applications and, indeed, appreciating the motivation for our CREV idiom in practice.

Hello World. We believe the examples from Figures 4.1.1 and 4.1.1 are very good starters, but to review the very basics of ISPC and make sure no reader may lag

```
// SIMD kernel: function executed in lock-step
task void f(varying int& data) {
    if (programIndex % 2 == 0) data = data + programIndex;
    else data = data - programIndex;
}
// SPMD function with invoking ISPC's task system
void f_matrix(varying int data[], uniform int length) {
    for (uniform int i = 0; i < length; ++i) launch f(data[i]);
    sync;
}
// Matrix procedure
export void proc_matrix() {

    varying int data[10];
    uniform int length = 10;

    for (uniform int i = 0; i < length; ++i) data[i] = 0;
    f_matrix(data, length);

    for (uniform int i = 0; i < length; ++i) {
        print("data[%]: %\n", i, data[i]);
    }

}
// Output
$ ./matrix-launch
data[0]: [0,-1,2,-3,4,-5,6,-7]
data[1]: [0,-1,2,-3,4,-5,6,-7]
...
data[9]: [0,-1,2,-3,4,-5,6,-7]
```

Figure 4.2. Sample matrix-based procedure in ISPC. `proc_matrix` creates a vector of `varying` values, which are zero-initialized and then, asynchronously, receive values depending on the thread it is subject to. In the upcoming section, we show how to process a matrix with two configurable dimensions – unlike this example, in which one dimension is parameterized and the other is given by the length of the processing warp (SIMD vector). The last lines show the output of running the program.

behind – which is important! –, we include this *hello world* example. Notice, though, the lessons here left are more technical, in the sense they broaden the familiarity with the ISPC syntax – whereas forecoming sample applications focus on exercising parallel concepts of the language.

As discussed before, ISPC code is organized in two main files: the main C/C++ file, which includes the header file with ISPC function declarations; and the ISPC implementation file, in which the developer shall define and export whatever functions should compose the ISPC interface with the C/C++ main. To access exported functions in the host language, use either C++ namespace `iscp::*`, or a separating

```
// C++ file: main.cpp
# include "helloworld_ispc.h"
int main(void) {
    ispc::hello_world();
    return 0;
}
// ISPC file: helloworld.ispc
export void hello_world() {
    print("Warp length: %\n", programCount);
    print("Active threads (all): %\n", programIndex);
    if (programIndex % 2 == 0) {
        print("Active threads (even ids): %\n", programIndex);
    } else {
        print("Active threads (odd ids): %\n", programIndex);
    }
}
// Output
$ ./hello_world
Warp length: 8
Active threads(all): [0,1,2,3,4,5,6,7]
Active threads(even ids): [0,((1)),2,((3)),4,((5)),6,((7))]
Active threads(odd ids): [((0)),1,((2)),3,((4)),5,((6)),7]
```

Figure 4.3. This example shows a very simple ISPC *hello world* program. We try to cover the notion of a running warp, possible divergences, and some of the basic keywords from ISPC.

underscore, as in `ispc_*`.

The ISPC compiler generates both an assembly and a header file. The header consists of composite types created in the ISPC source (special `structs` and `typedefs`), along with methods tagged with the `export` keyword. Therefore, apart from type declarations, any method expected to be available at the C/C++ source should be marked as `export`. The language also provides a `print` function with special implementation for dumping `varying` values. The expression string has a formatting similar to that from C's `printf`, except one needs only to add `%` as placeholders for variables – no type distinction is required.

ISPC has two reserved keywords that gives the programmer information about the warp size (`programCount`) and the index of each thread (`programIndex`). The first variable is a regular integer, and is output as such; whereas the second is a `varying` integer, with one value per thread in the warp (the index of each thread). When within a divergent region, inactive threads are marked by its index surrounded by a double pair of parantheses, like the output presented in our *HelloWorld* example.

Varying-Uniform data access. We also take the time to show, in the example from Figure 4.4, a syntatically invalid line, implementing the impossible assignment of a `varying` value to a `uniform` variable. This dummy procedure shows a very simple ISPC series of assignments that depend on whether the variable belong to the global address space or is local/private to each thread. The single invalid combination of variable attribution is that of assigning a `varying` value to a `uniform` var: the compiler may not know from which thread to extract the value and therefore cannot validate the syntax. This notion is revised and receives a new meaning when dealing with our CREV idiom extension, as will be discussed in Section 4.2.

```
export void var_uni() {
    uniform int a = programCount; // regular global-to-global assignment
    uniform int b = programIndex; // which private value should we keep?
    varying int c = programCount; // broadcast programCount
    varying int d = programIndex; // regular private-to-private assignment
}
```

Figure 4.4. This dummy procedure shows a very simple ISPC series of assignments that depend on whether the variable belong to the global address space or is local/private to each thread. The single invalid combination of variable attribution is that of assigning a `varying` value to a `uniform` var: the compiler may not know from which thread to extract the value and therefore cannot validate the syntax.

Renabling threads and Task parallelism. We know from our discussion in Section 6 that meeting up with the conditions that produce a situation that demands re-enabling threads within a divergent SIMT context is not a mere programming flourish. Re-enabling threads may become confusing depending on the organization of the code and on the level of nesting at which the divergence may be located. Such difficulty, along with the complexity of keeping track of the control flow still at a code level, characterizes warp-synchronous programming. Along with the source for our CREV-extended version of ISPC, the user shall find quite a few examples of warp-synchronous code. As hint of the level of detail required by such programming paradigm, we add to the appendix of this document the implementation of the intrinsically warp-synchronous algorithm BitonicSort A.1.

In Section 5 we present some examples of task parallelism, which is ISPC's take at dynamic parallelism. With special attention to the Mergesort and Quicksort algorithms (Section 5.5) we show how CREV may be an efficient surrogate to dynamic parallelism. In a nutshell, task parallelism in ISPC is materialized via the keyword `launch`, and its

usage – unexpected to be otherwise – is akin in syntax to that of **crev** and CUDA's dynamic parallelism. Overall, we hope the reader was able to follow the examples presented in this section. Any questions on specific syntatic constructs should be clarified by ISPC's documentation [Intel, 2016] available online.

4.1.3 ISPC Compiler Architecture

The ISPC compiler also falls into the common compiler organization, presenting a front-end to deal with its high-level C-based ISPC language; a middle-end for applying code optimizations, which uses mostly LLVM's intermediate representation; and the back-end, also based on the LLVM infrastructure, that further optimizes and lowers intermidiate representation to the selected target architecture. For the initial phase, ISPC uses Bison, a YACC-compatible parser generator; and Flex, a fast lexical analyzer generator. We believe that fine-grain info on these tools, including on the LLVM infrastructure, are out of the scope of this work, and thus will not be covered. There is, though, plenty of reference on these tools online, which may be found at:

- Flex, fast lexical analyzer generator (>= 2.5)
- Bison, YACC-compatible parser generator (>= 2.4)
- LLVM Compiler Infrastructure (3.7.1)

Although we do not get into the minute details of the tools that compose ISPC, we find it relevant to highlight where in lies each of the sections of the code we used to implement our idiom. This consists mostly of the organization of the compiler itself: we have added an extension to the compiler, which is accessible, implementation-wise, via a keyword, **crev**. To implement such idiom on top of ISPC, we had to modify: the lexer, as to identify our keyword **crev** within the code; the parser, in order to validate the keyword and verify the syntax of a composite function call (**crev** <function-call>); ISPC's type system, as to overlook formerly-invalid `varying` to `uniform` parameter passing, which become valid in the context of a **crev** function invocation; and, of course, the execution flow of **crev** itself, implemented thoroughly on top of LLVM – ISPC's intermediate representation.

The files we have imbued with **crev** interpreting and processing lines are listed below. In the next chapter, we show how we have implemented CREV in a lower level, closer to the actual source code. As a gentle reminder, our code is available online[1] for whoever may be interested in – and can afford – spending some time to check out what

[1] ISPC-CREV source code `http://cuda.dcc.ufmg.br/swan/src/crev-ispc.tar.gz`.

- `lex.ll`: defines the lexemes and lexical grammar for the ISPC language. We have included our **crev** token into ISPC accordingly.
- `parse.yy`: defines the syntatic grammar for the ISPC language. We have also incorporated the necessary rules to ISPC grammar in order to validate **crev** function calls.
- `type.cpp`: defines ISPC's type system, i.e., the set of types the language supports, as well as the possible coercions in between those types. We have gone through the entire type system, analyzing the cases in which **crev** would validate coercions from `varying` to `uniform` variable – utterly prohibited in plain ISPC.
- `ctx.h`: declares structures for function context emission. We have had but to add a flag to noitfy other structures in the code they were dealing with a **crev** function call.
- `expr.h`: declares types and structures required for processing ISPC expressions. We have had to update a few function headers, just enough to pass around a flag stating whether the expression under analysis was a **crev** call.
- `expr.cpp`: defines functions to process the varios expressions allowed in ISPC. We have extended the set of expressions to bestow upon ISPC a fully-working **crev** implementation.

Figure 4.5. List of files modified in ISPC to implement our CREV idiom.

exactly has been done. Still, we hope the description is sufficiently self-contained, so anyone is able to understand our approach.

4.2 Implementation of CREV on ISPC

We now give a brief, yet lower-level view of our CREV implementation on Intel's SPMD Program Compiler. Recall from the list of files present in Figure 4.1.3 file `expr.cpp`, which we have extended as to define the behavior of a **crev** expression. We shall focus our discussion on this particular file. Afterwards, we present a few constructive examples of our idiom in practice.

4.2.1 IPSC-CREV

We chose Intel's SPMD Program Compiler for it is open source, but were quite lucky to find it makes extensive use of the LLVM infrastructure, a compilation framework with a large active community and well documented code. Although we have applied

Algorithm 6: CREV(Function F, Function target, VarList args)

```
1  currentMask ← stores state of threads before CREV;
2  for each index i := 0 to warpSize do
3      insert test in F to check whether thread i is active;
       /* Re-building arguments for thread i                    */
4      args_i ← [];
5      for each arg in args do
6          if arg is varying but parameter is uniform then
7              append value from extract(arg, i) to args_i;
8          else
9              append arg to args_i;
       /* Enabling all threads and executing target function   */
10     everywhere;
11     runs target function with arg_i arguments;
       /* Resetting original thread activity                   */
12     set warp state back to currentMask;
13     insert branch in F to jump to next thread or to final basic block;
```

a handful of changes using domain specific languages (e.g.: Bison and Flex), most of the code for implementing **crev** has been written in plain C++, using library functions from LLVM. We now take a closer look at the main procedure in our solution, which defines the behavior of a **crev** expression.

As a side-note, during the implementation of our solution on top of ISPC we found Function Call Re-Vectorization to be more suitable a name to what we were trying to achieve, in detriment to its predecessor, Lightweight Dynamic Parallelism. As the code was mostly implemented, we kept the old naming in the actual ISPC-CREV code. Here, for consistency with the remaining of the work, we update whatever occurrence of the old acronym **ldp** to **crev**.

Algorithm 6 is a code generation procedure. That means there is no actual execution of the program going on, the algorithm simply describes how the code is generated. At the moment a **crev** expression is encountered by ISPC/LLVM code generation, it has been asserted the presence of a syntatically correct and type-sound expression: the expression and associated operands are checked for syntax (**crev** <function-call>) and for type equivalence. Type equivalence here means that each operand must be either the same as, or coercible to their associated formal parameters.

We begin by assigning to variable `currentMask` the state the running warp (which

threads are active and inactive). Right after, for every thread of the warp, we create a basic block that holds a test: if the corresponding thread i is active, the target function will be invoked using arguments `args`$_i$ specific for that thread; otherwise, the execution shall flow right into the subsequent basic block, which holds the test for thread $i + 1$. For simplicity, we do not dwell into the basic block chaining process.

In case the target function has to be invoked for thread i, we first extract the values from `varying` variables whose formal parameters are `uniform`. Recall from Example 4.1.2 (Varying-Uniform data access) that assginment from `varying` to `uniform` is strictly prohibited in plain ISPC. Our idiom allows invoking, from within divergent regions, SIMD functions that require all threads to be active. If the programmer wants to exploit the vector processing capabilities of their CPU, intuitively, there must be plenty of data associated to the threads that are active within the region of divergence. Therefore, it is imperative to provide developers with such coercion.

The extraction is quite straightforward: whenever the coercion must take place, we issue an extra instruction, namely `extract`, responsible for loading the i-th value from a vectorized variable into a `uniform` one. Following the extraction and re-building of the original list of arguments, we finally invoke the target function using `args`$_i$ as argument list. Before invoking the function, though, we must make sure all threads will be enabled. To this end, we also emit the instruction `everywhere`, that temporarily enables all threads within the warp. The remaining of our code emission procedure consists in resetting the warp to its original state and chaining the current basic block with subsequent ones (last two lines from Algorithm 6).

4.2.2 Active Load Balancing

Another benefit from our construct is its clear-cut syntax-friendly load balancing functionality. Suppose you are about to implement a depth-first search (DFS) in an SIMD language. We know each thread to have it own data at the beginning of the traversal, and we also know the execution to be in a lock-step fashion: whenever a thread runs out of data to process, it will be held inactive, anxiously lingering for the remaining threads to finish their workload. Now imagine the outcome of a heavily unbalanced initial data distribution – say a straight-line graph, rooted at a single thread.

If the graph was not that unbalanced, the developer should be able to always re-distribute the workload, in a way to actively maintain threads busy. This is exactly the approach depicted in Figure 4.2.2: the last conditional of function `dfs` creates a divergent region, which does not interfere in having all threads active under recursive calls to `dfs`, due to our idiom **crev**. In the case of a straight-line graph, not even our

```
// Traverses the matrix in a depth-first fashion
void dfs(uniform struct Graph& graph, uniform int root, float * uniform f) {

    // Setting current node as visited
    if (graph.node[root].visited) return;
    graph.node[root].visited = true;

    // Performing some computation (just a division, as example)
    f[root] = graph.node[root].length / (float) graph.num_nodes;

    // Parallel traversing
    foreach (i = 0 ... graph.node[root].length) {
        int child = graph.node[root].edge[i].node;
        if (!graph.node[child].visited) crev dfs(graph, child, f);
    }

}
// Interface function with C/C++ main function
export void graph_dfs(uniform struct Graph& graph, uniform int root,
    float * uniform f) {
    foreach (i = 0 ... graph.num_nodes) graph.node[i].visited = false;
    dfs(graph, root, f);
}
```

Figure 4.6. ISPC-CREV implementation of a Depth-First Search. We highlight the contribution of **crev** to achieving an active load-balancing policy during the traversal: whenever function `dfs` is called, the data within the `varying` variable `child` is distributed in independent calls to **crev**'s target function `dfs`. This allows having all threads active within inner calls of `dfs`, even within the divergent region created by the last conditional of that function.

approach would be able to make amends but, intuitively, we should be able to tackle quite a few cases of moderately unbalanced graphs.

4.3 Discussion

This chapter has presented the implementation of CREV in ISPC, the Intel SPMD Program Compiler. ISPC is an industrial-strength product, that gave us all the infra-structure necessary to concretize the abstract semantics seen in the last chapter into an actual implementation. One important question that arises, at this point, is: "Why not to implement CREV into an actual GPU?" As we have seen in the last chapter, the implementation of CREV requires the ability to wake up threads that, due to divergences, are in a dormant state. ISPC gives us this ability, inasmuch as the mask of active threads are visible to the developers of this compiler. Unfortunately, this is not the case in current Nvidia machines. Thus, we opted to demonstrate the

effectiveness of our ideas in vector instructions. Nevertheless, as we shall see in the next chapter, this implementation is solid enough to gives us the opportunity to design and test high-performant code. This is a story that we shall tell in the next chapter.

Chapter 5

Experimental Evaluation

In order to evaluate the ideas presented in this work, we have implemented crev onto ISPC. Because crev is a novel concept within ISPC, this compilation framework does not provide benchmarks to evaluate our contribution. Thus, we have re-implemented seven classic algorithms using the new keyword. We compare these algorithms against parallel versions written in ISPC. Our seven benchmarks are: (i) String Matching; (ii) Depth-First Search; (iii) Leader Election; (iv) Book Filter; (v) Bellman-Ford; (vi) Merge-Sort; and (vii) Quick-Sort.

How to read our results. Results are measured in millions of execution cycles, as reported by ISPC testing environment. All the numbers reported are the average of five, out of six samples. We have removed the first to avoid cold-start discrepancies. The reader must bear three observations in mind when analyzing our results: (i) speedups are due to the better load distribution that CREV accomplishes by transporting work to inactive threads; (ii) slowdowns are due to the boilerplate code necessary to serialize threads, before invoking r-functions; (iii) we are comparing against an industrial-strength compiler; hence, speedups tend to be modest.

Experimental Setup. We have implemented CREV onto ISPC v 1.9.1, and have used it to target a 6-core 2.00 GHz Intel Xeon E5-2620 CPU with 8-wide AVX vector units. Henceforth, we shall be using warps that contain eight threads, e.g., $|\Theta_{all}| = 8$.

5.1 String Matching

String matching is the problem of finding a pattern P within a text T. Algorithm 7 shows the CREV-based implementation of string matching. This is a warp-synchronous implementation of parallel matching: each thread t_{id} tries to match P at positions $T[t_{id} + n \times W]$, where $n \leq |T|$, and W is the warp size. Thus, in the best scenario,

runtime is divided by W. This implementation is irregular: divergences might happen at lines 6 and 10. Each call to memcmp [1] will commence a CREV sequence of computations.

Algorithm 7: Pattern matching: CREV vs. Nave

```
1  P ← pattern; T ← target text;
2  W ← warp size; t_id ← thread index;
3  Function memcmp(Offset k)
4      m ← True;
5      for i ← t_id to |P| do
6          if P[i] ≠ T[i + k] then m ← False ;
7      if all (m = True) then Found(k) ;
8  Function StringMatch
9      for i ← t_id to (|T| − |P|) step W do
10         if P[0] = T[i] then crev memcmp(i) ;
11 Function NaiveStringMatch
12     for i ← t_id to (|T| − |P|) step W do
13         j ← 0; k ← i;
14         while j < |P| and P[j] = T[k] do
15             j ← j + 1; k ← k + 1;
16         if j = |P| then Found(k) ;
```

Figure 5.1 compares our implementation, seen in Algorithm 7 (StringMatch) against the equivalent parallel version that uses ISPC primitives (PAR). This competitor is function NaiveStringMatch in Algorithm 7. To give the reader a better perspective of the results, we also compare against the Knuth£Morris£Pratt (KMP) [Knuth et al., 1977] algorithm. KMP is sequential, but has lower complexity than Algorithm 7. It runs in $O(|T|+|P|)$, whereas Algorithm 7 runs in $O(|T|\times|P|/|W|)$. For this experiment we searched for prefixes of the pattern "*She had been watching him the la*", of sizes $4, 8, \ldots, 28, 32$ in Jane Austen's book *Pride and Prejudice*, taken from Project Gutenberg[2]. CREV is always faster than NaiveStringMatch, and runs faster than KMP in more than half the cases. CREV beats plain parallelism because it distributes function *memcmp* among the eight available threads. On the other hand, NaiveStringMatch has a potentially long divergent block in line 14. In our best result, observed for patterns of size eight, CREV runs in 44% of the time taken by NaiveStringMatch, and in 40% of the time taken by KMP.

[1] Function memcmp is also used at line 5 of Algorithm 1

[2] https://www.gutenberg.org/ebooks/42671

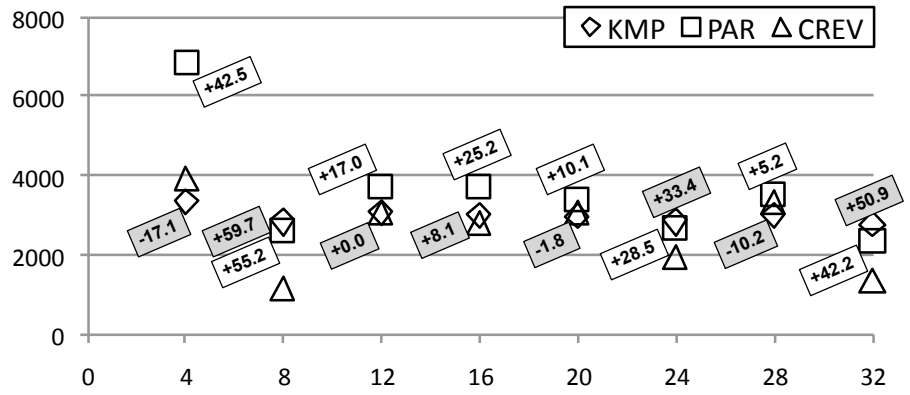

Figure 5.1. Comparison between CREV-based string matching (Algorithm 7), ISPC's parallel implementation, and the Knuth-Morris-Pratt version of pattern matching. The Y-axis shows runtime, in millions of cycles. The X-axis shows pattern sizes, in number of characters. The target text contains 256MB divided among 5,058,121 lines. White boxes show percentage of speedup (CREV over PAR); grey boxes show percentage of speedup (CREV over KMP).

5.2 Book Filter

We have compared Algorithm 1 against a parallel version, implemented in the original ISPC language. To avoid borrowing from the gains already observed in Section 5.1, we have only invoked function `memcpy` (Algorithm 1, line 6) using **crev**. We have used as input a binary assembly file, and have tried to copy chunks of 80 bits that contain the pattern of a register-register move instruction. To vary the input file, we have cropped the file at prefixes having 1K, 5K, 10K, 15K and 20K bits. The number of occurrences of the target pattern, which marks lines that must be copied, is similar in all the files: 4,097, 4,485, 4,795, 5,144 and 5,604 times. Figure 5.2 shows the result of this comparison. The CREV-based version runs faster in every sample, and the gap increases as the input increases. CREV gives a speedup of 11.2% in the 1K file, and of 13.2% in the 20K file. Gains in speed, in this experiment, are due to the r-function `memcpy` only, as `memcmp` was not invoked using **crev**.

5.3 Bellman-Ford

Our third experiment is an implementation of the classic Bellman-Ford algorithm [Bellman, 1958]. This algorithm computes shortest paths from a single source to all the other nodes in the graph. As input we use Erdös-Rényi [Erdos and Renyi, 1959] graphs with 216, 512, 1024 and 2048 vertices, and probability of 80% of existing an

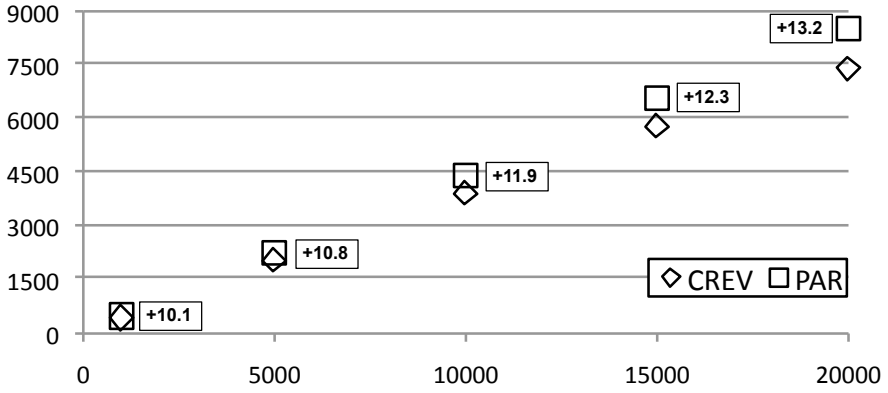

Figure 5.2. Comparison between CREV's and ISPC's book filter (Algorithm 1). Y-axis gives runtime, and X-axis input size, in bits. White boxes show speedup (%) over PAR.

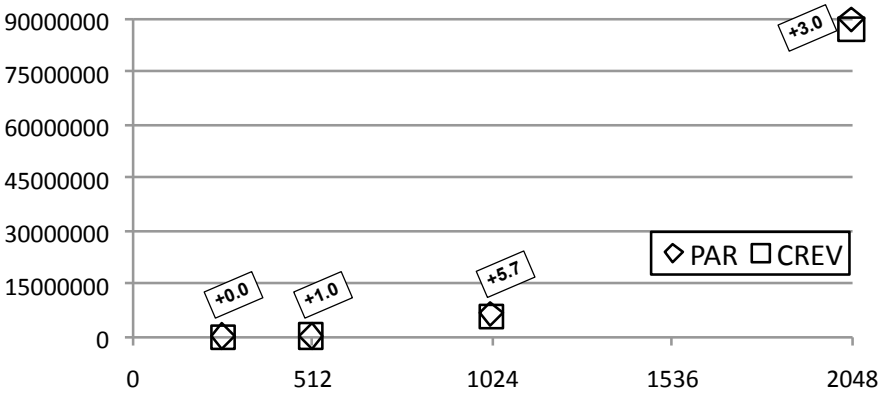

Figure 5.3. Comparison between CREV's and ISPC's version of Bellman-Ford. Y-axis gives execution time, in millions of cycles, and X-axis gives graph size, in number of nodes. White boxes show percentage of speedup over PAR.

edge between two nodes. Weights are randomly set from 1 to 100. Figure 5.3 shows the results that we have observed. CREV has yielded faster runtimes for all the input sizes, but the difference is small, and within error margin for graphs with 256 and 512 nodes. In the other cases, CREV is faster by 5.8% (1024 nodes) and by 3.1% (2048 nodes).

5.4 Depth-First Search and Leader Election

Our last two experiments use the same algorithm: Depth-First Search (DFS). We have compared CREV and ISPC on a straightforward implementation of DFS, seen in Algorithm 8. Algorithm 8 illustrates the composability of CREV, as it recursively calls the traversal routine for each vertex in the graph. To compare the CREV and plain parallel versions of DFS, we use the same graph model seen in Section 5.3. Figure 5.4 shows the results of this comparison. We have not observed any substantial difference between both implementations. In its best performance, the CREV version of DFS is 4.4% faster than ISPC's implementation (2,048 nodes). In the worst case (4,096) nodes, CREV got a slowdown of 1.1%, within the error margin of this experiment. We speculate that this slowdown is due to the overhead of serializing function calls and saving thread masks. Figure 3.8 shows this serialization. In all the other samples CREV has been slightly faster: 3.9% for 256 nodes and 2.7% for 1,204. Results were the same for graphs with 512 vertices. These numbers are highly input dependent. For instance, repeating the same experiment for full n-ary trees with 4K nodes reveals that CREV outperforms PAR by a greater margin. Full n-ary trees of height 3, 4 and 5, give us the following results: 0.036×0.043, 0.680×0.827 and 208.479×217.597. The first number is CREV's runtime; the second is PAR's, in millions of cycles.

Algorithm 8: SIMD Depth first traversal

```
1 W ← warp size; t_id ← thread index;
2 Function DFS(Node root, Function f)
3     f(root);
4     C ← child list of node root;
5     for i ← t_id to |C| step W do
6         c ← C[i];
7         if c is not null then
8             crev DFS(c, f);
```

We have used Algorithm 8 to build a leader election routine (LE). Leader Election uses a DFS, starting at each graph node, to propagate the identifier (ID) of that node to all the other vertices. Once a vertex receives an ID, it compares it against its current ID, keeping the greatest value. In the end, every vertex will have the largest ID among all the IDs within its connected component. Therefore, this algorithm finds connected components in graphs, by naming every vertex in the same component with a common identifier. Figure 5.5 shows the results of this experiment. CREV surpasses its original ISPC competitor in every sample. Its best result is a speedup of 15.5% for 2,048 nodes.

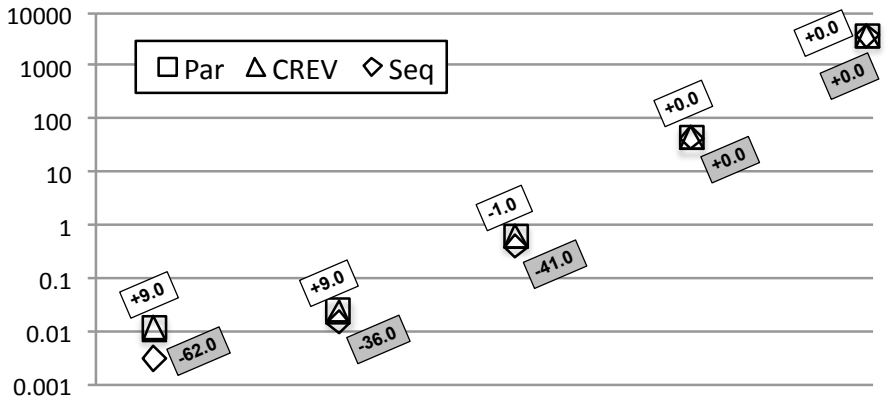

Figure 5.4. Comparison between CREV-based DFS and ISPC's parallel version. Y-axis gives execution time, in millions of cycles, and X-axis gives graph size, in number of nodes. White boxes show percentage of speedup over PAR.

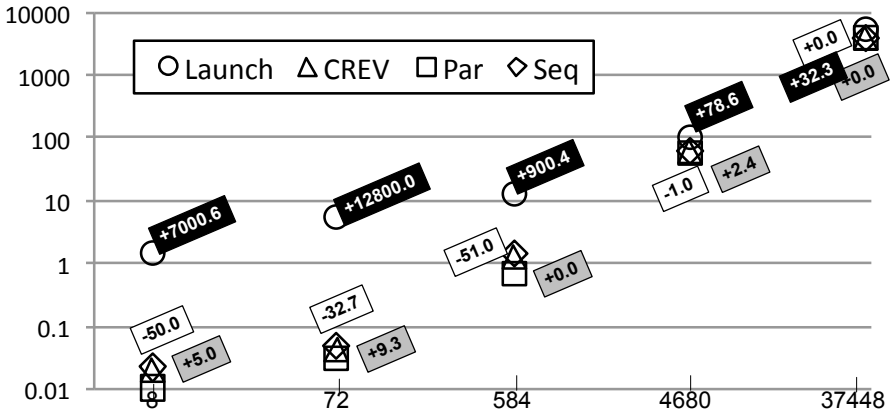

Figure 5.5. Comparison between CREV-based Leader Election and ISPC's parallel version. White boxes show percentage of speedup over PAR.

The worst result is a speedup of 3.0% for graphs with 4,096 vertices.

5.5 Merge-Sort and Quick-Sort

Sorting is part of the realm of classical algorithms studied in pretty much any computer science course as it serves as basis for many other solutions to be developed. Of course, we could not compose a benchmark test suite without evaluating our performance in this area as well. Towards this end, we have implemented both the Quick-Sort and the Merge-Sort algorithm using our **crev** extension to ISPC. First, let us present the Merge-Sort algorithm and how we have applied our solution to it. Afterwards, we show

Algorithm 9: Merge sort implementation

```
1  W ← warp size; t_id ← thread index;
2  Function mergesort-seq(Array A)
3      if |A| ≤ 2 then simple sorting of array A;
4      else
5          q ← |A| ÷ 2; A' ← {array, array + q};
6          mergesort-seq(A'[0]); mergesort-seq(A'[1]);
7          i ← 0; j ← q;
8          S ← empty vector with |A| positions;
9          for k ← 0 to |A| do
10             if i < q and j < len then
11                 if A[i] < A[j] then S[k] ← A[i], ++i;
12                 else S[k] ← A[j], ++j;
13             else if i < q then S[k] ← A[i], ++i;
14             else S[k] ← A[j], ++j;
15 Function mergesort-crev(Array A)
16     Equivalent to function mergesort-seq, except line 6 is replaced by:
17     if t_id < 2 then crev mergesort-crev(A'[t_id]);
```

our approach in face of the Quick-Sort algorithm, which is a similar solution, given the divide-and-conquer nature of both algorithms.

Algorithm 9 shows an implementation of Merge-Sort using `crev`. We aim at maximizing the parallelism at small workloads, i.e., we want all threads to be active during the execution of SIMD instructions. Our implementation relies on the fact that, upon reaching the branch at line 3, some threads may diverge, implying loss of control flow uniformity, and thus invalidating eventual SIMD kernel calls. Before we conclude on our solution, there are two important things to understand.

First, although the program is SIMT, and this is to say we have a group (warp) of threads executing the program, the parallelism is not perceived at times, as we may be dealing with purely global values. In other words, operations such as the ones described in lines 5 and the entire loop beginning at line 9 are sequential. The second important consideration, which may be regarded as even more peculiar, is the fact that we call the `mergesort-seq` procedure twice in line 6. In ISPC, the parallelism is structured in a bit more subtle way, as it is designed to exploit different capabilities of the CPU – those being the SIMD vector-processing, and the multi-tasking. Whenever a kernel is called within an ISPC kernel, its invocation is interpreted as: start the kernel with the threads that were active, as if the kernel were issued from a fresh C/C++ call, and upon its start, threads previously inactive were put back into sleep mode. This is important

Algorithm 10: Quick sort implementation

```
1  W ← warp size; t_id ← thread index;
2  Function quicksort-seq(Array A)
3      if |A| ≤ 2 then simple sorting of array A;
4      else
5          m ← |A| − 2; p ← A[|A| − 1];
6          i ← 0;
7          while i < m do
8              if A[i] ≤ p then ++i;
9              else swap elements A[i] and A[m]; −−m;
10         if A[m] ≤ p then m ← m + 1;
11         swap elements A[m] and A[|A| − 1];
12         quicksort-seq(A); quicksort-seq(A + m + 1);
13 Function quicksort-crev(Array A)
14     Equivalent to function quicksort-seq, except line 12 is replaced by:
15     if t_id < 2 then crev quicksort-crev(A'[t_id]);
```

as a clarification for the syntax used. Line 6 *does not* describe the execution of multiple instances of the `mergesort-seq` procedure. Only two new executions take place, one with $A'[0]$ half of the array A, and the other with the remaining $A'[1]$ elements.

Although the function name is tagged *sequential* (`mergesort-seq`), the function is valid in terms of ISPC. It just so happens that, be it implemented in plain C/C++, or as an ISPC kernel, the behavior is that of a purely sequential procedure – despite having the SIMD vector-processing capabilities at absolute disposal in the latter case. Now, back to the one line of thought that comprises our CREV solution, function `mergesort-crev` shows the simplicity of instilling effective parallelism in the once sequential algorithm. We simply replace line 6 by line 17. Upon reaching line 17, threads are filtered, in as much as to conform with the semantics of our solution: only up to two threads will be active when issuing the **crev** directive, and therefore, only two executions of `mergesort-crev` will take place – one with $A'[t_{id} = 0]$ and the other having $A'[t_{id} = 1]$ as input. During execution of each recursive invocation of `mergesort-crev`, all threads are active within the warp, thus holding sound the program: the execution is allowed to progress properly, as threads 0 and 1 will always be active upon entry in `mergesort-crev`, and the evaluation of the kernel is always syntatically correct.

The algorithm Quick-Sort, with all its distinctions w.r.t. Merge-Sort, does also port the same divide-and-conquer nature. Our CREV solution to Quick-Sort is analogous to the previous one described to Merge-Sort. We therefore leave the understanding

Algorithm 11: Bitonic Sort

```
1 if |A| < 2 then return;
2 if |A| = 2 and A[0] > A[1] then swap elements A[0] and A[1];
3 assert |A| to go up to the warp size;
4 load values from array A in private variables: val ← A[t_id];
5 generates a bitonic sequence;
6 sort bitonic sequence;
7 move sorted sequence into original container: A[t_id] ← val;
```

length	seq	laubch-def	launch-bi	crev-def	crev-bi
1	0.270	13.032	1.081	0.332	0.186
2	0.528	17.899	2.855	0.636	0.474
4	1.253	27.086	5.110	1.242	0.861
8	2.635	41.806	50.664	2.579	1.829
16	5.201	106.098	64.468	4.383	3.058
32	7.302	169.924	104.985	5.191	4.114
1	0.127	12.046	3.967	0.130	0.125
2	0.267	77.302	56.131	0.331	0.309
4	0.646	89.428	6.955	0.643	0.643
8	1.349	241.971	140.169	1.202	1.241
16	2.125	343.630	186.423	1.915	1.963
32	2.876	420.213	204.278	2.736	2.878

Table 5.1. Runtimes for sort algorithms on different input vector lengths. We wrote the mergesort and quicksort algorithms, both using `crev` and ISPC's `launch`, as well as relying on bitonic sort for fine-grain optimization. The results explicit how performant is our technique, in the sense we have got speedups at the cost of very smalls code changes. The first block of results is for the mergesort algorithm, wehereas the bottom half are results for the quicksort algorithm.

to the reader. Algorithm 10 describes the sequential and **crev**-based version of this sorting procedure. And we have also tried to improve the performance of both sorting algorithms. To do so, we replace the base step – which consisted of a small sequence of instructions to check for and swap unordered pairs – by an efficient implementation of bitonic sort. Bitonic sort is an essentially SIMD parallel algorithm, and thus fits more than adequetely into our CREV approach. Algorithm 11 presents a high-level description of the BitonicSort procedure, but implementation details are available in appendix A.1.

Our results show that CREV-based implementations of Merge- and Quick-Sort can be quite efficient, and even achieve better results with local algorithmic optimizations (bitonic sorting). Moreover, it is importatnt to realize how simple it was to put to work a CREV algorithm from an existing sequential procedure. Although we look

forward to achieving ever-improved runtimes, CREV main goal is to render efficient warp-synchronous programming a simple task.

5.6 Discussion

This chapter has presented an empirical evaluation of the implementation of CREV available in the ISPC compiler. To carry out this evaluation, we have used CREV to implement seven classic textbook algorithms. We believe that this chapter meets its goal: our algorithms have almost the same syntax of dynamic parallelism; however, they are as efficienty as warp-synchronous programs. And, contrary to warp-synchronous implementations, our algorithms do not suffer restrictions such as the inability to invoke functions within divergent regions. Indeed, the very fact that we can use vector instructions to boost the performance of quick-sort is a testimony of the usefulness of CREV. It would be very difficult to implement this algorithm in ISPC without any form of dynamic parallelism. Even though CREV is not dynamic parallelism per se, it affords the same syntax, but much better efficiency.

Chapter 6

A Scandalously Brief History of Vectorization

GPUs' increasing programmability and decreasing costs have made them very popular for the development of general purpose high performance applications [Nickolls and Dally, 2010]. This popularity has attracted the interest of programming language researchers, particularly for studies on control flow divergences. Therefore, the compiler-related literature contains a vast body of work describing analyses [Coutinho et al., 2011, Sampaio et al., 2012, Sampaio et al., 2013, Schaub et al., 2015] and optimizations [Coutinho et al., 2012, Coutinho et al., 2011, Zhang et al., 2011, Wu et al., 2016] that reduce the effects of divergences in GPGPU code. CREV is not a competitor of these analyses and optimizations. On the contrary, Call Re-Vectorization complements such techniques, giving programmers a tool that lets them deal with divergences at the software level.

Flynn's Taxonomy. From Chapter 1, recall our a quick discussion on some of the cases within Flynn's taxonomy. To refresh our memory on those concepts, let us go through them once more; this time, covering all of the terms proposed [Flynn, 1972]: SISD, MISD, SIMD, MIMD, and the SIMT variant. The **SISD** setting correspondes to a *Single Instruction* operating on a *Single Data* stream, and is examplified by regular serial computers. An **MISD** environment is hypothetically possible, but is often deemed impractical, as it would result in executing *Multiple Instructions* in a *Single Datum* – not much profitable in a general-purpose context [Duncan, 1990]. An **SIMD** architecture consists of a *Single Instruction* stream read, in lock-step, by a vector of n units, each using a different data source from the *Multiple Data* streams. The **SIMT** organization is a *Single Instruction* source executed by *Multiple Threads*, say n, and

each thread holding an SIMD lane of length m, i.e., every instruction runs $n \times m$ times. By times we shall refer to this latter model as either SIMT or multi-threaded (MT) programming. This is not originally part of Flynn's taxonomy, but is a useful case to bear in mind – an extension of the SIMD setting. Finally, we have the **MIMD**, which consists of multiple autonomous processors, and consequently *Multiple Instructions*, operate on *Multiple Data* streams.

A Scandalously Brief History of Vectorization The appearance of vector processor architectures dates back to the late 1960s and early 1970s, mostly in the form of processing machines designed to support massive mathematical computations (vector and matrix processing) [Watson, 1972, Lincoln, 1978]. A vector processor consists basically of multiple functional units working in parallel, each having a memory section from which to read input from. Such units could also be disposed as to pipeline tasks: the input data flows to the initial row of computing units, and outputs directly as input to the next row of units. Vector processors' computing units implemented mostly arithmetic and boolean operations – both for vectors and scalars [Duncan, 1990]. Due to the pipeline extension, such machines are hard to fit seemlessly within one of Flynn's classes: they neither present the SIMD lockstep execution, nor the asynchronous autonomy of the MIMD category. Some other machines, such as the famous Illiac IV, are categorized as SIMD, for they have vectors of processors executing in lockstep. Such computers are generally formed by a central CPU which feed instructions to the processing units; such units are connected via some network setting, by times enabling communication between processors and from processors to memory [Duncan, 1990]. This set up would later be present in massive parallel boards.

In the good old days, graphics hardware boiled down to the so-called VGA, or Video Graphics Array/Adapter: a display hardware first introduced with the IBM PS/2 line of computers in 1987 [Polsson, 2016]. A VGA card is basically an interface between the computer and its corresponding monitor. A program running on the CPU yields data that the target monitor uses to build and display images; it is the duty of a VGA card to keep the data the monitor will put up as pictures in an exhibition – the VGA functions as a *frame buffer*. There must, of course, be a means of communication between the VGA card and the CPU itself. The graphics card is, thus, plugged to the motherboard and, upon having the required drivers, data transferring between those two devices become possible. It is also noticeable the runtime involved in this process: subject to the memory hierarchy, data flows from the CPU to its main Random Access Memory (RAM); subsequently, the operational system puts the driver software to use, allowing the VGA card to copy data from the computer's RAM to VGA's frame

buffer. The runtime is, no doubt, dependent on the hardware's throughput/bandwidth capabilities, but as a rule of thumb, the less communication involved, the better.

Still, all the actual processing took place at the CPU, and was, therefore, implemented in software, which is costly both in terms of code development and eventual data transfer latencies. The programming downside, one may suggest to tackle with library code, which is reasonable, but brings forth compatibility issues in face of VGA devices from different manufactures – or even of versions of the same product. For memory access and communication related aspects, there are quite a few well-known optimizations, such *double buffering* and *page flipping* [Brackeen, 1996]. Double buffering consists in generating all the data that must be copied to the graphic device on the CPU memory first, and only then issuing the copy – to avoid multiple (unnecessary) requests. Page flipping, in its turn, focus on having visually smooth frame transitions: the data is written to the frame buffer at a memory location not being used to refresh images on the screen; and only after a whole new image has been built the frame buffer it is that frame pointers are updated, and the display device prints out fresh content.

Despite its importance as an interface between CPU and display device, graphics cards were still to be put to greater use. Optimizations help, but fall short in view of what could be done if the data had not to travel from CPU to graphics device. Of course the device must read *some* input information, yet the processing should ideally happen on the VGA, and the generated output be already dispatched to the VGA's frame buffer. But what would someone want to run at the hardware level? The answer: a series of well-defined functions that are applied to an image before it can be properly displayed, e.g., vertex operations, primitive assembly, rasterization, fragment operations, and composition into a final image [Owens et al., 2007]. Support for specific computations was added, and whence surged the closest ancestors of nowadays' video cards. Such boards, also referred to as *accelarators*, were indeed accelarating computations once performed entirely on the CPU: initial boards with processors allowed developers to run well-defined image processing tasks, instead of having to resort to CPU computation followed by data copies.

An interesting fact about the nature of those operations is their embarrassing parallelism: given an input image, usually represented as n-dimensional arrays, a.k.a. *textures*, it contains data and control independency w.r.t. its processing elements (e.g., vertices, triangles, fragments) [Moya et al., 2005]. Such nature was also found in the old SIMD machines, such as Illiac IV, and is a clear call for vectorization. As there was a definite demand for parallel computational power, general-purpose computing evolved on top of graphics boards: more transistors and new functionalities were added to those boards. However, it was only around 2001 such

Year	Transistors	Model	Tech	Max GFlops
1999	25M	GeForce 256	DX7, OpenGL	8
2001	60M	GeForce 3	Programmable Shader	10
2002	125M	GeForce FX	CG programs	29
2006	681M	GeForce 8800	C for CUDA	576
2008	1.4G	GeForce GTX 285	IEEE FP	1063
2010	3.0G	GeForce GTX 480	Cache, C++	1345

Table 6.1. A scandalously brief timeline on GPUs. It is clear both the number of transistors and exponentially increasing maximum GFlops delivered by top-performance graphics processing boards throughout the past ten years.

boards became practical and popular. Its wide-spread was due to the advent of both programmable shaders and floating point support on the graphics cards. Matrix- and general vector-based problems were easily implemented on top of this hardware, which significantly accelerates the computation; one of the first common scientific programs to run faster on GPUs than CPUs was an implementation of LU factorization(2005) [Du et al., 2012]. In addition, even a single GPU-CPU framework provided advantages that multiple CPUs on their own would not offer, due to the specialization in each chip [Mittal and Vetter, 2015]. Graphics cards' manufacturers oblige such parallelism by jamming massive amounts of processing cores on their boards. The growth on the computational power of this hardware is reflected by the number of transistors, as shown in Table 6.1 [Pereira, 2014, Hardware-INFOS, 2017].

Forms of Parallelism. General-purpose graphics processor units have four forms of parallelism. The hardware pipeline is divided into hundreds of single cycle stages to increase the throughput and the GPU clock frequency; this technique is named *pipeline parallelism*, and relies on overlapping computations at the many different processing units at different stages. For example, if an instruction at procesing unit A requires some data to be loaded from memory, it may cause the execution of the instruction to be stalled; but as long as there is any further operation not depending on the result of the memory load, such computation may take place at a different stage of the pipeline. Moreover, the pipeline stages are replicated to process in parallel multiple vertices, triangles and fragments. This is the basic proposal from SIMD architectures, and such approach, namely *data parallelism*, consists in supplying developers more processing units at a time, each unit drawing data from its own source stream [Moya et al., 2005].

Another form of parallelism bestowed in GPUs is *multi-threading*: multiple processing elements are stored and processed concurrently to hide memory latencies in a specific stage or processing unit. At the programming point of view, multi-threading

comprises a series of independent processes, each reading input data from a distinct data stream, and such behavior is achieved by launching different kernels in parallel. Finally, instructions can be re-ordered and combined into groups which may executed in parallel without changing the result of the program. This way, GPUs also allow its processing units to execute independent instructions in parallel, thus conveying the so-called *instruction level parallelism* [Moya et al., 2005].

Vector Processor Languages. There are many different approaches towards exploiting the computational powers of vector processors, be it CPU- or GPU-based. Languages may be *machine assembly* code, in which case the programmer directly describe the sequence of operations desired; or *high-level programming* abstractions of such machine code. We now give a quick overview on some well-known vector-based programming languages.

C is not uncommonly chosen as base language for many vector-processing extensions. C is originally a scalar language, with no intrinsics for data distribution nor parallel execution. As C is but a step up from assembly code, it gives programmers a proximity with the assembly code and memory acesses not possible in higher-level languages. Since vector-processing extensions involve many vector copies and accessess, it is convenient to use languages such as C. Nevertheless, it is possible to implement efficient vectorized code in C, by inlining assembly instructions for vector-processors.

SSE, or Streaming SIMD Extensions, is an SIMD instruction set extension to the x86 architecture, designed by Intel in 1999. SIMD instructions can greatly increase performance when exactly the same operations are to be performed on multiple data objects. Typical applications are digital signal processing and graphics processing. SSE was subsequently expanded by Intel to SSE2, SSE3, SSSE3, and SSE4 [Wikipedia, 2017b]. *AVX*, or Advanced Vector Extensions, is another extension to the x86 instruction set architecture for microprocessors from Intel and AMD, and was originally proposed by Intel, in March 2008. AVX provides new features, new instructions and a new coding scheme, and has later been extended by AVX2 and AVX-512 [Wikipedia, 2017a].

Intel's SPMD Program Compiler, *ISPC* for short, is both a language and a compiler developed by Intel. Its code is open source and focus on allowing programmers to write efficient parallel multi-threaded programs for CPUs. There is also an absolute interest in minimizing the complexity of code writing, as it provides abstractions for SIMD and SIMT programming. The code is lowered to Intel's SSE/AVX assembly languages as the programmer see fit. ISPC is an extension of the C programming language,

and also supplies coders with C++ library interfaces and compilation infrastructure, i.e., ISPC kernels may be used in conjunction with either C or C++ [Intel, 2016].

CUDA is a parallel computing platform and programming model invented by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU). Its software layer grants developers direct access to the GPU's virtual instruction set for the execution of compute kernels. GPGPU programming was far from easy at the early years of graphics boards, as developers had to map scientific calculations onto problems that could be represented by triangles and polygons. It was in 2013, a team of researchers led by Ian Buck unveiled *Brook*, the first widely adopted programming model to extend C with data-parallel constructs. The language ported concepts such as streams, kernels and reduction operators, and its compiler and runtime system exposed the GPU as a general-purpose processor in a high-level language. Most importantly, Brook programs were not only easier to write than hand-tuned GPU code, they were seven times faster than similar existing code. NVIDIA then invited Ian Buck to join the company and start evolving a solution to seamlessly run C on the GPU, which culminated in CUDA in the year 2006 [NVIDIA, 2017].

Finally, *OpenCL* views a computing system as consisting of a number of compute devices, which might be central processing units (CPUs) or accelerators such as graphics processing units (GPUs), attached to a host processor (a CPU). OpenCL defines a C-like language for writing programs, and akin to CUDA, functions executed on an OpenCL device are called kernels [Howes and Munshi, 2015]. Processing nodes are defined according to the hardware setting, but should correspond to the number of SIMD processing lanes available at runtime. It is hard to define what a processing node is, even within a CPU, as it may have a static number of cores, but enhance its power virtually via hyperthreading [Gaster et al., 2012]. A single kernel execution can run on all or many of the processing nodes, in parallel. In addition to its C-like programming language, OpenCL defines an application programming interface (API) that allows programs running on the host to launch kernels on the compute devices and manage device memory, which is (at least conceptually) separate from host memory. Programs in the OpenCL language are intended to be compiled at run-time, so that OpenCL-using applications are portable between implementations for various host devices [Stone et al., 2010]. The OpenCL standard defines host APIs for C and C++, but third-party APIs exist for other programming languages and platforms such as Python [Klöckner et al., 2012], Java and .NET [Gaster et al., 2012].

Chapter 7

Final Thoughts

Primitives such as warp vote and shuffle have given experts the possibility of writing efficient SIMD code, by programming from the point of view of one warp. This coding style has been used in CUB and many other CUDA libraries[1]. However, warp-synchronous code does not play well with branch divergence. Most warp-synchronous algorithms require all threads in a warp to be active. This is a problem for the common usage scenario of a simple MT-style CUDA kernel that calls warp-synchronous library functions. It is our vision that the application developer writing the kernel should not be concerned with the internal implementation of library functions, and should be able to call any function inside divergent program regions. To meet the demands of this vision, this paper has introduced the notion of Call Re-Vectorization(CREV). We have described the building blocks necessary to implement CREV. Looking towards compatibility with future hardware, we have proposed low-level primitives with well-defined semantics and a high-level interface, the **crev** idiom, that makes programmer intent explicit. Thus, our notion of CREV does not rely implicitly on current hardware behavior, which might eventually change. We have implemented CREV into ISPC, using Intel instructions, and have shown how to code irregular algorithms in this environment. Our implementations are not only clearer than non-CREV based approaches, but also more efficient, as they balance work among inactive warp threads.

We believe that this work opens up several different research directions. For instance, we have been using CREV manually. That is to say: thus far, the developer must manually annotate functions with the CREV high-level keyword. However, nothing hinders a compiler from providing static analyses that add this primitive directly onto source code already in place. Whoever decides to follow this direction will have a number of concerns to worry about, such as: which functions to vectorize,

[1] See `https://nvlabs.github.io/cub/`

which thresholds to consider when invoking r-functions, how to measure the benefit of function re-vectorization, etc.

List of Terms

I

ISPC

Intel's *Single Program Multiple Data* compiler. An open-source compiler available online at `https://ispc.github.io/`. xvii, xix, xxii, xxiii, xxv, xxvii, xxviii, 1, 3, 5, 10, 21, 22, 24, 29–41, 43–52, 57–59, 61, 63, 66, 69, 70

S

SIMD

An SIMD architecture consists of a *Single Instruction* stream read, in lock-step, by a vector of n units, each using a different data source from the *Multiple Data* streams. xvii, xix, xxi, xxii, 1–5, 7–21, 24, 25, 27, 29–31, 33, 34, 40, 47, 49–51, 53–59, 61, 63, 64, 69

SIMT

An SIMT machine has a *Single Instruction* source executed by *Multiple Threads*, say n, and each thread holding an SIMD lane of length m, i.e., every instruction runs $n \times m$ times. xxi, 2, 3, 5, 11, 13, 29, 36, 49, 53, 54, 57, 61, 66

SPMD

SPMD, or *Single Program, Multiple Data*, identifies architectures with a *Single Program* executed independently by many processing units, each with its own data stream. xvii, xix, 3, 5, 29–31, 33, 38, 41, 57, 61, 66

Bibliography

[Bellman, 1958] Bellman, R. (1958). On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87--90.

[Bougé and Levaire, 1992] Bougé, L. and Levaire, J.-L. (1992). Control structures for data-parallel SIMD languages: semantics and implementation. *Future Generation Computer Systems*, 8(4):363--378.

[Brackeen, 1996] Brackeen, D. (1996). 256-color vga programming in c. `http://www.brackeen.com/vga/unchain.html`. Accessed: 2017-02-19.

[Brodman et al., 2014] Brodman, J., Babokin, D., Filippov, I., and Tu, P. (2014). Writing scalable SIMD programs with ISPC. In *WPMVP*, pages 25--32. ACM.

[Catanzaro, 2012] Catanzaro, B. (2012). Trove cuda library. `https://github.com/bryancatanzaro/trove`. Accessed: 2017-02-02.

[Chen and Shen, 2015] Chen, G. and Shen, X. (2015). Free launch: optimizing GPU dynamic kernel launches through thread reuse. In *Micro*, pages 407--419. ACM.

[Coutinho et al., 2011] Coutinho, B., Sampaio, D., Pereira, F. M. Q., and Jr., W. M. (2011). Divergence analysis and optimizations. In *PACT*, pages 320--329. IEEE.

[Coutinho et al., 2012] Coutinho, B., Sampaio, D., Pereira, F. M. Q., and Jr., W. M. (2012). Profiling divergences in GPU applications. *Concurrency and Computation: Practice and Experience*, 1(10.1002/cpe.285-15):1--15.

[Dias et al., 2016] Dias, V., Moreira, R., Meira, W., and Guedes, D. (2016). Diagnosing performance bottlenecks in massive data parallel programs. In *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*, pages 273--276. IEEE.

[DiMarco and Taufer, 2013] DiMarco, J. and Taufer, M. (2013). Performance impact of dynamic parallelism on different clustering algorithms. *Modeling and Simulation for Defense Systems and Applications*, 8752(VIII):87520E–87520E:8.

[Du et al., 2012] Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G., and Dongarra, J. (2012). From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Computing*, 38(8):391--407.

[Duncan, 1990] Duncan, R. (1990). A survey of parallel computer architectures. *Computer*, 23(2):5--16.

[Erdos and Renyi, 1959] Erdos, P. and Renyi, A. (1959). On random graphs. I. *Publicationes Mathematicae*, 6(1):290--297.

[Farrell and Kieronska, 1996] Farrell, C. A. and Kieronska, D. H. (1996). Formal specification of parallel SIMD execution. *Theo. Comp. Science*, 169(1):39--65.

[Ferrante et al., 1987] Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319--349.

[Flynn, 1972] Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948--960.

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

[Gaster, 2014] Gaster, B. (2014). An execution model for OpenCL 2.0. Technical report 2014-02, Computer Sciences.

[Gaster et al., 2012] Gaster, B., Howes, L., Kaeli, D. R., Mistry, P., and Schaa, D. (2012). *Heterogeneous Computing with OpenCL: Revised OpenCL 1.* Newnes.

[Han and Abdelrahman, 2013] Han, T. D. and Abdelrahman, T. S. (2013). Reducing divergence in GPGPU programs with loop merging. In *GPGPU*, pages 12--23. ACM.

[Hardware-INFOS, 2017] Hardware-INFOS (2017). Grafikkarten - nvidia. `http://www.hardware-infos.com/grafikkarten_nvidia.php`. Accessed: 2017-03-02.

[Hoogvorst et al., 1991] Hoogvorst, P., Keryell, R., Paris, N., and Matherat, P. (1991). POMP or how to design a massively parallel machine with small developments. In *PARLE*, pages 83--100. Springer.

[Howes and Munshi, 2015] Howes, L. and Munshi, A. (2015). The opencl specification.

[Intel, 2016] Intel (2016). Ispc documentation. `https://ispc.github.io/ispc.html`. Accessed: 2017-02-09.

[Jones, 2014] Jones, S. (2014). Introduction to dynamic parallelism – Invited Talk. In *GPU Technology Conference*, pages 1--33. NVIDIA.

[Khorasani et al., 2015] Khorasani, F., Gupta, R., and Bhuyan, L. N. (2015). Efficient warp execution in presence of divergence with collaborative context collection. In *Micro*, pages 204--215. ACM.

[Klöckner et al., 2012] Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., and Fasih, A. (2012). Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation. *Parallel Computing*, 38(3):157--174.

[Knuth et al., 1977] Knuth, D. E., Jr., J. H. M., and Pratt, V. R. (1977). Fast pattern matching in strings. *Journal of Computing*, 6(2):323--350.

[Kushner, 2013] Kushner, D. (2013). The real story of stuxnet. *ieee Spectrum*, 3(50):48--53.

[Lincoln, 1978] Lincoln, N. (1978). A safari through the control data star-100 with gun and camera. In *Proc. AFIPS NCC*, volume 47.

[MasPar, 1992] MasPar (1992). *MasPar Programming Language (ANSI C compatible MPL) Reference Manual*.

[Merrill and Grimshaw, 2011] Merrill, D. and Grimshaw, A. (2011). High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Proceessing Letters*, 21(2):245--272.

[Mittal and Vetter, 2015] Mittal, S. and Vetter, J. S. (2015). A survey of cpu-gpu heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, 47(4):69.

[Moreira et al., 2017] Moreira, R., Collange, S., and Pereira, F. (2017). Function call re-vectorization. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

[Moreira and Tymburibá, 2016] Moreira, R. E. and Tymburibá, M. (2016). Riprop-deducer. `http://cuda.dcc.ufmg.br/rip-rop-deducer/`. Accessed: 2017-02-07.

[Moreira et al., 2015] Moreira, R. E., Tymburibá, M., and Quintão Pereira, F. M. (2015). Inferência estática da frequência máxima de instruções de retorno para

detecção de ataques rop. In *Proceedings of the 2015 Brazilian Symposium on Information and Computational Systems Security*, pages 2–15. SBC.

[Moreira et al., 2016] Moreira, R. E. A., Collange, S., and Quintão, F. M. (2016). Definição semântica de blocos everywhere para programação simd.

[Moya et al., 2005] Moya, V., Gonzalez, C., Roca, J., Fernandez, A., and Espasa, R. (2005). Shader performance analysis on a modern gpu architecture. In *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on*, pages 10--pp. IEEE.

[Munshi et al., 2011] Munshi, A., Gaster, B., Mattson, T. G., Fung, J., and Ginsburg, D. (2011). *OpenCL Programming Guide*. Addison-Wesley, 1st edition.

[Nickolls and Dally, 2010] Nickolls, J. and Dally, W. J. (2010). The GPU computing era. *IEEE Micro*, 30:56--69.

[Novak, 2015] Novak, R. (2015). Loop optimization for divergence reduction on GPUs with SIMT architecture. *IEEE Transactions on Parallel and Distributed Systems*, 26(6):1633--1642.

[NVIDIA, 2016] NVIDIA (2016). Cub cuda library. `http://nvlabs.github.io/cub/group___warp_module.html`. Accessed: 2017-02-02.

[NVIDIA, 2017] NVIDIA (2017). Cuda parallel programming and computing platform. `http://www.nvidia.com/object/cuda_home_new.html`. Accessed: 2017-02-22.

[Owens et al., 2007] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T. J. (2007). A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80--113. Wiley Online Library.

[Pereira, 2014] Pereira, F. M. Q. (2014). Divergence analysis. `http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/DivergenceAnalysis.pdf`. Accessed: 2017-02-20.

[Pharr and Mark, 2012] Pharr, M. and Mark, W. R. (2012). ISPC: A SPMD compiler for high-performance CPU programming. In *InPar*, pages 1--13. IEEE.

[Polsson, 2016] Polsson, K. (2016). Chronology of ibm personal computers. `http://pctimeline.info/ibmpc/ibm1987.htm`. Accessed: 2017-02-19.

[Rose and Steele, 1987] Rose, J. and Steele, G. (1987). C*: An extended C language for data parallel programming. In *ICS*.

[Sampaio et al., 2013] Sampaio, D., de Souza, R. M., Collange, S., and Pereira, F. M. Q. (2013). Divergence analysis. *ACM Trans. Program. Lang. Syst.*, 35(4):13.

[Sampaio et al., 2012] Sampaio, D., Martins, R., Collange, S., and Pereira, F. M. Q. (2012). Divergence analysis with affine constraints. In *SBAC-PAD*, pages 67--74. IEEE.

[Sanders and Kandrot, 2010] Sanders, J. and Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 1st edition.

[Schaub et al., 2015] Schaub, T., Moll, S., Karrenberg, R., and Hack, S. (2015). The impact of the simd width on control-flow and memory divergence. *TACO*, 11(4):54:1--54:25.

[Sodani et al., 2016] Sodani, A., Gramunt, R., Corbal, J., Kim, H.-S., Vinod, K., Chinthamani, S., Hutsell, S., Agarwal, R., and Liu, Y.-C. (2016). Knights Landing: Second-generation Intel Xeon Phi product. *IEEE Micro*, 36(2):34--46.

[Spink, 2016] Spink, T. (2016). Cgo 2016 - awards. `http://cgo.org/cgo2016/awards/`. Accessed: 2017-02-07.

[Stone et al., 2010] Stone, J. E., Gohara, D., and Shi, G. (2010). Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66--73.

[Tymburibá et al., 2015] Tymburibá, M., Moreira, R. E., and Quintão Pereira, F. M. (2015). Riprop: A dynamic detector of rop attacks. In *Proceedings of the 2015 Brazilian Congress on Software: Theory and Practice*, pages 2–15. SBC.

[Tymburibá et al., 2016] Tymburibá, M., Moreira, R. E., and Quintão Pereira, F. M. (2016). Inference of peak density of indirect branches to detect rop attacks. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 150--159. ACM.

[Wang et al., 2015] Wang, J., Rubin, N., Sidelnik, A., and Yalamanchili, S. (2015). Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on gpus. In *ISCA*, pages 528--540. ACM.

[Wang et al., 2016] Wang, J., Rubin, N., Sidelnik, A., and Yalamanchili, S. (2016). LaPerm: Locality aware scheduler for dynamic parallelism on GPUs. In *ISCA*.

[Wang and Yalamanchili, 2014] Wang, J. and Yalamanchili, S. (2014). Characterization and analysis of dynamic parallelism in unstructured GPU applications. In *IISWC*, pages 51--60. IEEE.

[Watson, 1972] Watson, W. (1972). The ti asc: a highly modular and flexible super computer architecture. In *Proceedings of the December 5-7, 1972, fall joint computer conference, part I*, pages 221--228. ACM.

[Wikipedia, 2017a] Wikipedia (2017a). Advanced vector extensions. `https://en.wikipedia.org/wiki/Advanced_Vector_Extensions`. Accessed: 2017-02-22.

[Wikipedia, 2017b] Wikipedia (2017b). Streaming simd extensions. `https://en.wikipedia.org/wiki/Advanced_Vector_Extensions`. Accessed: 2017-02-22.

[Wu et al., 2016] Wu, J., Belevich, A., Bendersky, E., Heffernan, M., Leary, C., Pienaar, J., Roune, B., Springer, R., Weng, X., and Hundt, R. (2016). Gpucc: An open-source gpgpu compiler. In *CGO*, pages 105--116. ACM.

[Yang and Zhou, 2014] Yang, Y. and Zhou, H. (2014). CUDA-NP: realizing nested thread-level parallelism in GPGPU applications. In *PPoPP*, pages 93--106. ACM.

[Zhang et al., 2011] Zhang, E. Z., Jiang, Y., Guo, Z., Tian, K., and Shen, X. (2011). On-the-fly elimination of dynamic irregularities for GPU computing. In *ASPLOS*, pages 369--380. ACM.

Appendix A

ISPC-CREV Code

We here leave some code we find relevant to the reader. The implementations in this section are ISPC-CREV code, and should thus be compiled with our extend compiler, available online at `http://cuda.dcc.ufmg.br/swan`.

A.1 Bitonic Sort

Bitonic Sort is a well-known parallel sorting algorithm. We implement Bitonic Sort in our ISPC-CREV extend language to improve the performance of the Merge- and Quick-Sort algorithms, both implemented using the **crev** keyword. The following `BitonicSort` kernel is SIMD, i.e., all threads within a warp must be active upon call to this function. We use **crev** to guarantee such property holds, validating the execution of the program.

```
static int bitonic_mask[6][2][8] = {
    { {0, -1, -2, 3, 4, -5, -6, 7}, {1, 0, 3, 2, 5, 4, 7, 6} },
    { {0, 1, -2, -3, -4, -5, 6, 7}, {2, 3, 0, 1, 6, 7, 4, 5} },
    { {0, -1, 2, -3, -4, 5, -6, 7}, {1, 0, 3, 2, 5, 4, 7, 6} },
    { {0, 1, 2, 3, -4, -5, -6, -7}, {4, 5, 6, 7, 0, 1, 2, 3} },
    { {0, 1, -2, -3, 4, 5, -6, -7}, {2, 3, 0, 1, 6, 7, 4, 5} },
    { {0, -1, 2, -3, 4, -5, 6, -7}, {1, 0, 3, 2, 5, 4, 7, 6} }
};

void set_order(int * uniform array, uniform int len) {
    if (len == 2 && array[0] > array[1]) {
        uniform int aux = array[1];
        array[1] = array[0];
        array[0] = aux;
    } else if (len > 2) print("ERROR!\n");
```

```
}

void bitonic_sort(int * uniform array, uniform int len) {

    if (len < 2) return;
    if (len == 2 && array[0] > array[1]) {
        uniform int aux = array[1];
        array[1] = array[0];
        array[0] = aux;
    }

    if (len > programCount) { print("ERROR!\n"); return; }

    // Fetching values
    varying int val = (1 << 30), tmp = (1 << 30);
    if (programIndex < len) val = array[programIndex];

    // Bitonic sorting masks
    varying int cmp1 = bitonic_mask[0][0][programIndex];
    varying int cmp2 = bitonic_mask[1][0][programIndex];
    varying int cmp3 = bitonic_mask[2][0][programIndex];

    varying int val1 = bitonic_mask[0][1][programIndex];
    varying int val2 = bitonic_mask[1][1][programIndex];
    varying int val3 = bitonic_mask[2][1][programIndex];

    varying int cmp4 = bitonic_mask[3][0][programIndex];
    varying int cmp5 = bitonic_mask[4][0][programIndex];
    varying int cmp6 = bitonic_mask[5][0][programIndex];

    varying int val4 = bitonic_mask[3][1][programIndex];
    varying int val5 = bitonic_mask[4][1][programIndex];
    varying int val6 = bitonic_mask[5][1][programIndex];

    // Generating a bitonic sequence

    // Step 1: distance 1
    tmp = shuffle(val, val1);
    if (programIndex == cmp1) val = (val > tmp) ? tmp : val;
    else val = (val < tmp) ? tmp : val;

    // Step 2.1: distance 2
    tmp = shuffle(val, val2);
    if (programIndex == cmp2) val = (val > tmp) ? tmp : val;
```

```
    else val = (val < tmp) ? tmp : val;

    // Step 2.2: distance 4
    tmp = shuffle(val, val3);
    if (programIndex == cmp3) val = (val > tmp) ? tmp : val;
    else val = (val < tmp) ? tmp : val;

    // Sorting a bitonic sequence

    // Step 3.1: distance 1
    tmp = shuffle(val, val4);
    if (programIndex == cmp4) val = (val > tmp) ? tmp : val;
    else val = (val < tmp) ? tmp : val;

    // Step 3.2: distance 2
    tmp = shuffle(val, val5);
    if (programIndex == cmp5) val = (val > tmp) ? tmp : val;
    else val = (val < tmp) ? tmp : val;

    // Step 3.3: distance 4
    tmp = shuffle(val, val6);
    if (programIndex == cmp6) val = (val > tmp) ? tmp : val;
    else val = (val < tmp) ? tmp : val;

    // Storing sorted array
    if (programIndex < len) array[programIndex] = val;

}
```