FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Parallelisierung der Kommutationseigenschaft für Funktionen über kleinen Wertebereichen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Markus Scherer, B.Sc.
Matrikelnummer 1028046

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Gernot Salzer

Wien, 19. April 2016

Markus Scherer	Gernot Salzer

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Parallelizing the Commutation Property for Functions over Small Domains

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computational Intelligence

by

Markus Scherer, B.Sc.
Registration Number 1028046

to the Faculty of Informatics

at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Gernot Salzer

Vienna, 19th April, 2016

Markus Scherer

Gernot Salzer

Erklärung zur Verfassung der Arbeit

Markus Scherer, B.Sc.
Wien 1040
Große Neugasse 22-24/2/14

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 19. April 2016

Markus Scherer

Danksagung

An dieser Stelle möchte ich allen Menschen danken, die zum Gelingen meiner Diplomarbeit beigetragen haben:

An erster Stelle meinem Betreuer Prof. Gernot Salzer, der mein Interesse an der algebraischen Betrachtung informatischer Problemstellungen genährt hat, und dessen Tür mir bei meinen vielen Fragen immer offen stand.

Weiters Dr. Mike Behrisch, der geduldig meine Fragen zur Klontheorie beantwortete.

Prof. Jesper Larsson Träff und Markus Levonyak von der Forschungsgruppe für paralleles Rechnen gebührt mein Dank für die Zurverfügungstellung der technischen Infrastruktur, die ich für meine Diplomarbeit benutzt habe.

Nicht zuletzt möchte ich meiner Familie danken, insbesondere meinen Eltern und meiner Tante Gitti, die mich sowohl moralisch als auch finanziell während meines Studiums unterstützt haben.

Acknowledgements

At this point I would like to thank everyone who contributed to the success of my diploma thesis:

First of all Prof. Gernot Salzer, who nurtured my interest in the algebraic view on problems in computer science and who always welcomed me and my questions, of which there were many.

Furthermore Dr. Mike Behrisch, who patiently answered all my questions on clone theory.

Special thanks go to Prof. Jesper Larsson Träff and Markus Levonyak of the research group for parallel computing for providing the technical infrastructure which I used for my diploma thesis.

Last but not least, I want to thank my family, especially my parents and my aunt Gitti, who supported me morally as well as financially.

Kurzfassung

Funktionale und relationale Klone, das heißt Mengen von Funktionen oder Relationen, die unter einer Form von Komposition abgeschlossen sind, sind ein wichtiges Werkzeug zur Untersuchung von Eigenschaften von Algebren. Ein Verständnis von Klonen und den Beziehungen zwischen ihnen ist beispielsweise hilfreich für das Beweisen komplexitätstheoretischer Resultate für das Constraint-Satisfaction-Problem.

Eine spezielle Form von Klonen sind sogenannte primitiv-positive Klone, von denen es über jedem Wertebereich nur eine endliche Anzahl gibt. Diese Klone sind theoretisch berechenbar, in der Realität ist der Suchraum allerdings in den meisten Fällen zu groß.

Das Ziel dieser Diplomarbeit ist es zu untersuchen, in welchem Ausmaß Parallelisierung die Grenzen dessen, was als praktisch berechenbar gilt, erweitern kann.

Wir werden daher unterschiedliche Ansätze zur Berechnung von primitiv-positiven Klonen entwickeln und sie gegeneinander evaluieren. Bei den untersuchten Ansätzen handelt es sich um Parallelismus auf der Anweisungsebene, klassische Nebenläufigkeit und Grafikkartenprogrammierung.

Resultat der Diplomarbeit wird, neben der Auswertung der verschiedenen Ansätze, außerdem ein Programm sein, das für Experimente im Bereich der Klontheorie genutzt werden kann.

Abstract

Functional and relational clones, that is sets of functions or relations closed under some form of composition, are an important tool for investigating properties of algebras. Understanding clones and the relation between them can for example help us to prove complexity-theoretic results for Constraint Satisfaction Problems.

A special form of clones are so-called primitive positive clones, of which there are only finitely many over each domain. These clones are computable in principle, but in reality the search space too huge in most cases. The aim of this thesis is to investigate to what extent parallelization can push the boundaries of what is thought to be practically computable.

We will therefore develop different parallel approaches to calculate primitive positive clones – using instruction level parallelism, classical multithreading and GPU programming – and evaluate them against each other.

Result of this thesis will be, in addition to the evaluation of the different approaches, a program which can be used for experiments in the context of clone theory.

Contents

CHAPTER 1

Introduction

Motivation Functional and relational clones, that is sets of functions or relations which are closed under some form of composition, are an important tool for investigating properties of algebras.

Knowing the clones over a certain domain and the relation between them can make it easier to prove certain properties. Two accessible articles demonstrating this are [BCRV03] and [BCRV04].

In [BCRV03] a succinct proof of [Lew79] is presented: every set of Boolean connectives which is able to represent $x \wedge \neg y$ makes the corresponding SAT problem NP-complete; all other sets of connectives can only produce SAT instances which are polynomial time solvable.

In [BCRV04] Schaefer's important classification of subclasses of the SAT problem [Sch78] is tackled by using clones. Furthermore a correspondence between functional and relational clones is established which allows us to limit our investigation to functional clones.

In [JCG97] necessary and sufficient conditions for the generalization of SAT, the Constraint Satisfaction Problem (CSP), are given by examining clones. Similar results have been presented in [Zad07].

A special form of clones are so-called primitive positive clones, a concept introduced by [Kuz79] (for a modern and accessible presentation of these results see [Her08]). Kuznetsov's investigation, which considered parametric clones over a two-element domain, was extended by Danil'chenko to three-element domains in her dissertation and related articles [Dan77, Dan79].

Primitive positive clones are those clones which can be defined as the centralizer of a function set. The centralizer of a function set consists of all functions which *commute* with all functions of this set.

Problem Statement The commutation property is very demanding from a computational point of view. Given two functions of arity m and n over a domain with d elements, we have to check d^{mn} matrices of variables to prove commutation in the worst case. Each of these matrices again needs $m + n + 2$ function evaluations.

Already for functions of arity 4 over a 4-element domain, naive implementations can take several hours for such a check. The way commutation is computed, however, lends itself to parallelization. We will therefore implement a program which facilitates parallelization in order to check functions for commutation as fast as possible.

Aim of the Work Using different parallelization techniques we want to evaluate how far the borders of what is thought to be *practically* computable can be extended . Since the techniques we are using differ in many points, we want to learn about and document the benefits an disadvantages of each approach.

Furthermore, we will publish the code under the following URL so that researchers may make use of our program.

`https://github.com/markusscherer/commutation-test`

Methodology Following the model proposed in [RJ10], we will optimize our performance by structuring our computation in a layered way: on the CPU-level we will use SIMD instructions to speed up our commutation test, on a higher level we will use different multithreading technologies.

To facilitate experimentation, we will implement the SIMD-based code with so called *policies*. In the end, this will allow us to analyze which optimization influenced the performance in which way and to which degree. We will also use two different compilers and evaluate the binaries produced by them against each other.

In addition to an implementation on conventional hardware, we will take the huge potential of specialized hardware into account which is promised by articles such as [ND10] by implementing our commutation test also on graphics processing units (GPUs).

After implementing the different approaches, we will evaluate them against each other and draw our conclusions.

Structure of the Work In the first part of Chapter 2 we will introduce basic concepts of clone theory. The second part surveys some results which allow us to prune our search space. In Chapter 3 we will describe our central design pattern, policy-based-design, which allows us to experiment with different optimizations. Chapter 4 is divided in three parts: in each we will present on of our different approaches in detail, together with an introduction to the used concepts. Each such presentation is directly followed by a discussion of the performance. In Chapter 5 we draw our conclusions, discuss problems for the future and present some of the pitfalls we encountered, so that future researchers who are faced with similar problems do not have to repeat our mistakes.

CHAPTER 2

Mathematical Background

2.1 Basic Definitions

We refer to a set $\{0, \ldots, k-1\}$ as A_k. The set of all functions over A_k is called P_k. A function $f \in P_k$, $f\colon A^n \mapsto A$, is said to have arity n, in symbols $ar(f) = n$. Given a function f with $ar(f) = n$ and a tuple $\vec{x} \in A^n$ we may write $f(\vec{x})$ instead of $f(x_1, \ldots, x_n)$. When talking about the arguments of a function, we will sometimes call x_i more significant than x_{i+1} and x_1 the most respectively x_n the least significant argument. A subset $R \subseteq A^n$ is called an n-ary[1] relation over A.

Definition 2.1.1 (Projection)**.** A function π_i^n, $A^n \mapsto A$, for which $\pi_i^n(x_1, \ldots, x_n) = x_i$ holds, is called *projection*. The set of all projections is denoted by *Pr*. We note in passing, that the identity function $id(x) = x$ is just a special case of a projection.

Definition 2.1.2 (Composition)**.** Given a function f, with $ar(f) = p$, and p functions g_i with $ar(g_i) = q$, the function $h(x_1, \ldots, x_q) = f(g_1(x_1, \ldots, x_q), \ldots, g_p(x_1, \ldots, x_q))$ is called the *composition* of f and g_i, in symbols $h = f \circ g_i$.

Definition 2.1.3 (Clone)**.** A set of functions Σ is called a *clone*, if it contains all projections and is closed under composition. Given an arbitrary set of functions Σ', $\langle\Sigma'\rangle$ denotes the smallest clone containing Σ'. A subset $\Sigma'' \subseteq \langle\Sigma'\rangle$ is called a *base* of $\langle\Sigma'\rangle$ if $\langle\Sigma''\rangle = \langle\Sigma'\rangle$ and for each proper subset $\Sigma''' \subset \Sigma''$ we have $\langle\Sigma'''\rangle \neq \langle\Sigma\rangle$.

Definition 2.1.4 (Expressibility)**.** Given a function f and set of functions Σ, we say f is *(explicitly) expressibly* by Σ, in symbols $f \leq_e \Sigma$, if $f \in \langle\Sigma\rangle$. Given two sets of functions Σ_1 and Σ_2, we say Σ_1 is expressible by Σ_2, in symbols $\Sigma_1 \leq_e \Sigma_2$, if $\forall f \in \Sigma_1\colon f \leq_e \Sigma_2$.

[1] For $n = 1, 2, 3$ we say *unary*, *binary* and *ternary*. Furthermore, we overload the $ar(\cdot)$ for relations.

Definition 2.1.5 (Fictitious Argument)**.** We say a function f contains *fictitious arguments* if there is a function g with $p = ar(g) < ar(f) = q$ and p not necessarily distinct q-ary functions $h_i \in Pr$ such that $f = g(h_1, \ldots, h_p)$.

Definition 2.1.6 (Preservation, Invariants, Polymorphisms)**.** Let f be a function with $ar(f) = p$ and $R \subseteq A^q$ a relation. We say f *preserves* R (or f is a *polymorphism* of R) if for every selection of p not necessarily distinct tuples $x_i \in R$ we have

$$\Big(f\big((x_1)_1, \ldots, (x_p)_1\big), \ldots, f\Big((x_1)_q, \ldots, (x_p)_q\Big)\Big) \in R$$

Alternatively we say R is an *invariant* of f. The set of all functions that preserve a given relation R is denoted as $Pol(R)$. In a similar fashion, we denote the set of all invariant of a given function f as $Inv(f)$.

Definition 2.1.7 (Graph of a function)**.** Given a q-ary f function we can construct a $q+1$-ary relation $f^\bullet$, the *graph* of f, as follows:

$$f^\bullet = \{(x_1, \ldots, x_q, f(\vec{x})) \mid \vec{x} \in A^q\}$$

We overload the $\cdot^\bullet$-symbol for sets of functions $\Sigma \subseteq P_k$ as follows:

$$\Sigma^\bullet = \{f^\bullet \mid f \in \Sigma\}$$

Definition 2.1.8 (Commutation)**.** Two functions f, g over a domain A_k with $ar(f) = p$ and $ar(g) = q$ *commute*, in symbols $f \perp g$, if we have

$$f(g(x_{11}, \ldots, x_{q1}), \ldots, g(x_{1p}, \ldots, x_{qp})) = g(f(x_{11}, \ldots, x_{1p}), \ldots, f(x_{q1}, \ldots, x_{qp}))$$

for all $x_{ij} \in A_k$ (with $1 \leq i \leq q$, $1 \leq j \leq p$).

By interpreting the values x_{ij} as $q \times p$-matrix m (see Figure 2.1), we can rephrase the definition of the commutation property a bit. By applying f to each row, we get a new q-tuple $\vec{y}$. Likewise, we can apply g column-wise to obtain a new p-tuple $\vec{z}$. If we have for all matrices $m \in A^{q \times p}$ and all tuples $\vec{y}$, $\vec{z}$ obtained as described above $f(\vec{z}) = g(\vec{y})$, we have $f \perp g$.

We overload the $\perp$-symbol for sets of functions: $f \perp \Sigma \iff \forall g \in \Sigma\colon f \perp g$.

It is easy to see, that $f \perp g \iff g \perp f \iff f \in Pol(g^\bullet) \iff f^\bullet \in Inv(g)$.

Definition 2.1.9 (Centralizer)**.** Given a set of functions $\Sigma \subseteq P_k$, we refer to the set $\{f \in P_k \mid \forall g \in \Sigma\colon f \perp g\}$ as *centralizer* of Σ, in symbols Σ^*. For any single function $f \in P_k$ we refer to $\{f\}^*$ as centralizer of f, in symbols f^*.

Following equalities hold for any set of functions $\Sigma \subseteq P_k$:

$$\Sigma^* = \{f \in P_k \mid \forall g \in \Sigma\colon f \perp g\} = \bigcap_{g \in \Sigma} g^*$$

$$\begin{array}{cccccccc}
 & & \overset{g}{\frown} & & \overset{g}{\frown} & & \overset{g}{\frown} & \\
f & (& x_{11} & \cdots & x_{1p} &) & = & y_1 \\
 & & \vdots & & \vdots & & & \vdots \\
f & (& x_{q1} & \cdots & x_{qp} &) & = & y_q \\
 & & \underset{\smile}{\|} & & \underset{\smile}{\|} & & & \underset{\smile}{\|} \\
f & (& z_1 & \cdots & z_p &) & = & (*)
\end{array}$$

Figure 2.1: Illustration of the commutation property as computation on matrices.

Definition 2.1.10 (Parametric Expressibility)**.** In [Kuz79] Kuznetsov introduces the notion of parametric expressibility. Given a set of functions Σ, we say an n-ary function f is *parametrically expressible* (or shorter *p-expressible*) by Σ, in symbols $f \leq_p \Sigma$, if $f^\bullet$ is expressible by an equation of the following form

$$\vec{x} \in f^\bullet \iff \exists y_1, \ldots, y_k \colon \bigwedge_i^\ell (A_i = B_i) \quad \text{for all } \vec{x} \in A^{n+1}$$

where $k, \ell \in \mathbb{N}$ and A_i, B_i are well-formed terms using only variables from $\vec{x}$ and $\vec{y}$ and functions from Σ. We call formulae of these form *primitive positive*.

We note that, given a p-ary function f, p q-ary functions g_i and their composition $h = f(g_1(x_1, \ldots, x_q), \ldots, g_p(x_1, \ldots, x_q))$ we have $h \leq_p \{f, g_1, \ldots, g_p\}$, since

$$\vec{x} \in f^\bullet \iff \exists y_1, \ldots, y_p \colon \bigwedge_i^p (g_i(x_1, \ldots, x_q) = y_i) \wedge f(y_1, \ldots, y_p) = x_{q+1} \quad \text{for all } \vec{x} \in A^{n+1}.$$

For each function f and each set of functions Σ we therefore have: $f \leq_e \Sigma \implies f \leq_p \Sigma$.

A set of functions that is closed under parametric expressibility is called *primitive positive clone*[2]. For every primitive positive clone C there is a set of functions Σ with $\Sigma^{**} = C$, we therefore call C *generated* by Σ [Kuz79].

2.2 Exploring Clones

From [BW87] we know that for any finite k there are only finitely many primitive positive clones over P_k. Kuznetsov is reported to have shown that for P_2 there are 25 primitive

[2]Other names for these construct are *parametric clone* or, due to the property mentioned next, *centralizer clone*.

positive clones (his result has been reproven in [Her08]). For P_3 Danil'chenko proved in her dissertation [Dan79] that there are 2986 primitive positive clones which can be obtained by intersecting primitive positive clones generated by a set of 197 base functions.

Furthermore, it is claimed in [BW87] that for each primitive positive clone over A_k there is a finite base containing only functions of arity at most k^k. It is, however, conjectured in the same paper that for $k \geq 3$ functions of arity k could suffice to express all primitive positive clones over A_k.

Given our knowledge of the primitive positive clones over P_2 and P_3, the aim of this thesis is to provide a fast program to check commutation of functions in P_4. To limit the scope and allow us to apply some optimizations, we will only consider functions of arity 4 or less.

The results above show us that it is possible in principle to generate all centralizers for functions up to a certain arity and thereby generating all primitive positive clones. In reality the search space is unfortunately too huge, as, assuming the conjecture from above holds, we have to check k^{k^k} functions *with each other*. For $k = 4$ these are 2^{1023} checks[3] – an enormous number. Without any significant breakthroughs in theory this number will stay prohibitively large. Nevertheless we will now provide some ways to prune our search space.

Kuznetsov [Kuz79] provided a useful criterion which links parametric expressibility and centralizers[4]:

Theorem 1 (Kuznetsov Criterion)**.** Given any function $f \in P_k$ and any set of functions $\Sigma \subseteq P_k$ we have $f \leq_e \Sigma \iff f^* \supseteq \Sigma^*$.

We illustrate this theorem with some examples:

Example 1. For each set of functions $\Sigma \subseteq P_k$ we have $id^* \supseteq \Sigma^*$. On the one side it is easy to see that a commutation test between id and any function p-ary function has to succeed, since obviously $f(id(x_1), \ldots, id(x_p)) = id(f(x_1, \ldots, x_p))$ holds for any x_i.

On the other hand $id^\bullet$ is trivially definable as primitive positive formula using no function symbols at all.

Since id commutes with any function, we don't have to check it.

Example 2. For each function f that contains fictitious arguments and can be reduced to a function g of lower arity we have $f \leq_p g$ and $g \leq_p f$, which means that $f^* = g^*$.

Adding fictitious arguments and setting fictitious arguments equal to other arguments is possible in primitive positive formulae. When interpreting the commutation property as

[3] Since commutation is commutative we only have to consider $f \perp g$ or $g \perp f$, thereby saving half of the checks.

[4] A modern proof is given in [Beh14], which is available at request.

checking matrices, it becomes immediately clear that any function commuting with f also commutes with g and vice versa.

This means that we do not have to check any function with fictitious arguments, as long as we have a "reduced" function (like here g) in our function set.

We note in passing, combining our two examples that, since projections are in principle just instances of the identity function with fictitious arguments, we do not have to check any projections, since they commute with any function.

Example 3. Given a p-ary function f and a permutation σ of the set $\{1,\ldots,p\}$. For any function g, for which $g(x_1,\ldots,x_p) = f\Big(x_{\sigma(1)},\ldots,x_{\sigma(p)}\Big)$ holds, we have $f \leq_p g$ and $g \leq_p f$, which means $f^* = g^*$.

Permuting the variables is possible with primitive positive formulae and, again by interpreting the commutation check as computation on matrices, it is obvious that any function that commutes with f also commutes with g.

We therefore only have to consider one function f in our function set that represents all functions that can be built from f by permuting the input arguments.

Example 4. The trick presented this example does not allow us to prune the search space upfront, but to check commutation in a smart order to avoid certain checks. By the Kuznetsov criterion we know that $f \leq_p \Sigma \iff f^* \supseteq \Sigma^*$. If we now know $f \leq_p \Sigma$ and find a function h with $h \perp \Sigma$, we automatically know that $h \perp f$.

Unfortunately, in general we do not know which functions are p-expressible by which set (otherwise we would not have to examine clones at all). We can, however, starting with a certain base of functions, generate functions and remember which functions were used in the creation these functions.

Given a function set Σ, let C^n_Σ denote the set functions that can be formed from functions in Σ with than n or less compositions, e.g.

$$C^2_{\{f\}} = \{f(x,y), f(f(x,y),z), f(x,f(y,z)), f(f(w,x),f(y,z))\}$$

where $ar(f) = 2$.

Since we have $C^i_\Sigma \leq_p \left(C^{i+1}_\Sigma \setminus C^i_\Sigma\right)$, we can, given a base set Σ, check the functions in the order implied by i (i.e. checking the functions contained in C^i_Σ before the functions contained in $\left(C^{i+1}_\Sigma \setminus C^i_\Sigma\right)$). We note, however, that since in general we can only lose expressive power by function composition, $\left(C^{i+1}_\Sigma \setminus C^i_\Sigma\right)$ may very well be empty or at least so small that the generation is not worthwhile after a certain point. This is only an exemplary way of generating functions from a base set. Depending on the way we generate them, we may combine this approach with other pruning techniques.

Given that we already know the base functions of P_2 and P_3, we can apply some of this knowledge to deal with functions over P_4.

Definition 2.2.1 (Extension)**.** Given a function $f \in P_k$. We call a function g the *extension* of f to P_{k+1} if

$$g(x_1, \ldots, x_p) = \begin{cases} f(x_1, \ldots, x_p) & \text{if } x_i < k \text{ for all } x_i \\ k & \text{else} \end{cases}$$

Theorem 2. Given two function f, g and their extensions f', g' we always have

$$f \perp g \iff g' \perp f'.$$

Proof: Let $f, g \in P_k$, $ar(f) = p$ and $ar(g) = q$. Assume $f \perp g$, then we only have to check matrices containing k to prove $f' \perp g'$. As soon as one element in thematrix equals k the result row as well as the result column contain k. Therefore the result of applying f' and g' to the results will both equal k.

Assume $f' \perp g'$, then there exists no matrix witnessing the non-commutation of f' and g', and therefore, since $A_k^{q \times p} \subset A_{k+1}^{q \times p}$, no matrix witnessing the non-commutation of g and f. □

This knowledge does not help us much with reducing the number of commutation checks, since there are only few functions over P_3 compared to P_4 for any given arity. Nevertheless, the lemma proved above comes in handy in other situations: we can check if our implementation works for functions over P_4 by encoding functions from P_3 as extensions to P_4 and then recalculating already known results.

Another way to group our function into classes, similar to the classes generated by permutations of the input parameters in Example 3, are so called dualities – permutations of the input domain. We denote the set of all permutations of A_k as S_k.

Definition 2.2.2 (Dualities)**.** Given a function $f \in P_k$ and a permutation $\sigma \in S_k$, we call g the σ-dual of f if

$$g(x_1, \ldots, x_p) = \sigma^{-1}(f(\sigma(x_1), \ldots, \sigma(x_p))).$$

We denote the σ-dual for a function f as f^σ.

Theorem 3. For any two functions $f, g \in P_k$ and any permutation $\sigma \in S_k$ we have

$$f \perp g \iff f^\sigma \perp g^\sigma.$$

Proof. Let $f, g \in P_k$, $ar(f) = p$, $ar(g) = q$ and $\sigma \in S_k$. Then

$$
\begin{aligned}
& f^\sigma \perp g^\sigma \\
& \quad \Longleftrightarrow \\
& \qquad f^\sigma\left(g^\sigma\left(x_{11},\ldots,x_{q1}\right),\ldots,g^\sigma\left(x_{1p},\ldots,x_{qp}\right)\right) \qquad = \\
& \qquad g^\sigma\left(f^\sigma\left(x_{11},\ldots,x_{1p}\right),\ldots,f^\sigma\left(x_{q1},\ldots,x_{qp}\right)\right) \\
& \quad \Longleftrightarrow \sigma^{-1}\Big(f\Big(\sigma\Big(\sigma^{-1}\left(g\left(\sigma\left(x_{11}\right),\ldots,\sigma\left(x_{q1}\right)\right)\right)\Big),\ldots,\sigma\Big(\sigma^{-1}\left(g\left(\sigma\left(x_{1p}\right),\ldots,\sigma\left(x_{qp}\right)\right)\right)\Big)\Big) \qquad = \\
& \qquad \sigma^{-1}\Big(g\Big(\sigma\Big(\sigma^{-1}\left(f\left(\sigma\left(x_{11}\right),\ldots,\sigma\left(x_{1p}\right)\right)\right)\Big),\ldots,\sigma\Big(\sigma^{-1}\left(f\left(\sigma\left(x_{q1}\right),\ldots,\sigma\left(x_{qp}\right)\right)\right)\Big)\Big) \\
& \quad \Longleftrightarrow f\left(g\left(\sigma\left(x_{11}\right),\ldots,\sigma\left(x_{q1}\right)\right)\right),\ldots,g\left(\sigma\left(x_{1p}\right),\ldots,\sigma\left(x_{qp}\right)\right) \qquad = \\
& \qquad g\left(f\left(\sigma\left(x_{11}\right),\ldots,\sigma\left(x_{1p}\right)\right)\right),\ldots,f\left(\sigma\left(x_{q1}\right),\ldots,\sigma\left(x_{qp}\right)\right)
\end{aligned}
$$

and $f \perp g$

$$
\begin{aligned}
& \quad \Longleftrightarrow \\
& \qquad f\left(g\left(x_{11},\ldots,x_{q1}\right)\right),\ldots,g\left(x_{1p},\ldots,x_{qp}\right) \qquad = \\
& \qquad g\left(f\left(x_{11},\ldots,x_{1p}\right)\right),\ldots,f\left(x_{q1},\ldots,x_{qp}\right)
\end{aligned}
$$

for all $x_{ij} \in A_k$ (with $1 \leq i \leq q$, $1 \leq j \leq p$).

Since σ is a permutation and both formulae have to hold for all x_{ij}, they have to hold for the same values. Thus they are equivalent. □

Corollary 1. For any two functions $f, g \in P_k$ and any permutations $\sigma, \rho \in S_k$ we have

$$f \perp g^\sigma \iff f^{\sigma^{-1}} \perp g$$

or more generally

$$f \perp g^\sigma \iff f^\rho \perp \left(g^\sigma\right)^\rho.$$

From Corollary 1 we derive the following strategy: Let $f, g \in P_k$, $\Sigma_f = \{f^\sigma \mid \sigma \in S_k\}$ and $\Sigma_g = \{g^\sigma \mid \sigma \in S_k\}$. W.l.o.g. assume that $m = |\Sigma_f| \geq |\Sigma_g| = n$. Then we check for commutation between f and all functions in Σ_g, and generate all other commutation results by applying all permutations in S_k to the obtained check result. Thus we reduce the number of commutation checks from $m \cdot n$ to n.

CHAPTER 3

Architecture

We implemented all our approaches in C++. This has several reasons:

- All low-level interfaces that we used are accessible via C or C++.
- As a compiled language it has an inherent performance bonus.
- There are several mature compilers available.
- It provides us with many compile-time abstractions which do not introduce any run time overhead.

To allow us to experiment with different aspects of our implementation, we implemented our program using *policy-based-design*.

3.1 Policy-based design

One of the central patterns used in our prototype is *policy-based design*. Policy-based design is a pattern that allows us to move certain behavioral aspects of a class (the so-called *host class*) to other classes (the so-called *policies*). The pattern was first popularized by Alexandrescu (for an in-depth description refer to [Ale01]) and is closely tied to the C++-scene, although theoretically applicable in different programming languages. It is similar to the strategy pattern [VHJG95], with the difference that the choice of behavior happens at compile time (and not, as with the strategy pattern, at run time).

To illustrate the pattern let us consider a toy problem. The problem at hand is outputting a greeting to the world. One behavioral aspect we could consider is the language in which this greeting is given. Our host class, the greeter, therefore has to have a policy slot for a language policy – this is realized via a template parameter. The greeter and two exemplary language policies are given in Figure 3.1.

```
#include <iostream>
#include <string>

using namespace std;

struct english_language_policy {
  static string get_greeting() {
    return "Hello, world!";
  }
};

struct russian_language_policy {
  static string get_greeting() {
    return "Привет, мир!";
  }
};

template<class LanguagePolicy> struct greeter {
  static void greet() {
    cout << LanguagePolicy::get_greeting() << endl;
  }
};

int main() {
  greeter<english_language_policy>::greet();
  greeter<russian_language_policy>::greet();
}
```

Figure 3.1: Simple example for policy based design.

An orthogonal aspect to the language is the output stream to which the greeting is outputted. If the need arises to consider different output streams, we could extend our existing greeter with a second policy slot as shown in Figure 3.2.

Further aspects could be considered (e.g. output formats, such as LaTeX, HTML etc.) or existing aspects could be extended (by introducing more languages or allowing different output mechanisms). The crucial point is that, while there are exponentially[1] many ways to instantiate host class, the implementation effort of the host class is roughly linear, given an orthogonal factorization of the behavioral aspects. Since the different policies do not directly interact with each other, their implementation effort may be considered as "constant".

[1] in the number of policy slots

```
#include <iostream>
#include <string>

using namespace std;

/* implementation of the language policies omitted */

struct stdout_output_policy {
  static void output(string s) {
    cout << s << endl;
  }
};

struct stderr_output_policy {
  static void output(string s) {
    cerr << s << endl;
  }
};

template<class LanguagePolicy, class OutputPolicy> struct greeter {
  static void greet() {
    OutputPolicy::output(LanguagePolicy::get_greeting());
  }
};

int main() {
  greeter<english_language_policy, stdout_output_policy>::greet();
  greeter<russian_language_policy, stderr_output_policy>::greet();
}
```

Figure 3.2: Extended example for policy based design.

Since all information about the specific instantiation of policy slots is already known at compile time, policy-based design belongs to the family of compile time abstractions. This allows the compiler to apply more optimizations than with patterns that facilitate run time abstractions – mostly inlining of policy methods which reduces call overhead and may lead to further optimization possibilities.

For our prototype the advantages of policy-based design lie in several related points:

- Since we are interested in the run time of different implementations for certain aspects, we can focus on the implementation of a certain aspect while temporarily disregarding others.

- We can easily study the interaction of different policies (which is especially interesting when comparing the output of different compilers, for results see section 4.1.6).
- For the final implementation we can just take the policy combination which proved to be the most efficient or are especially suited for a certain use case.

Policy-based design has of course also disadvantages:

- Since it relies on templates, the error messages produced (even by modern compilers) are notoriously difficult to understand.
- The interactions of policies (on type level as well as in their behavior) may be subtle and require rigorous and extensive testing.
- Since every instantiation of a policy requires the compiler to generate code, the compilation process may become uncomfortably slow even for a modest code base – in the worst case the compiler may even crash.
- In some cases providing a generic interface within the host classes obscures the relation of data in a way that confuses the compiler and prevents certain optimizations. We therefore had to implement some host classes multiple times in order to gain optimal performance which of course undoes some of the benefits of policy based design (see subsection 5.1.3).

The different behavioral aspects we consider are matrix generation (see subsection 4.1.2), function application (see subsections 4.1.1, 4.1.3 and 4.1.4) and handling of the results (see subsection 4.1.5).

The different synchronization strategies in subsection 4.2.1 are also implemented as policies.

CHAPTER 4

Applying Parallelism

In [Fly72] Flynn defines a well-known two-dimensional taxonomy of computer systems. The two dimensions are the *data*-dimension and the *instruction*-dimension which both may assume the values *multiple* and *single*. In this way we get four different categories:

- Single Instruction, Single Data (SISD): This is the conventional computation pattern. In every step one particular instruction is applied to one particular datum.
- Single Instruction, Multiple Data (SIMD): Here one instruction is uniformly applied to multiple data. Most current processors support SIMD to a certain degree (see for example [Int07]). An example for SIMD would be a componentwise vector addition: a single instruction (addition) is performed on multiple data (the components).
- Multiple Instruction, Single Data (MISD): This mainly theoretical [FR96] category describes multiple functional units which independently operate on a single datum, forwarding their results from one unit to the next.
- Multiple Instruction, Multiple Data (MIMD): Here several processors perform heterogeneous operations on multiple, possibly heterogeneous data. This is the most conventional form of parallel computation and is widely supported by current processors.

As we have seen, modern processors support different modes of parallelism. As suggested in [RJ10], we will facilitate them in a hierarchical manner. On the lowest level we will try to get the best possible speedup by implementing our primitive operation (function application on matrices) with SIMD instructions, on higher levels we will distribute the workload over different processors.

4.1 The SIMD-Implementation

Our prototype will use the *Intel © Streaming SIMD Extension* (SSE). We chose this technology because it is readily available on in many processors and it provides an easy-to-use interface to C++ [Int07].

We will present the implementation for functions over A_4, since it only differs from the A_3-implementation in details. We will not consider functions over A_2 explicitly. This has two reasons: on the one hand, the functions over A_2 are already completely classified (not only do we know all primitive positive clones, but also all clones in general [Her08]), on the other hand, checking all functions (of arity ≤ 3) is only a matter of seconds, even when encoding them as functions over A_3 or A_4.

4.1.1 Evaluating Arity 2 Functions

The biggest data type that SSE provides are 128-bit registers. These registers can be interpreted as fields over integers and floating point numbers of different sizes (e.g. integers with 8, 16, 32 or 64 bits). We will always mention what our current interpretation is and refer to the values over the field a as $a_0, \ldots, a_n$. a_0 will always hold the value that is stored in the least significant bits and when listing the contents of a register the values will be shown left to right from most to least significant.

Unfortunately we cannot use the dense packing which is assumed in Figures 4.3 and 4.2, due to the fact that the smallest usable datatype in SSE are 8-bit (and not 2-bit) integers. By storing each possible result value in such a 8-bit field, we can store functions up to arity 2 in one register.

For the evaluation, we can use the intrinsic `_mm_shuffle_epi8`. This function takes two 128-bit registers a and b: the first one is interpreted as field of 16 8-bit integers, the second one contains 16 8-bit integers, whose respective 4 least significant bits serve as index values. The result is a field c for which holds $c_i = a_{b_i}$[1] (see Figure 4.1 for an illustration).

The way in which `_mm_shuffle_epi8` works already implies a sensible memory layout for functions and matrices: an arity 2 function $f(x, y)$ is stored in the same order as in the function table, starting with $f(0, 0)$ in the least significant byte and ending with $f(3, 3)$ in the most significant (see Figure 4.4).

The layout for matrices (of height at most 2) is also clear: we will use the 4 least significant bits in a 8-bit value b. The bits b_0 and b_1 of will encode the less significant argument (here y) while b_2 and b_3 will encode the more significant one (here x). This way we can put 16 function evaluations in one instruction – this corresponds to 4 2×4- or 8 2×2-matrices.

[1] Strictly speaking $c_i = \begin{cases} a_{b_i} & \text{if } b_i < 127 \\ 0 & \text{else} \end{cases}$ but in our application we will always have $b_i < 16$.

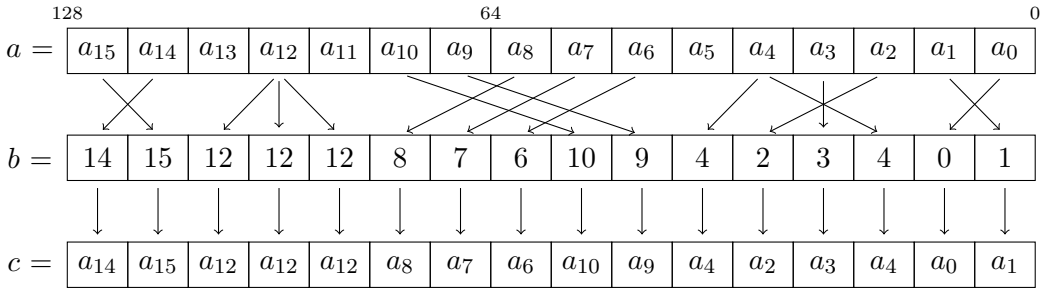

Figure 4.1: Illustration of `_mm_shuffle_epi8`.

size [bit] d / a	2	3	4
1	2	6	8
2	4	18	32
3	8	54	128
4	16	162	512

Figure 4.2: Minimal storage sizes for functions with arity a over d-sized domains. Values given by $\lceil log_2(d) \rceil \cdot d^a$.

size [bit] m / n	1	2	3	4
1	1	2	3	4
2	2	4	6	8
3	3	6	18	12
4	4	8	12	16

(a) domain size 2

size [bit] m / n	1	2	3	4
1	2	4	6	8
2	4	8	12	16
3	6	12	18	24
4	8	16	24	32

(b) domain size 3 and 4

Figure 4.3: Minimal storage sizes for $m \times n$-matrices over d-sized domains. Values given by $\lceil log_2(d) \rceil \cdot mn$.

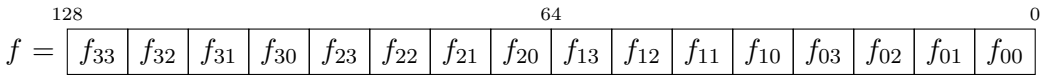

Figure 4.4: Illustration of the memory layout of an arity 2 function. f_{xy} is shorthand for $f(x, y)$.

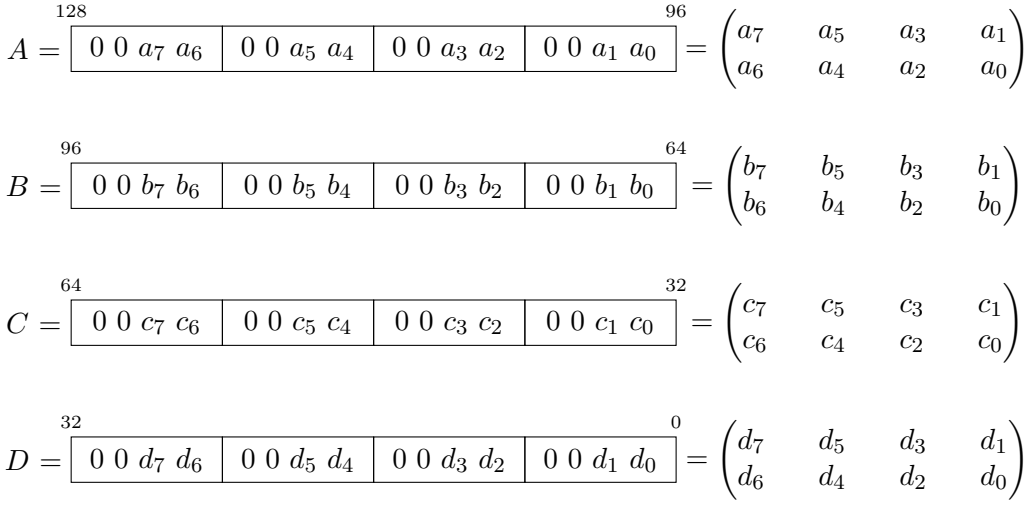

Figure 4.5: Illustration of the memory layout of 4 2×4-matrices. The function is assumed to be applied vertically.

"Blowing Up" a Function

While the layout described above is – to our knowledge – the most efficient representation from a run time perspective, we would like to come closer the storage sizes described in Figure 4.2 when we are not busy evaluating a function (i.e. when it lies in mass storage or in RAM, waiting for evaluation).

We therefore bring our densely packed functions only in the 128-bit format when we are checking them for commutation. We call this step "blowing a function up". The process of blowing up a 32-bit densely packed arity 2 function is illustrated in Figure 4.6.

4.1.2 Generating All Matrices

As described in section 2.1 we have to consider all $m \times n$-matrices when checking $f(x_1, \ldots, x_m)$ and $g(x_1, \ldots, x_n)$ for commutation. How can we implement this generation now and what do we have to consider?

First of all we note that if we want to keep function evaluation as simple as described in subsection 4.1.1, we have to generate not only a matrix M in each step, but also its transpose $M^\top$.

For that reason we have two policy slots for matrix generation in our implementation: one for generating the regular matrices and one for generating the corresponding transposes.

In general we considered two different methods to generate our matrices; incremental generation and the generation from an integer (similar to the "blow up" from above).

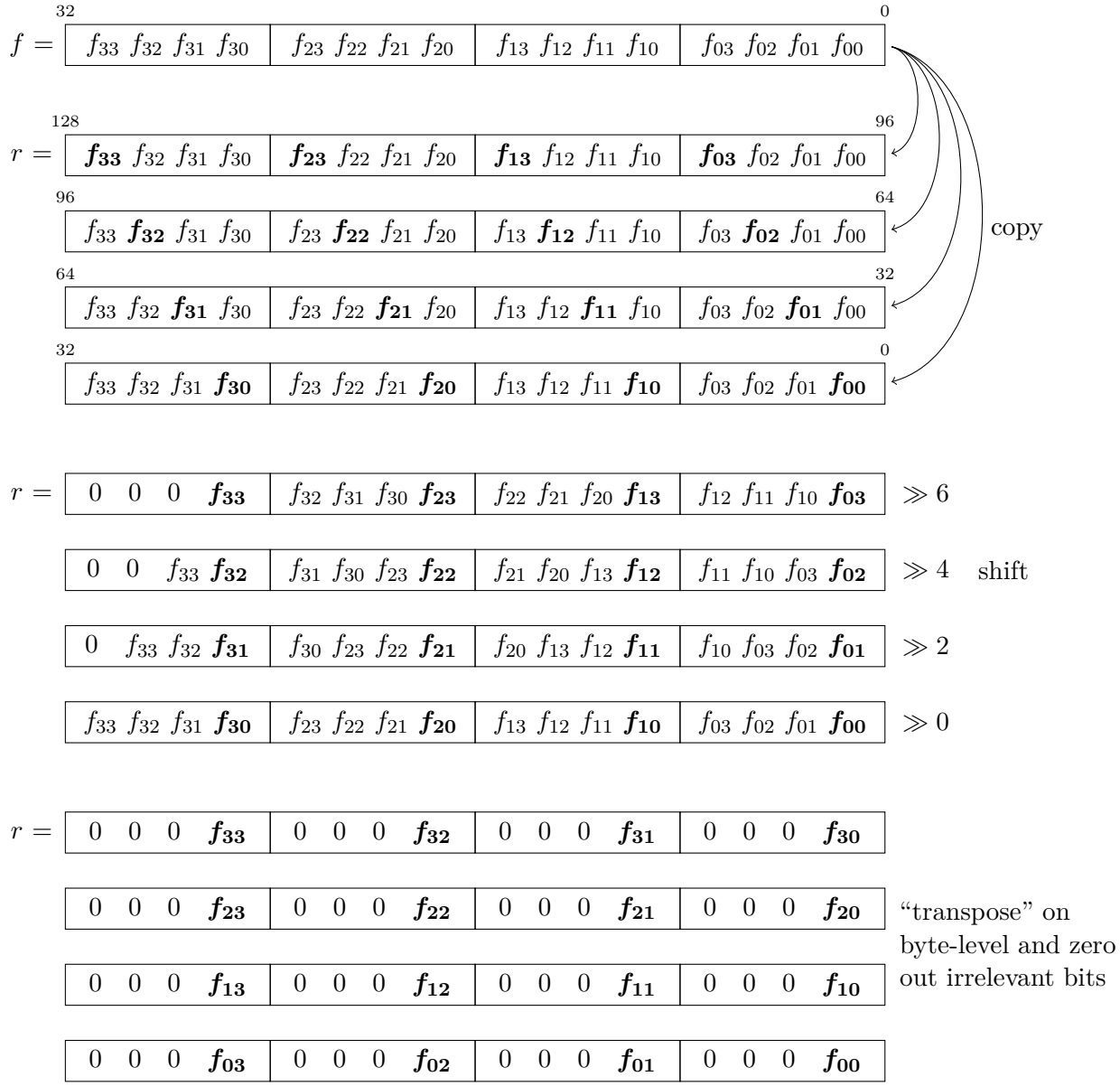

Figure 4.6: Illustration of the “blow up” process for arity 2 functions. Separate shifting of 32-bit values only becomes available in AVX, but can be emulated without much additional effort in SSE4.1 in this special case. The transposition in the last step is implemented via `_mm_shuffle_epi8`.

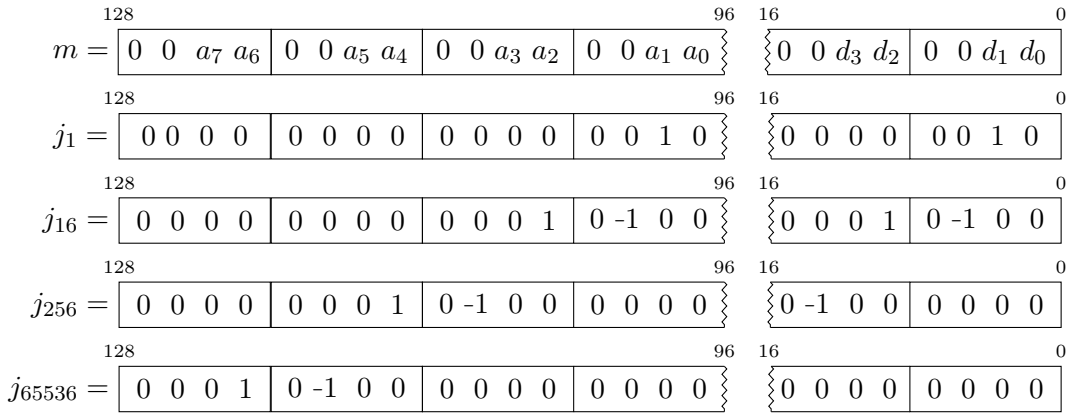

Figure 4.7: Incremental generation for 2×4-matrices. If the counter is divisible by i, j_i is added to m. Since 4 matrices are generated at each step, the counter is increased by 4 in each step (see Figure 4.5 for exact layout).

Incremental Matrix Generation

In this approach we have some registers dedicated to hold our current matrices. In every step we increment the 8-bit values in these registers according to a certain pattern. A counter variable helps us handling overflows (that is values greater 16) while some "overflows" (interpreting the values as 2-bit integers) happen naturally and to our advantage.

In most cases[2] we generate 4 matrices in one step.

This means that if the initial set of matrices is initialized with different values (e.g. $a_0 = 0$, $b_0 = 1$, $c_0 = 2$ and $d_0 = 3$ in Figure 4.5), the remaining operation are homogeneous (i.e. exactly the same operations are applied to each of the 4 32-bit values representing one matrix).

"Blow Up" Matrix Generation

Here we interpret the counter as densely packed matrix similar to the densely packed functions in subsection 4.1.1.

After blowing up the counter to a matrix, it is copied four times in the 32-bit fields of the register holding the matrices and least significant components are modified (e.g. $a_0 + 0$, $b_0 + 1$, $c_0 + 2$ and $d_0 + 3$ in Figure 4.5) to gain 4 different matrices.

[2] In fact the one case in which we considered generating 8 matrices at a time is when checking arity 1 and arity 2 functions with each other, since other pairings would have needed rather complex result handling policies (see subsection 4.1.5).

4.1.3 Evaluating Arity 3 Functions

An arity 3 function fills, densely packed, exactly one 128-bit register. Unfortunately, as we have already seen, this is not suitable for evaluation, therefore we have to handle arities greater than 2 in a more complex manner.

In addition to our existing matrix (from now on "the lower matrix") we will generate a second "higher" one which contains the new most significant argument. The memory layout will be the same used for the lower matrix although the evaluation differs in certain points.

When evaluating an arity 3 function, we will perform 4 uniform steps. Each of these steps is parameterized by a number from 0 to 3 and handles the case that the most significant argument takes this value.

Given the parameter value s and the densely packed function in a 128-bit register r (interpreted as 4 32-bit integers), the evaluation proceeds as follows:

1. broadcast r_s to all fields of r
2. blow up the function as described in subsection 4.1.1
3. evaluate this blown up function with the lower matrix
4. select all indices in the higher matrix which are equal to s
5. at theses indices write the values obtained in step 3 to the final result

We call this the "partial evaluation procedure".

4.1.4 Evaluating Arity 4 Functions

In principle, the evaluation of arity 4 functions is similar to evaluation of arity 3 function with some minor differences. First of all, a densely packed arity 4 function needs 512 bits of storage. We therefore have to use four 128-bit registers $r^0, \ldots, r^3$, where r^p contains the rows of the function table, where the most significant argument equals p.

We then have to call the same procedure as in subsection 4.1.3, while the parameter s takes values from 0 to 15. For $0 \leq s \leq 3$ we have to call the procedure with the function register r^0, for $4 \leq s \leq 7$ with r^1 and so on. Furthermore, in step 1 of the partial evaluations procedure, we may only consider the two least significant bits of s (encoding the second most significant argument).

A "Selective" Evaluation Strategy

Interpreting the counter as densely packed matrix, we can obtain information of the matrix by examining it. With this approach the partial evaluation procedure is not

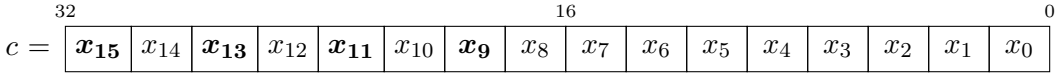

Figure 4.8: Interpreting the loop counter c as densely packed 32-bit matrix. If, for example, none of the highlighted values equals 0, the partial evaluation procedure does not have to be called for $0 \leq s \leq 3$.

necessarily called for each possible parameter but only if the examination of the counter indicates that a corresponding value is in the matrix. Furthermore, the examination happens in two phases: first the most significant argument is checked, then, if necessary, the second most significant.

This approach introduces a fair amount of non-linearity to the control flow of our, until now, rather linear program which can be bad for performance. We therefore implemented two different variants: one as described above and one where only the most significant argument is considered. The results in section 4.1.6 turned out to be interesting.

4.1.5 Handling the Results

After evaluating the functions f and g according to the current matrix and its transpose, we have two registers, r_f and r_g containing at most 16 values from 0 to 3.

These can be handled in different ways. The most primitive way is just building two new matrices m_f and m_g from r_f and r_g and evaluating f according to m_g and g according to m_f. If the new results are not equal, f and g do not commute. If the results are equal, we check the next matrix.

Accumulating the Results

If f and g turn out not to commute soon, the simple method from above is perfectly fine. In the worst case – that is from a run time perspective $f \perp g$ – it is wasteful, because depending on the packing of the matrices, only 4 (respectively 8) functions are evaluated in each step, instead of the maximally possible 16.

If we accumulate the matrices m_f and m_g in a designated register, we only have to evaluate the functions every fourth (respectively second) step, which helps to reduce the run time in the worst case.

A "Selective" Result Handling Strategy

Consider the way our matrices (and their transposes) are generated and evaluated:

$$\begin{array}{llll} f(x_{15} & x_{13} & x_{11} & x_9) = y_3 \\ f(x_{14} & x_{12} & x_{10} & x_8) = y_2 \\ f(x_7 & x_5 & x_3 & x_1) = y_1 \\ f(x_6 & x_4 & x_2 & x_0) = y_0 \end{array} \qquad \begin{array}{llll} g(x_{15} & x_{14} & x_7 & x_6) = z_3 \\ g(x_{13} & x_{12} & x_5 & x_4) = z_2 \\ g(x_{11} & x_{10} & x_3 & x_2) = z_1 \\ g(x_9 & x_8 & x_1 & x_0) = z_0 \end{array}$$

When generating all matrices in standard order (i.e. increasing x_0 and then increasing x_i by one if x_{i-1} would reach 4 while setting x_{i-1} to 0), we can make the following observation: y_0 and z_0 may change every step, z_1 changes at most every 16th, z_2 at most every 256th step and so on.

This allows, in combination with accumulating result handling and selective evaluation, two optimizations. First consider that y_i and z_i are the components of the new matrices used in the final step of the commutation test. Since the constituents of the new upper matrix change seldom compared to those of the lower matrix (at most every 65536 steps for m_f and every 256 step m_g) we can save some performance by only calculating them when needed.

Secondly, since the upper matrix always consists of the slow changing values y_2, y_3, z_2 and z_3 we can further optimize the selective evaluation: according to the value of y_3 and z_3 we remember which of the four registers holding our function will be used for evaluation. Then only exactly one call to the partial evaluation procedure has to be performed, according to the value of y_2, respectively z_2.

4.1.6 Performance

All benchmarks in the following section were performed on a customary notebook with a Intel® Core™ i5-5300U CPU. The code was compiled with g++ 5.3.0 and clang++ 3.7.1 respectively using the highest available optimization level.

As mentioned before, the choice of the benchmark can drastically influence the performance of the tested programs: in most cases non-commutation can be decided after considering only a few matrices – most of our optimizations on the other hand aim to reduce the computational work in the long run, i.e. when checking commuting functions.

We therefore will consider different benchmarks and start with the most difficult case at first, commuting arity 4 functions over A_4. What we see immediately in Figure 4.9 is that the choice of the compiler is a huge factor for our performance: clang already produces fast code without optimizations, while it actually produces worse code than gcc when all optimizations are applied. What is interesting is the fair balance between the amount of executed code and the non-linearity in the control flow one has to consider: the selective evaluation which takes only the most significant parameter into account (denoted by $\text{selective}_{\text{ms}}$ in the figures) seems to hit a sweet spot for gcc but performs badly when compiled with clang.

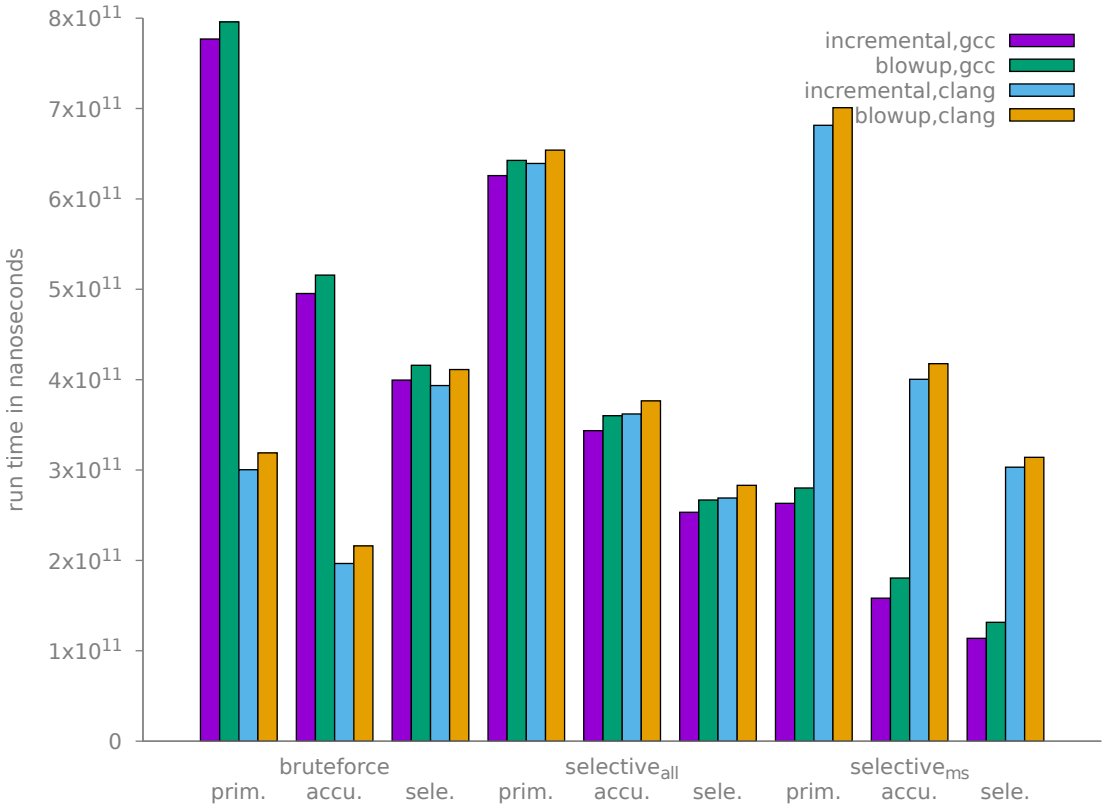

Figure 4.9: Run time of benchmark consisting of all 4 constant functions over A_4, grouped by evaluation and result handling policy.

The selective evaluation that takes all contents of the upper matrix into account (denoted by $\text{selective}_{\text{all}}$ in the figures) produces much slower results for gcc and slightly faster results for clang.

The result handling strategies work as expected (with one exception: brute force evaluation with selective result handling, compiled with clang).

Let us now consider a set of 10,000 randomly generated non-commuting arity 4 functions over A_4. As we can see in figure 4.10, accumulating (and to a lesser degree also selective) result handling affects the run time negatively – the overhead needed to accumulate matrices is wasted on functions which can be decided to be non-commuting using few matrices, which is the case for most randomly generated functions. What is interesting is that the differences between gcc and clang have become less pronounced for this benchmark.

Since matrix generation does not seem to influence the performance significantly, we only implemented the blowup policy for arity 4 functions over A_4. Therefore, the following

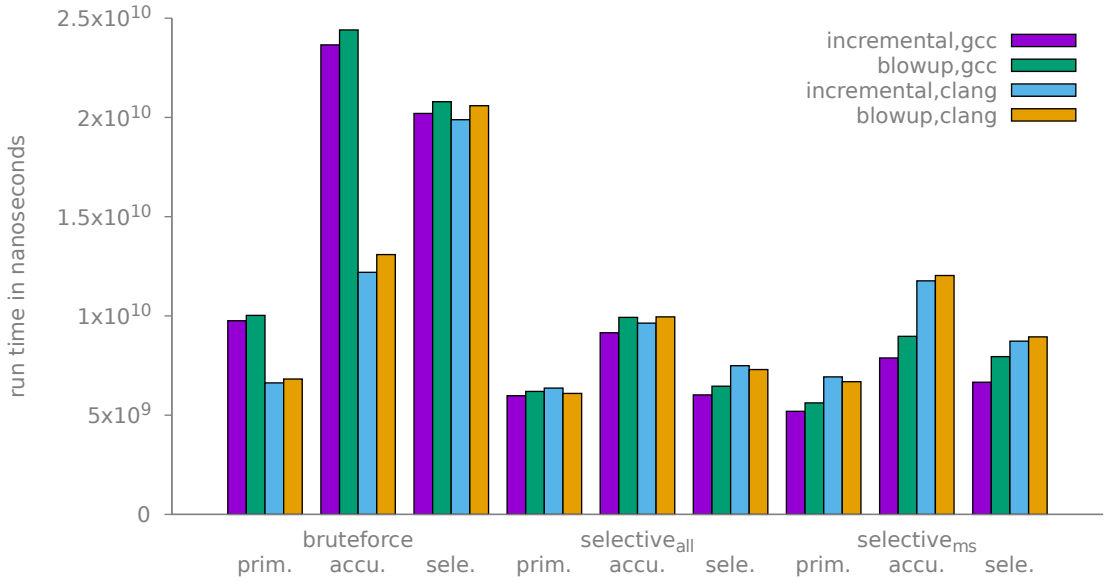

Figure 4.10: Run time of benchmark consisting of 10,000 randomly generated non-commuting functions over A_4.

benchmarks will only contain results for incremental matrix generation.

As we have seen now, the properties of the examined functions greatly influence the run time behavior of our programs. What is now the optimal strategy for real world function sets? One rather small but, in a sense, complete benchmark which we can easily test is the set $\{f \in P_3 \mid ar(f) \leq 2\}$ which has only $3^{3^1} + 3^{3^2} = 19710$ elements.

Because of the small arities of the functions under consideration, we cannot apply the advanced selective evaluation and result handling policies, but a new chance for optimization appears: dense matrix packing, i.e. generating 8 instead of 4 matrices per iteration. As we can see in figure 4.11, the overall effect of dense matrix packing is positive but modest. For commuting functions, however, the expected factor 2 speed up occurs. Another interesting observation is that even when considering only positive instances, accumulating the results has no positive effect for matrices of this size.

4.2 Parallelization on a Higher Level

Having done our best to optimize our implementation on one processor, we now move up one level and try to distribute our workload efficiently among processors. We will first implement the parallelization with the well-established OpenMP-Framework and then try to improve on this results with a custom implementation based on pthreads.

As the single commutation test is rather fast in most cases, we will focus on distributing

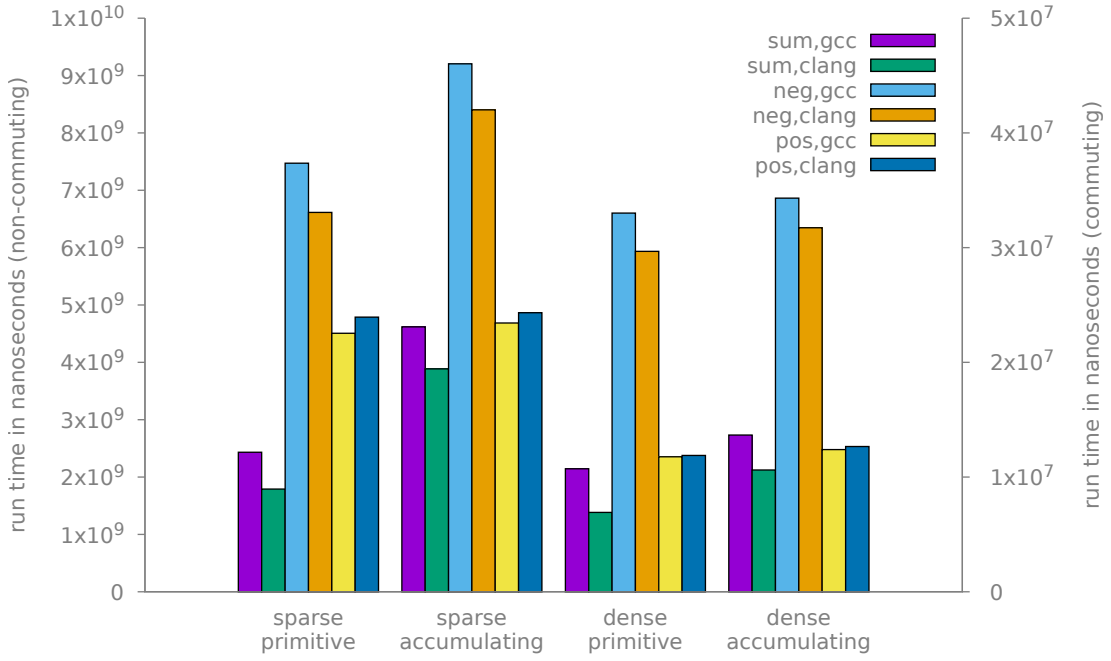

Figure 4.11: Run time of benchmark consisting of all functions of arity 2 or lower over A_3 (encoded as functions over A_4). Please note the different scales for positive (commuting) and negative (non-commuting) instances. Since the separation of commuting and non-commuting instances introduced considerable overhead, we also give the run time of the non instrumented binary in the first two columns of each cluster. These columns use the left scale.

the commutation tests among processors instead of further parallelizing the single commutation test.

4.2.1 OpenMP

OpenMP [Ope13] is a collection of compiler directives and library functions for C, C++ and Fortran. It provides an easy-to-use interface for shared-memory parallel programming, but relies, due to its nature as language extension, on compiler support.

The simplest form of parallelization is demonstrated in figure 4.12. Let us assume we want to check the contents of `vec1` for commutation with the contents of `vec2`. `A1`, `A2` and `D` are global constants that describe the arities of the checked functions and the domain size respectively. We also assume that if `A1` equals `A2`, `vec1` equals `vec2`, which allows the optimization in the initialization of `j`.

The code now is the same as the single threaded version with the exception of two compiler directives. The first one directs the compiler to parallelize the first loop. All

```
#pragma omp parallel for schedule(runtime)
for (uint64_t i = 0; i < vec1.size(); ++i) {
    for (uint64_t j = A1 != A2 ? 0 : i; j < vec2.size(); ++j) {
        if (solver<D, A1, A2>::commutes(vec1[i], vec2[j])) {
            #pragma omp critical
            {
                matches.add(i, j, A1, A2);
            }
        }
    }
}

return matches;
```

Figure 4.12: Simple parallelization with OpenMP.

variables declared in a parallel block (`i` and `j` in this case) become private to a thread. To avoid race conditions and other errors when we access shared variables (such as `matches` in this case), we have to be careful. One way to handle concurrent access to shared variables, such as in the innermost block in figure 4.12, are so-called critical sections.

Only one thread at a time can execute the commands in a critical section, which means that if there are too many of these sections, we loose the advantages of multithreading [SL08]. In our case the critical section is rarely entered, but nevertheless we examine two approaches two avoid it completely.

Using Separate Variables

Consider the code in figure 4.13. The two new OpenMP library functions `omp_get_max_threads` and `omp_get_thread_num` have the expected semantics: the first one returns the maximal number of threads that can be created when entering a parallel section, the second returns an identifier from 0 to the `omp_get_max_threads()` -1. Furthermore we assume `join_matches` to be a function that constructs a new instance of `matches_type` which contains all matches from the two input parameters.

We then avoid using a critical section the following way: for each potential thread we create an instance of `matches_type` which we access using the unique thread number. At the end of the parallel execution we accumulate the results manually.

Using a User-Defined Reduction

Starting with version 4.0, OpenMP allows us to define our own reductions (where earlier versions only offered a fixed set of mathematical operations on basic types). A reduction is an operation which accumulates all values of a thread private variable to a single

```
const size_t max_threads = omp_get_max_threads();
std::vector<matches_type> thread_matches(max_threads);

#pragma omp parallel for schedule(runtime)
for (uint64_t i = 0; i < vec1.size(); ++i) {
    for (uint64_t j = A1 != A2 ? 0 : i; j < vec2.size(); ++j) {
        if (solver<D, A1, A2>::commutes(vec1[i], vec2[j])) {
            thread_matches[omp_get_thread_num()].add(i, j, A1, A2);
        }
    }
}

return std::accumulate(thread_matches.begin(), thread_matches.end(),
                       matches_type(), join_matches);
```

Figure 4.13: Using separate variables to avoid critical sections.

```
#pragma omp declare reduction(match_join : matches_type :\
 omp_out = join_matches(omp_out, omp_in))
matches_type matches;
#pragma omp parallel for schedule(runtime)\
 reduction(match_join : matches)
for (uint64_t i = 0; i < vec1.size(); ++i) {
    for (uint64_t j = A1 != A2 ? 0 : i; j < vec2.size(); ++j) {
        if (solver<D, A1, A2>::commutes(vec1[i], vec2[j])) {
            matches.add(i, j, A1, A2);
        }
    }
}
```

Figure 4.14: Using user-defined reductions to avoid critical sections.

variable at the end of a parallel block (using a reduction on a variable automatically makes it thread private). In principle, we are doing the same thing as in the previous subsection, only with language support. For a demonstration refer to figure 4.14.

The `declare reduction` compiler directive consists of three mandatory parts[3]:

1. an identifier which is used in `reduction`-clauses
2. the type on which the reduction operates

[3]The fourth would allow specifying an initial value for the reduction.

3. an expression with which the reduction is calculated. Here we can make use of the variables `omp_in` and `omp_out` which hold the two values to be reduce. It is assumed that after applying the expression `omp_out` holds the reduced value.

Of the compilers we used only GCC supports user-defined reductions so far.

Other Optimizations

In addition to these approaches, we can try some other ways of tuning our implementation. First of all we can try different schedules. The `schedule` directive (which is applied to loops) can takes two arguments: the *kind* and the *chunk size*. The standard defines three schedule strategies and two additional possible values for the kind parameter:

- `static`: Here *chunk size* iterations are grouped together and assigned to the threads in a static, round robin fashion. Let $c =$ *chunk size* and m be the maximal number of threads, then the assignment of iterations works in the following way: the first thread in thread group is assigned the first c iterations, the second one the second c iterations and so on. If e.g. the first thread is finished with its assigned iterations, it starts with working on the next c iterations starting with $m \cdot c$. This strategy has the lowest overhead and works best if all iterations take more or less the same amount of time.
- `dynamic`: Here every thread is assigned *chunk size* iterations. If it is finished it takes another *chunk size* iterations from an internal work queue. This strategy has more overhead than the `static` strategy but should, in theory, be more suited for tasks where the run times of iterations vary much.
- `guided`: This strategy is similar to the `dynamic` strategy, but the size of a group of iterations assigned to thread is proportional to the number of remaining iterations to perform. This happens to even out run time differences between the iterations. The chunk size parameter is used to determine the minimum number of assigned iterations.
- `auto`: Here the implementation chooses a strategy.
- `runtime`: When this value is specified, we can set the strategy via environment variables or functions in the OpenMP library. We used this parameter value in our implementation and set the strategy via command line switches.

Checking a pair of functions for commutation can, depending on the result of the check, take a different amount of time – for higher arities and domain sizes, the run time of the check may take any time from some nanoseconds to tens of seconds. If now, for example, `vec1[i]` contains a function that commutes with many other functions (e.g. a constant function), the execution of the thread handling `vec1[i]` may take much longer

```
#pragma omp parallel for collapse(2)
for (uint64_t i = 0; i < vec1.size(); ++i) {
    for (uint64_t j = 0; j < vec2.size(); ++j) {
        if (A1 == A2 && j < i) {
            continue;
        }
        if (solver<D, A1, A2>::commutes(vec1[i], vec2[j])) {
            #pragma omp critical
            {
                matches.add(i, j);
            }
        }
    }
}

return matches;
```

Figure 4.15: Illustration of the `collapse` directive (applied to the code in figure 4.12).

than the execution of other threads – it may even run longer than all other threads combined, which leads to a suboptimal utilization of the processor. Therefore, it may be beneficial to "collapse" our nested for loop to one loop (such that every pair `i` and `j` may be examined in its own thread). To do this, we can use the `collapse` directive.

Unfortunately, to be able to apply a `parallel for` directive to a loop, it must be in "canonical form" (see [Ope13], section 2.6). Our inner loop (to which the `parallel for` directive is applied, because of the `collapse` directive), however, is not in canonical form. The problem is the initialization of `j`, which is not loop invariant. We can try to mitigate this by skipping the iterations which do not interest us. For an illustration of the code using collapse refer to figure 4.15. Another possibility would be replacing the nested loop with a non-nested one and calculating the two indices from the loop counter.

4.2.2 Using C++ Threads

Starting with C++11, C++ specifies a multi threaded computation model and a fitting API for harnessing the power of concurrency. On Linux, which was our target operating system, the implementation of this API is based on pthreads [Gro13].

Our implementation has (beside the number of threads of course) only one parameter, the chunk size. Every thread maintains its own matches and the only time the need for synchronization arises is, when a thread requests a new chunk of iterations.

Thus, our implementation emulates the separate strategy with a dynamic scheduler.

4.2.3 Performance

For benchmarking multi threading on a higher level, we have variety of parameters we may consider: different scheduling and synchronization strategies, thread counts and chunk sizes may influence our performance as much as a different inputs. The discussed benchmark here consisted of a set of 500,000 randomly generated arity 2 functions over A_4 which contained commuting functions. The hardware used was a 80-core shared memory node based on the Intel® Xeon® E7-885 processor, that provided 160 virtual threads.

To isolate trends in the such a big amount of data is of course difficult, especially if one tries to make general statements. Nevertheless we shall try to at least gather identify at least some interesting details.

First of all it shall be noted that the choice of the synchronization strategy – that is whether we are using critical sections, separate variables or custom reductions – is largely irrelevant for our application: when the other parameters were appropriately set, the differences between run times were within tenths of seconds.

As for different schedulers: the `guided` scheduler seemed to be a bad match for our application. Not considering the outliers mentioned below, it consistently performed worse than the `dynamic` or `static` schedulers.

When parallelizing only the outer of our two loops, the chunk size does not seem to have big influence on the performance, as the execution of the inner loop takes a long time (when compared to a single commutation check). The scheduling is thus already with chunk size 1 “coarse grained” enough not to cause a significant overhead.

When we are however using the `collapse` directive as described above, the behavior drastically changes. When using a too small chunk size, e.g. 1, the synchronization overhead for the `dynamic` scheduler becomes so large, that in our benchmarks regularly crashed. In the runs that did not crash, we could observe a huge performance penalty. A similar effect, although not as pronounced, can be observed with the `static` scheduler. The performance when using the `guided` scheduler is, for our application, not influenced by the choice of the chunk size. For an illustration, refer to figure 4.16.

We conclude this section with a presentation of the benefit gained through parallelization. First we note that the pthread-based approach ran slightly, but consistently, faster than the non-collapsed OpenMP-based approach which in turn ran slightly, but consistently, faster than a collapsed version with an appropriate chunk size as we see in figure 4.17. Furthermore, we see that our problem scales well over multiple processors and that, as a CPU bound task, using virtual threads does not yield a big performance gain.

4.3 The GPU-Implementation

In graphics contexts often the need arises to apply a certain operation homogeneously and in parallel to huge amounts of input data. To transform or move, for example, a three

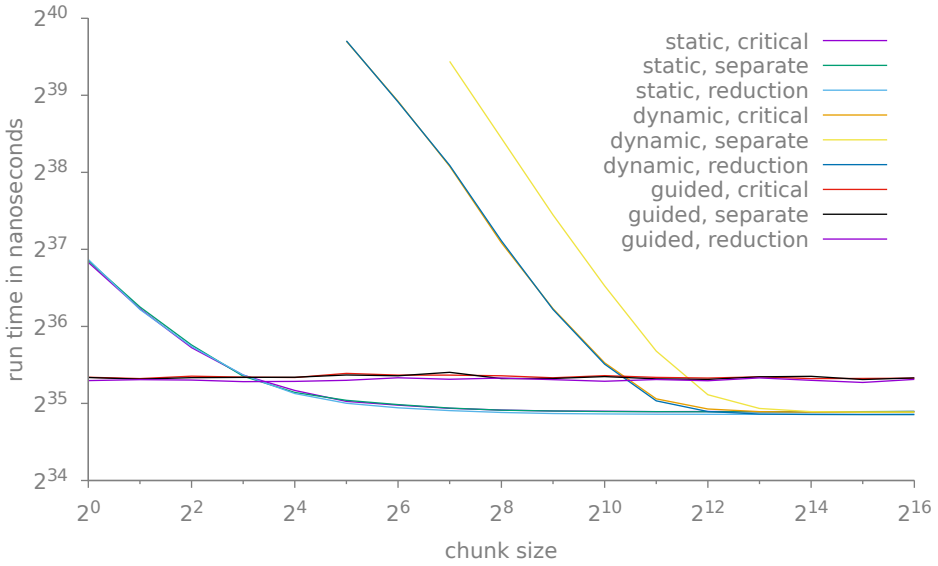

Figure 4.16: Run time of a benchmark relative to the chunk size. The three recognizable groups correspond to the different schedulers.

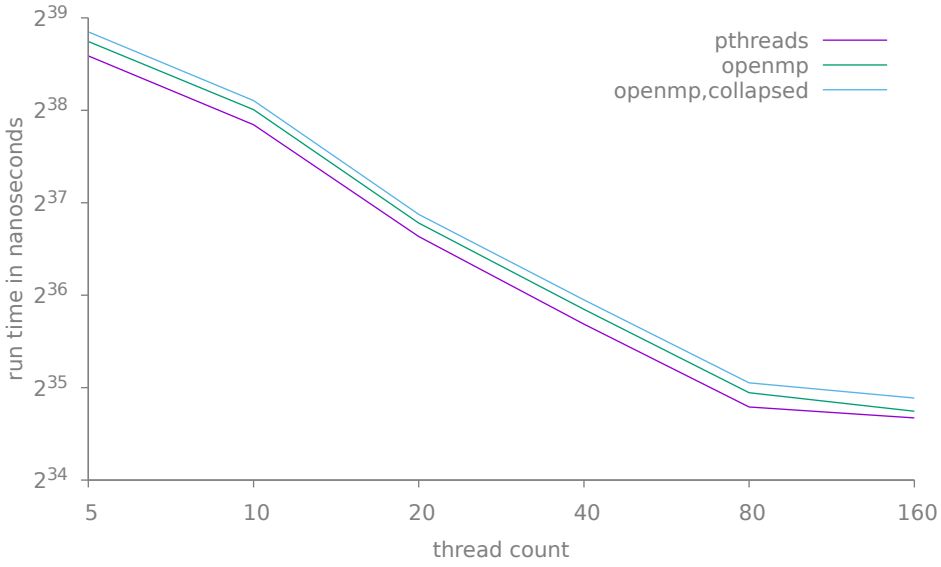

Figure 4.17: Run time of a benchmark relative to the number of threads. The OpenMP variants used for illustration purposes uses the “separate” synchronization strategy and dynamic scheduling. The chunk sizes are 1 for the non-collapsed versions and 65536 for the collapsed one.

dimensional model on screen, each of the vertices describing its surface has to be modified (usually via a matrix multiplication). Another example are so-called "fragment shaders" which apply some visual effect on already rendered images, pixel by pixel. Additionally when used in real-time applications, such as video games, these computations have to be performed at an acceptable rate.

To cope with these demands, starting in the late nineties, special coprocessors – GPUs (graphics processing units) – were developed. These coprocessors evolved from modest chips that offered a rather inflexible fixed-function pipeline to the flexible and powerful parallel computing systems we have nowadays. For a historical overview consider [ND10].

To make the computational power of GPUs available for more general applications (GPGPU, general purpose computing on graphics processing units), different interfaces such as OpenCL [Gro] and NVIDIA's CUDA [Cor] have been introduced.

To keep our program vendor neutral, we first considered an implementation in OpenCL, but since the performance of our prototype was rather disappointing on our available hardware (NVIDIA K20x), we changed our application to use CUDA. The following exposure therefore uses CUDA-specific concepts and terms which, in our experience, can be easily applied in OpenCL contexts.

As with the more complex optimizations in section 4.1 we only consider our "worst case": arity 4 functions over A_4.

4.3.1 A CUDA Primer

CUDA is a minimal extension to the C++ programming language, coupled with a library of functions and a runtime environment.

The extension consists of the possibility to specify and call so-called kernels. A kernel is a procedure which is executed in parallel on the GPU. The data on which a kernel operates must be transferred to the GPU memory upfront via special functions in the CUDA library.

CUDA allows us to organize computation in a two-level hierarchy. On the higher level we have the *grid* of *blocks*. These blocks, which get assigned a unique identifier, cannot communicate with each other and may therefore be executed in parallel and independently from each other. Within these blocks, we have *threads* which get assigned an identifier which is unique within a block. Threads can share memory which is private to a block and have a simple form of synchronization [NBGS08].

When calling a kernel, the programmer may specify the number of blocks within a grid and the number of threads within each block; the latter is also called "block dimension". For convenience, the grid may have up to three dimensions, which leads to a three-dimensional block identifier, but this feature is not used in our implementation.

4.3.2 Checking Functions

As before, we represent our functions by storing them into function tables. This time, instead of densely packed in SSE registers, we store them into byte arrays in the GPU memory. CUDA offers different memory regions which are optimized for different use cases, e.g. memory for textures which usually get accessed in certain patterns, which in turn may be exploited for cache locality.

We chose the so-called "constant memory", which promises fast access in exchange for the ability to write in the memory from a kernel, but in our benchmarks the different memory regions did not influence the performance critically.

Before calling the kernel, we have to decide how many of the 4^{16} matrices we want to check at once. If we generate n matrices at once, we have to allocate a size n result array a, which will contain $a_i = 1$ if the ith matrix of this run is a witness for the non-commutation of the two checked functions, and $a_i = 0$ otherwise.

This way, we need to call the kernel $\lceil \frac{4^{16}}{n} \rceil$ times. The offset mentioned further below equals $\lceil \frac{4^{16}}{n} \rceil \cdot k$ in the kth run.

Within the kernel, the commutation test consists of these parts:

1. generate a matrix from block and thread identifier and offset
2. generate array indices for each row/column
3. look up function values in arrays
4. generate new array indices from results
5. look up new results from theses indices
6. if results are equal write 0 in output array, if the results are unequal write 1 in the output array

Since the code is very short, it is presented in its entirety in figure 4.18.

In its simplicity this code, which is more or less the first version taken from the OpenCL prototype, does not offer many possibilities to optimize. One ostensible way to improve performance would be to simplify the matrix layout from

$$\begin{pmatrix} x_{15} & x_{13} & x_{11} & x_9 \\ x_{14} & x_{12} & x_{10} & x_8 \\ x_7 & x_5 & x_3 & x_1 \\ x_6 & x_4 & x_2 & x_0 \end{pmatrix} \quad \text{to} \quad \begin{pmatrix} x_{15} & x_{11} & x_7 & x_3 \\ x_{14} & x_{10} & x_6 & x_2 \\ x_{13} & x_9 & x_5 & x_1 \\ x_{12} & x_8 & x_4 & x_0 \end{pmatrix}.$$

This would reduce the generation of the first array indices to one shift and one bitwise and, and would not change the number of operations in the second one.

Interestingly, the code with the simpler layout reproducibly has worse performance on our hardware than the initial code.

Other approaches which tried to reduce the number of array index calculations by using shared memory also led to a worse performance than the already presented approach.

4.3.3 Reducing the Result Array

After checking the functions in the previous subsection, we have to check if the array contains an element different from zero. Since bitwise or, which we will use to check for non-zero elements, is an associative operation, we may organize our search as a balanced binary tree that gets evaluated layer for layer in parallel. See figure 4.19 for an illustration of the principle.

Parallel Reduction is a subtle topic, as is demonstrated in [H+07]. The performance depends on hardware details as well on seemingly unimportant aspects of the implementation. Thankfully, newer installations of CUDA already ship with the thrust library which provides efficient implementations of parallel algorithm primitives such as reduction.

Since we are only interested in the two possible results zero or non-zero, we can save some execution time by interpreting the input array not as bytes when reducing, but as 32-bit values. This optimization speeds up the reduction step by more than factor three.

4.3.4 Performance

The main degrees of freedom we checked for were number of threads per block and number of matrices generated per call to the kernel. Especially the second parameter is a subtle point: since small group sizes could lead to an earlier identification of non-commuting function pairings the choice of the benchmark is important.

Our used benchmark consisted of all constant functions over A_4. This means that there were 4 positive commutation tests and 6 which could be decided to be negative after the first call to the kernel.

Another thing to consider is the fact that kernel calls in CUDA are performed asynchronously from the perspective of the CPU. Only when, for example, memory is transferred between host and device memory or synchronization is explicitly requested by the `cudaDeviceSynchronize` function, the CPU and GPU are synchronized. We therefore have to pay attention when we benchmark single kernels.

Let us first consider the reduction operation. Since we rely on the thrust library, which chooses the number of threads, the only degree of freedom is the group size. Here the experiments are consistent with our expectations of logarithmic run time behavior: as seen in figure 4.20 doubling the group size roughly halves the total run time (in a certain interval).

```
const uint64_t f1l = cpow(D,A1);
const uint64_t f2l = cpow(D,A2);

//function that is applied vertically
__constant__ char df1[f1l];
//function that is applied horizontally
__constant__ char df2[f2l];

__global__ void apply_function(char* C, unsigned int offset){
    const unsigned int result_position = blockIdx.x*blockDim.x +
                                         threadIdx.x;
    const unsigned int mat = result_position + offset;

    unsigned int r0 = ((mat >> 12) & 0xF0) | (mat         & 0x0F);
    unsigned int r1 = ((mat >> 16) & 0xF0) | ((mat >> 4)  & 0x0F);
    unsigned int r2 = ((mat >> 20) & 0xF0) | ((mat >> 8)  & 0x0F);
    unsigned int r3 = ((mat >> 24) & 0xF0) | ((mat >> 12) & 0x0F);

    r0 = df1[r0];
    r1 = df1[r1];
    r2 = df1[r2];
    r3 = df1[r3];

    unsigned int ra = r0 | (r1 << 2) | (r2 << 4) | (r3 << 6);
    ra = df2[ra];

    r0 = ((mat >>  6) & 0xC0) | ((mat >>  4) & 0x30) |
         ((mat >>  2) & 0x0C) | (mat         & 0x03);
    r1 = ((mat >>  8) & 0xC0) | ((mat >>  6) & 0x30) |
         ((mat >>  4) & 0x0C) | ((mat >>  2) & 0x03);
    r2 = ((mat >> 22) & 0xC0) | ((mat >> 20) & 0x30) |
         ((mat >> 18) & 0x0C) | ((mat >> 16) & 0x03);
    r3 = ((mat >> 24) & 0xC0) | ((mat >> 22) & 0x30) |
         ((mat >> 20) & 0x0C) | ((mat >> 18) & 0x03);

    r0 = df2[r0];
    r1 = df2[r1];
    r2 = df2[r2];
    r3 = df2[r3];

    unsigned int rb = r0 | (r1 << 2) | (r2 << 4) | (r3 << 6);
    rb = df1[rb];

    C[result_position]= (ra != rb);
}
```


Figure 4.18: CUDA kernel which is used to apply functions on the GPU.

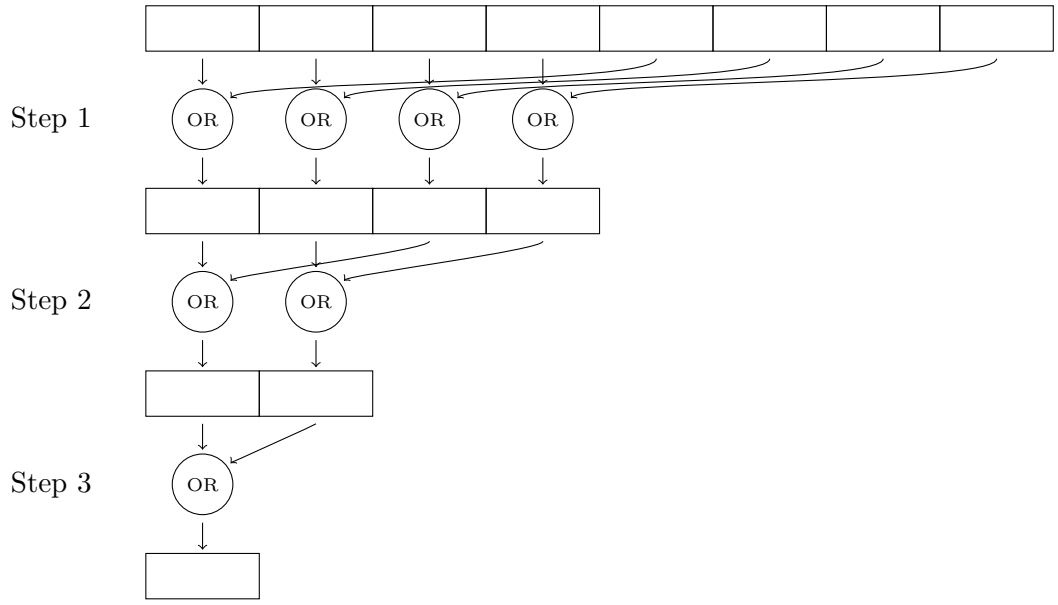

Figure 4.19: Illustration of the parallel reduction algorithm. The operations in each step may be executed in parallel. This tree structure accesses the memory in an efficient way but requires commutativity in addition to associativity.

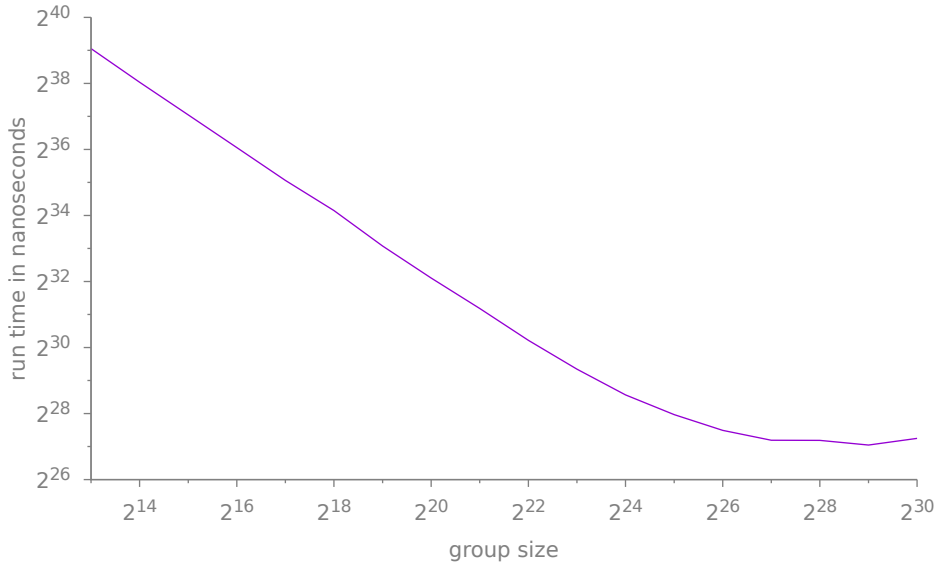

Figure 4.20: Run time of the reduction step relative to the group size.

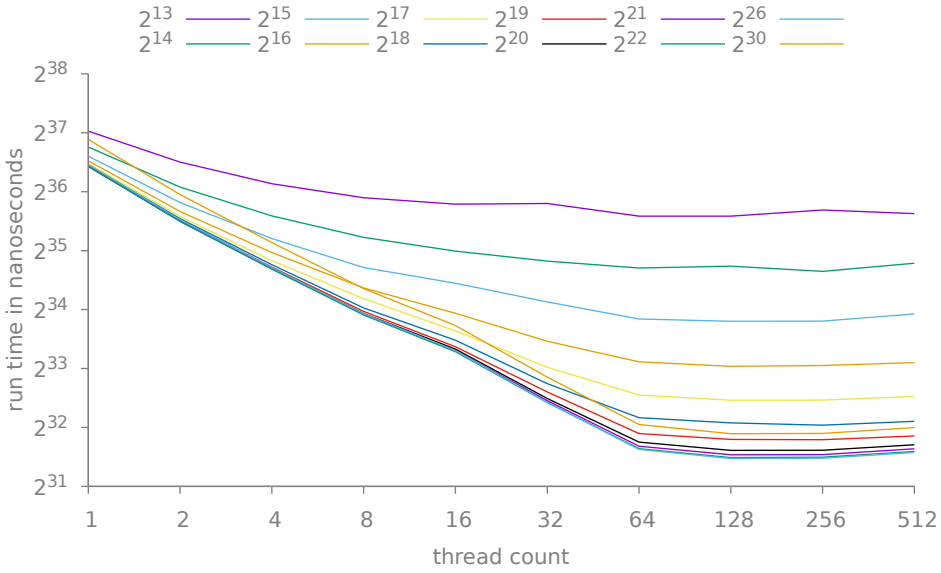

Figure 4.21: Run time of the apply step relative to the number of threads for different group sizes. Increasing the group size stops yielding a benefit after a certain point.

Let us then consider the function application step. Here the algorithm has in principle linear performance run time, however, as we see in figures 4.21 and 4.22 increasing the thread count and group size has nevertheless a beneficial effect.

Increasing the number of threads stops yielding a benefit at 64 threads and increasing the group size over a certain limit of about 2^{26} has (in our bench mark) even a negative effect.

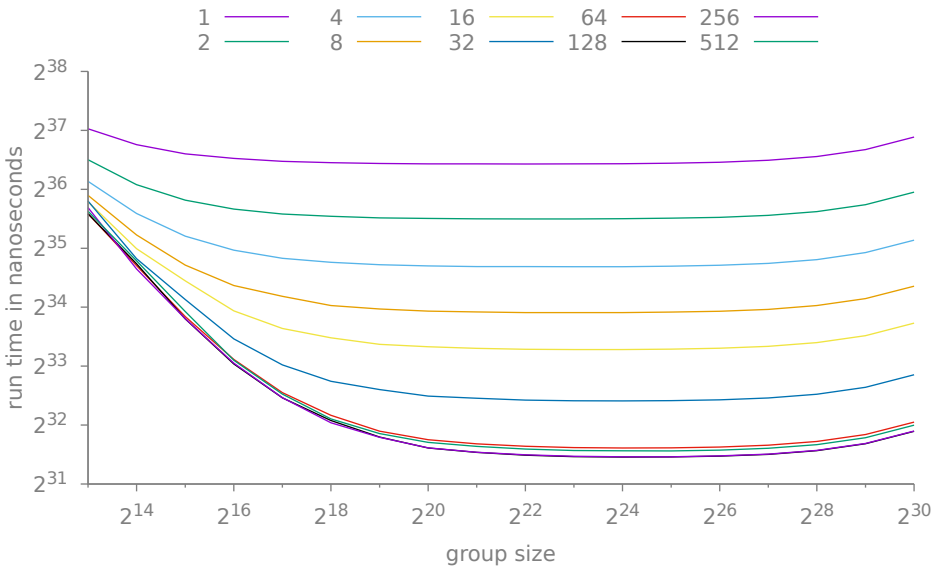

Figure 4.22: Run time of the apply step relative to the group size.

CHAPTER 5

Conclusions

After reducing the run time of one positive commutation check from 220 minutes with our, admittedly inefficiently implemented[1], first prototype to around 30 seconds with the fastest SIMD-based executable we see our expectation that the commutation property is highly parallelizable confirmed.

The GPU-based approach provided an even better performance of around one second, although the results are of course not directly comparable. We deem the possibility of parallelizing this problem with the help of graphics hardware especially interesting, as the performance of GPUs right now seems to grow faster than that of ordinary processors.

At the same time we acknowledge that for domains larger than A_3, a "brute force" calculation of all centralizers can never succeed.

We, however, do hope that our tool will help researchers to test their ideas with relative ease, so that new results in clone theory can be discovered.

5.1 Lessons learned

While implementing the different approaches we, of course, made several mistakes. These mistakes, in our opinion, also deserve to be documented.

5.1.1 Correctness *Always* Reigns Supreme

It seems trivial, but a program that does the wrong thing fast is not as useful as a program that does the right thing slowly. But what we want to address here is a slightly more subtle point: often programmers think, that the "minor inaccuracies" introduced

[1] A subsequent prototype which was still naive but had no glaring inefficiencies still took around 60 minutes.

by programming mistakes will not influence the performance of a program much if the overall structure of the implemented algorithm does not change by fixing the bug. Given the abilities of modern compilers, this, however, is not true for many bugs. If a bug, for instance, alters the data flow in a way that allows some "incorrect" optimizations to happen, benchmarking an incorrect program is an exercise in futility.

The most trivial instance, which is not a bug *per se*, is not outputting the result of a computation during development when one tries to get an intuition about the performance of different parts of the program. Oftentimes, the compiler may identify a computation whose result is discarded as side-effect free and does not generate code for it at all – likely the fastest possible variant but not what we want.

Another example were a similar phenomenon could be observed was the selective evaluation from subsection 4.1.4: Before thorough testing the functions which decided if a evaluation step was necessary was incorrect and deemed many necessary evaluations unnecessary. This led to faster run times as less code was executed, but of course did not provide correct outputs.

We conclude this subsection with an example from the CUDA-based approach. Due to a missing compiler flag, the kernel for evaluating the function was not executed when using groups of matrices over a certain size, leaving the result array filled with zeroes – as initialized. Since we were testing only with commuting functions at that time, the error went unnoticed which led to some wasted hours of benchmarking and "optimizing" a kernel which did not get executed. There are several ways which can mitigate problems of this kind: this specific problem would not have occurred if we had not initialized our result with the expected value. In general only thorough testing can give reasonable confidence in the correct execution behavior of a program in the real world – even more so as the program itself was correct in this case.

5.1.2 Try to Eliminate Noise When Benchmarking

Another time sink when benchmarking the CUDA-based approach was that CUDA, then unbeknownst to us, needs a complex initialization procedure which gets called lazily on the first call to a function from the CUDA-API. When working only with small problem instances the run time cost of the initialization easily dwarves the run time of the actual computation. Additionally, the time that initialization takes varies between executions, which makes it impossible to recognize small changes in the performance. The solution to this problem is to initialize CUDA explicitly and then measuring the execution time from that moment on. Since CUDA provides no dedicated initialization function, the idiomatic way to do this is a call to `cudaFree(0)`.

When we are instrumenting our code manually, we may introduce significant noise to the performance. One example for this is shown in figure 4.11. It is therefore important that we always consider the run time of the non-instrumented versions too when we draw our conclusions. It shall be noted explicitly that instrumentation also may change the run time behavior of non-instrumented parts of the code. The, to us, most surprising

```
for (uint64_t i = 0; i < vec1.size(); ++i) {
    for (uint64_t j = A1 != A2 ? 0 : i; j < vec2.size(); ++j) {
        std::string id1 = to_string(i) + "/" + to_string(A1);
        std::string id2 = to_string(j) + "/" + to_string(A2);

        if (solver<D, A1, A2>::commutes(vec1[i], vec2[j])) {
            matches[id1].insert(id2);
            matches[id2].insert(id1);
        }
    }
}
```

Figure 5.1: Example of needlessly slow code.

instance of this phenomenon was that measuring the time a single function call on the call site changed the run time of the function call.

5.1.3 Subtle Changes May Have a Huge Impact on the Run Time

The following example would also fit in the preceding subsection, but at the same time it demonstrates the limits that compilers face in certain situations. Consider the code in figure 5.1. While testing with large, commuting functions, the code in question did not show any obvious weaknesses. When we, however, went to big amounts of small, non-commuting functions the construction of the string ids, which only would have been needed in the commuting case, took much more time than the commutation test itself. The solution was, of course, to move the declaration of the strings into the if block – an optimization we deemed within the possibilities of modern compilers.

As discussed in subsection 4.1.3, when we need two 128-bit registers for our matrices when evaluating functions of arity ≥ 3. A natural way to handle this case would have been implementing an abstract matrix type that holds two registers in case of arity 3 and 4 functions and only one register for smaller arities. Unfortunately, this layer of indirection obviously confused the compiler and the produced code turned out to be significantly slower. We solved this problem by implementing different classes depending on the arities of the functions. This lead to code duplication, with all the negative consequences on maintainability, but on the other hand conserved the good run time behavior.

5.2 Prospects

After spending a considerable time implementing the different approaches what are our insights and what potential improvements do we deem possible?

The SIMD-based approach took by far the most time to implement, but has the advantage that it runs on many computers and is easily distributable on clusters. Furthermore, at

the time of writing many newer processors already supported 256-bit[2] registers and more advanced instruction sets, which may cut the execution time in the worst case in half. On the other hand, for bigger domains or functions of bigger arities the SIMD-based approach may not be suited.

This has several reasons. First of all A_4 has many nice properties: we can use overflows that "naturally" happen for matrix generation, a call to `_mm_shuffle_epi8` can express the evaluation of two parameters, the encoded 2-bit integers are always neatly aligned in our registers and we can use fast bit operations, like shifts, instead of slow operations, like divisions, in many cases. All of these properties are, for example, missing for A_5. Furthermore, the SIMD-based approach already takes tens of seconds in the worst case (that is, when probing all 4^{16} matrices). Even if the evaluation of arity 5 functions would be as fast as the evaluation of arity 4 functions[3], we would still have to check 4^{25}, that is 512 times more, matrices in the worst case. For larger domains, like already A_5, the SIMD-based approach would take years on a single CPU to check only two functions (with arities ≥ 5).

A possibility to manage this complexity would be parallelizing the commutation test itself over multiple CPUs.

The CUDA-based approach can remedy some of these problems: first of all it is comparatively simple and therefore easier to extend. Secondly, it is faster in absolute terms, although the sheer number of matrices for bigger domains of course is also a problem for this approach.

Another, yet unexplored, possibility would be the distribution of the workload over several computers in a cluster via OpenMPI [Ope12]. Unfortunately, we could not try this, since the CPUs in our available OpenMPI cluster did support some crucial SSE instructions.

If one were now to attempt computing the commutation property of functions (with arity 4) on a grand scale, this is how we would recommend to proceed:

- Use existing theoretical knowledge to prune as many functions upfront from the set of functions to test.
- Use the SIMD-based approach to proof non-commutation for many functions, by considering only the first n matrices (with e.g. $n = 1024$). This should already suffice for most functions to show non-commutation.
- Use the CUDA-based approach to check the remaining functions, which are now more likely to commute.

This thesis just considered the parallelizability of the commutation property by using a "brute force" approach which did not try to reduce the number of checked matrices.

[2] The AVX-512 architecture even provides 512-bit registers.

[3] which it likely would not be

How to separate primitive positive clones from each other by constructing functions in an intelligent way is discussed in Artem Revenko's dissertation [Rev15]. To which extent the two approaches are compatible remains an open question.

List of Figures

Bibliography

[Ale01] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley, 2001.

[BCRV03] Elmar Böhler, Nadia Creignou, Steffen Reith, and Heribert Vollmer. Playing with boolean blocks, part i: Post's lattice with applications to complexity theory. volume 35, pages 22–35. ACM, 2003.

[BCRV04] Elmar Böhler, Nadia Creignou, Steffen Reith, and Heribert Vollmer. Playing with boolean blocks, part ii: Constraint satisfaction problems. volume 34, pages 38–52. ACM, 2004.

[Beh14] Mike Behrisch. Notes on commutation and centraliser clones. Technical report, Vienna University of Technology, 2014.

[BW87] St Burris and R Willard. Finitely many primitive positive clones. *Proceedings of the American Mathematical Society*, 101(3):427–430, 1987.

[Cor] NVIDIA Corporation. Cuda parallel computing platform. `http://www.nvidia.com/object/cuda_home_new.html`. Accessed: 2016-04-15.

[Dan77] Anna Fedorovna Danil'chenko. Parametric expressibility of functions of three-valued logic. *Algebra and Logic*, 16(4):266–280, 1977.

[Dan79] Anna Fedorovna Danil'chenko. *Questions on the expressibility of functions of three-valued logic (In Russian)*. PhD thesis, Academy of Sciences of the Moldavian Soviet Socialist Republic, 1979.

[Fly72] Michael J Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972.

[FR96] Michael J Flynn and Kevin W Rudd. Parallel architectures. *ACM Computing Surveys (CSUR)*, 28(1):67–70, 1996.

[Gro] Khronos Group. The open standard for parallel programming of heterogeneous systems. `https://www.khronos.org/opencl/`. Accessed: 2016-04-15.

[Gro13] The Open Group. The open group base specifications issue 7 IEEE Std 1003.1, 2013 edition, pthreads. `http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html`, 2013. Accessed: 2016-04-15.

[H+07] Mark Harris et al. Optimizing parallel reduction in cuda. *NVIDIA Developer Technology*, 2(4), 2007.

[Her08] Miki Hermann. On boolean primitive positive clones. *Discrete Mathematics*, 308(15):3151–3162, 2008.

[Int07] Intel. Intel® C++ intrinsic reference. `https://software.intel.com/sites/default/files/a6/22/18072-347603.pdf`, 2007.

[JCG97] Peter Jeavons, David Cohen, and Marc Gyssens. Closure properties of constraints. *Journal of the ACM (JACM)*, 44(4):527–548, 1997.

[Kuz79] A. V. Kuznetsov. On detecting non-deducibility and non-expressibility. In *Logical Deduction*, pages 5–33. Nauka, Moscow, 1979.

[Lew79] Harry R. Lewis. Satisfiability problems for propositional calculi. *Mathematical Systems Theory*, 13(1):45–53, 1979.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[ND10] John Nickolls and William J Dally. The gpu computing era. *IEEE micro*, (2):56–69, 2010.

[Ope12] MPI Open. Open source high performance computing. `https://www.open-mpi.org/`, 2012.

[Ope13] ARB OpenMP. Openmp 4.0 specification, june 2013. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`, July 2013.

[Rev15] Artem Revenko. *Automatic Construction of Implicative Theories for Mathematical Domains*. PhD thesis, Technische Universität Dresden, 2015.

[RJ10] Arch D. Robison and Ralph E. Johnson. Three layer cake for shared-memory programming. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ParaPLoP '10, pages 5:1–5:8, New York, NY, USA, 2010. ACM.

[Sch78] T. J. Schaefer. The complexity of satisfiability problems. In *Proceedings 10th Symposium on Theory of Computing (STOC'78), San Diego (California, USA)*, pages 216–226, 1978.

[SL08] Michael Süß and Claudia Leopold. Common mistakes in openmp and how to avoid them. In *OpenMP Shared Memory Parallel Programming*, pages 312–323. Springer, 2008.

[VHJG95] John Vlissides, Richard Helm, Ralph Johnson, and Erich Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49(120):11, 1995.

[Zad07] Laszlo Zadori. Solvability of systems of polynomial equations over finite algebras. *International Journal of Algebra and Computation*, 17(04):821–835, 2007.