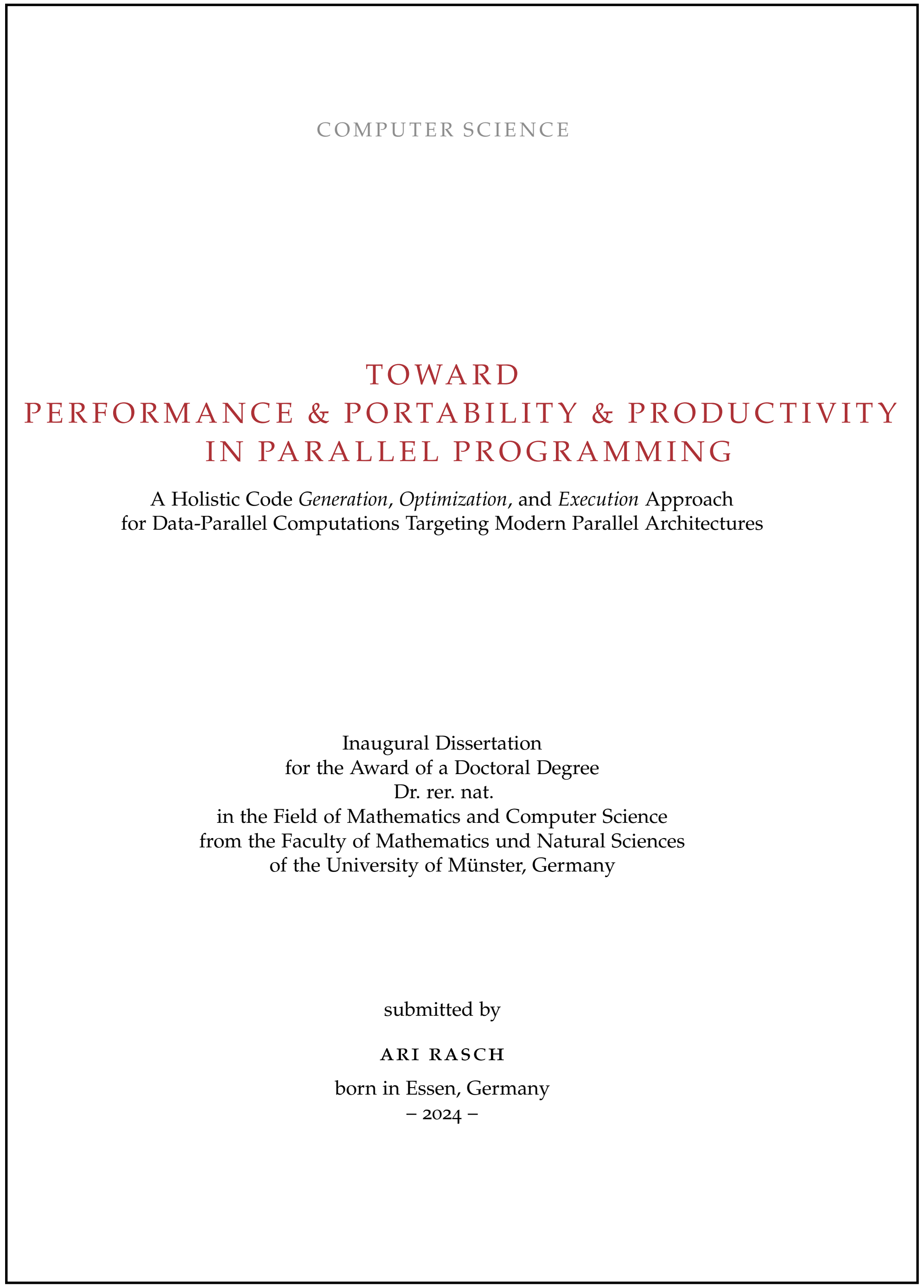
Toward Performance & Portability & Productivity in Parallel Programming

A Holistic Code *Generation*, *Optimization*, and *Execution* Approach for Data-Parallel Computations Targeting Modern Parallel Architectures

Ari Rasch
University of Münster, Germany

Introductory Remark


COMPUTER SCIENCE

TOWARD
PERFORMANCE & PORTABILITY & PRODUCTIVITY
IN PARALLEL PROGRAMMING

A Holistic Code *Generation*, *Optimization*, and *Execution* Approach
for Data-Parallel Computations Targeting Modern Parallel Architectures

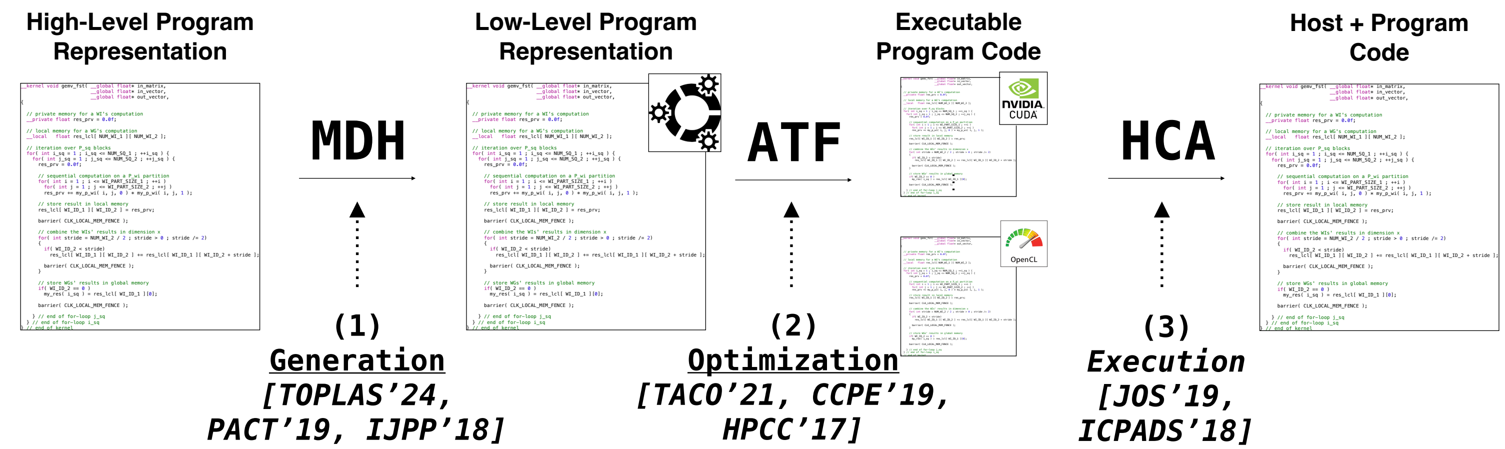
Inaugural Dissertation
for the Award of a Doctoral Degree
Dr. rer. nat.
in the Field of Mathematics and Computer Science
from the Faculty of Mathematics und Natural Sciences
of the University of Münster, Germany

submitted by

ARI RASCH

born in Essen, Germany
– 2024 –

<u>This thesis describes three major (sub-)projects:</u>



The sub-project complement each other to a holistic approach to code
<u>Generation</u> & <u>Optimization</u> & <u>Execution</u>

<u>Major</u> Contribution | **<u>Medium</u> Contribution** | **<u>Minor</u> Contribution**

Parallel Programming in Today's World

<u>Parallel programming is hard:</u>

Parallel Architectures

Programming Models

Microsoft

NVIDIA CUDA

OpenMP

intel

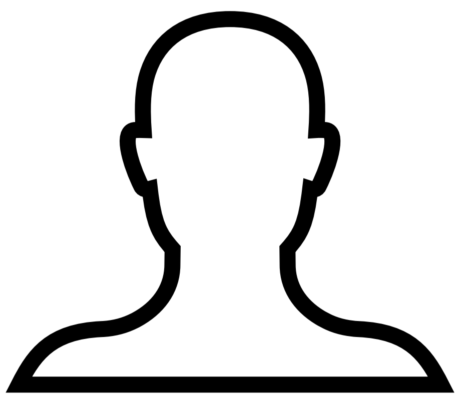
NVIDIA

tvm

OpenCL

...

Google



Domain Scientist

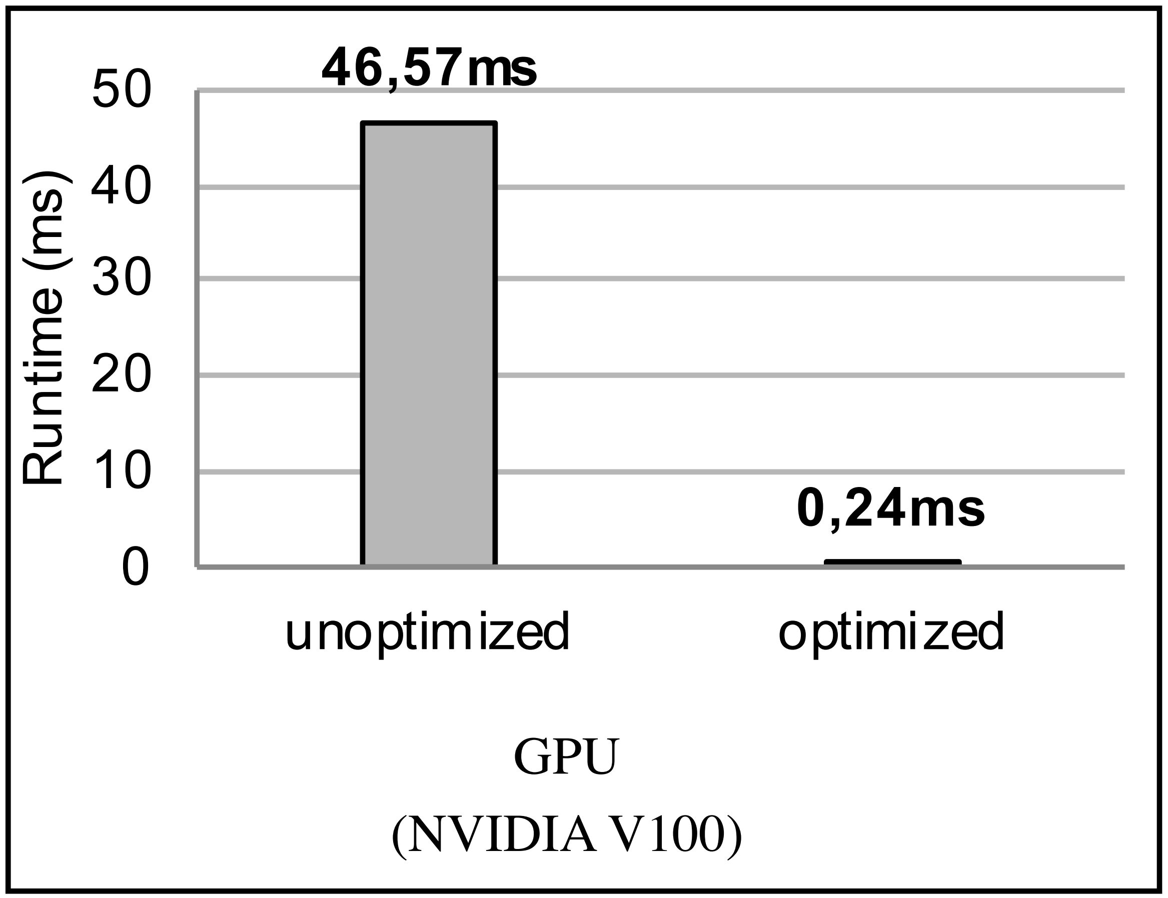
+ domain knowledge
- but lacks hardware & optimization details

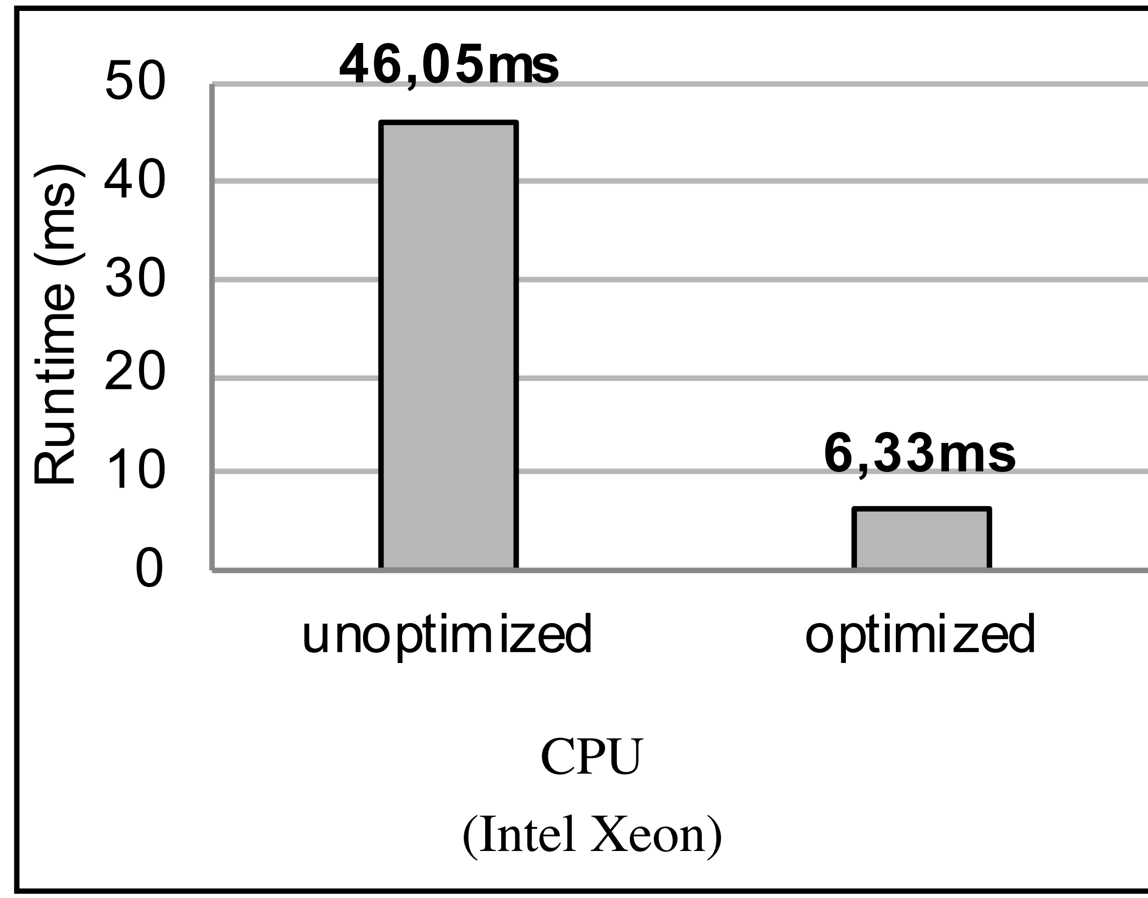
Struggle with
***simultaneously* achieving**
<u>Performance</u> & <u>Portability</u> & <u>Productivity</u>

+ tremendous performance
- require advanced & specific optimizations

Challenges: Performance & Portability & Productivity

The *Performance* challenge:



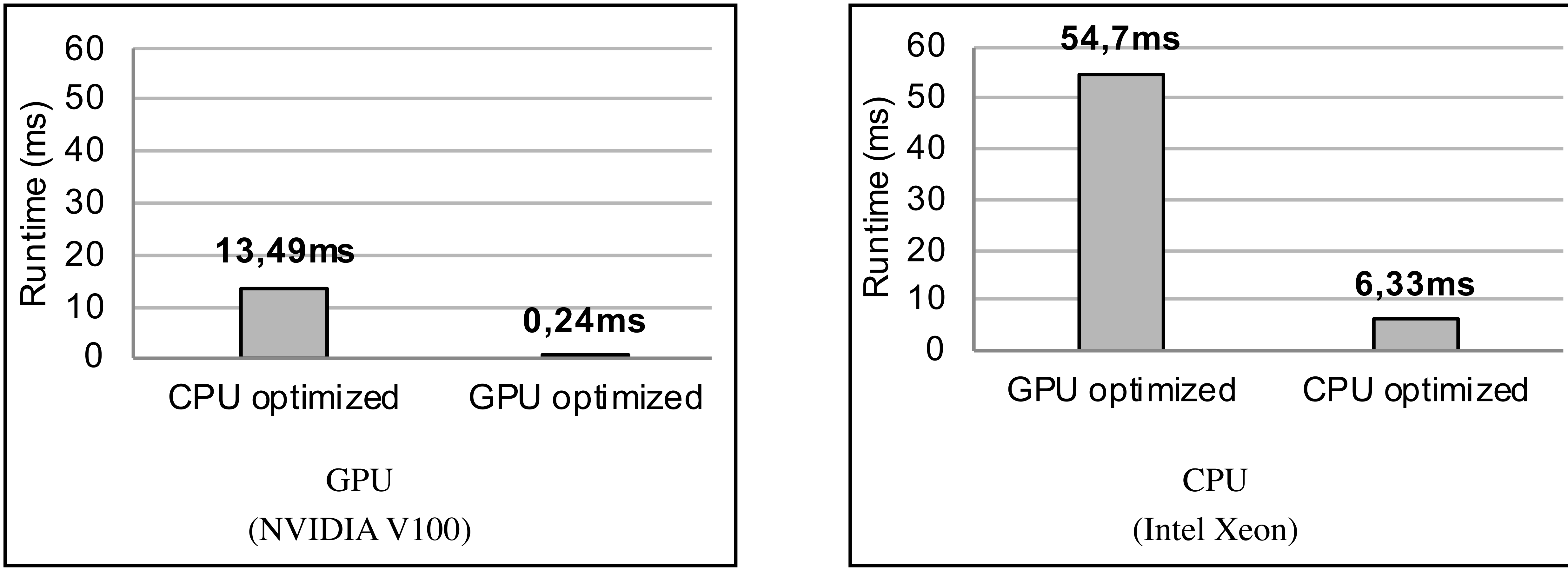


Runtime (lower is better) of unoptimized vs optimized matrix multiplication on GPU (left) and CPU (right).

High *Performance* requires *complex* optimizations

Challenges: Performance & Portability & Productivity

The Portability challenge:



Runtime (lower is better) of GPU/CPU-optimized matrix multiplication on GPU (left) and CPU (right).

High _Portability_ requires _architecture(/data)-specific_ optimizations

<u>Challenges:</u> Performance & Portability & Productivity

The Productivity challenge:

```
__kernel void MatMul( __global const float A[M][K] ,
                      __global const float B[K][N] ,
                      __global       float C[M][N] )
{
  int i = get_global_id(0);
  int j = get_global_id(1);

  for( int k=0 ; k<K ; ++k )
    C[i][j] += A[i][k] * B[k][j];
}
```

Naive OpenCL implementation of matrix multiplication

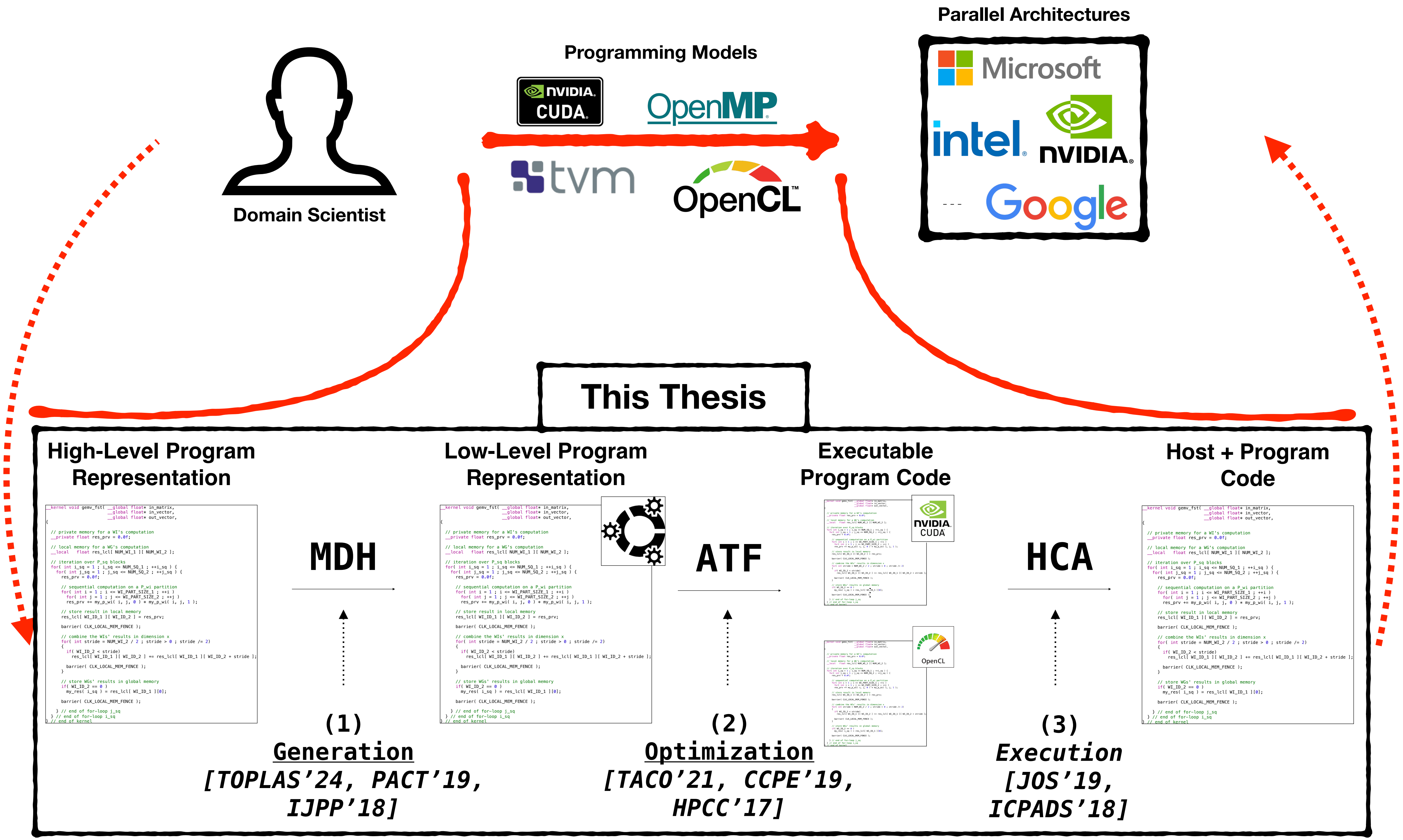
```
__kernel void MatMul( /* ... */ )
{
  const size_t i_wg_l_1 = get_group_id(2);
// ... 5 lines skipped

  __private TYPE_TS res_p[/*...*/][/*...*/];
  {
// ... 7 lines skipped
          for (size_t p_iteration_l_1 = 0; p_iteration_l_1 < (2);
              ++p_iteration_l_1) {
            for (size_t p_iteration_l_2 = 0; p_iteration_l_2 < (1)
                ; ++p_iteration_l_2) {
              size_t p_iteration_r_1 = 0;
              res_p[p_step_l_1][((p_iteration_l_1) * 1 + 0)][(0)][
                  p_step_l_2][((p_iteration_l_2) * 1 + 0)] = f(
                  a[(((l_step_l_1 x* (32 / 1) + (((p_step_l_1 *
                      (2) + (((p_iteration_l_1) * 1 + 0)) / 1) * 1
                       + i_wi_l_1 * 1 + ((((p_iteration_l_1) * 1 +
                       0)) % 1))) / 1) * (64 * 1) + i_wg_l_1 * 1 +
                       ((((p_step_l_1 * (2) + (((p_iteration_l_1)
                      * 1 + 0)) / 1) * 1 + i_wi_l_1 * 1 + ((((
                      p_iteration_l_1) * 1 + 0)) % 1))) % 1))) *
                      1024 + (((l_step_r_1 * (2 / 1) + (((
                      p_step_r_1 * (1) + (((p_iteration_r_1) * 1 +
                       0)) / 1) * 1 + i_wi_r_1 * 1 + ((((
                      p_iteration_r_1) * 1 + 0)) % 1))) / 1) * (2
                      * 1) + i_wg_r_1 * 1 + ((((p_step_r_1 * (1) +
                       (((p_iteration_r_1) * 1 + 0)) / 1) * 1 +
                      i_wi_r_1 * 1 + ((((p_iteration_r_1) * 1 + 0)
                      ) % 1))) % 1)))],
// ... 107 lines skipped
}
```

Optimized OpenCL implementation of matrix multiplication

High <u>*Productivity*</u> requires <u>*automatic* optimization</u>

Contributions of this Thesis

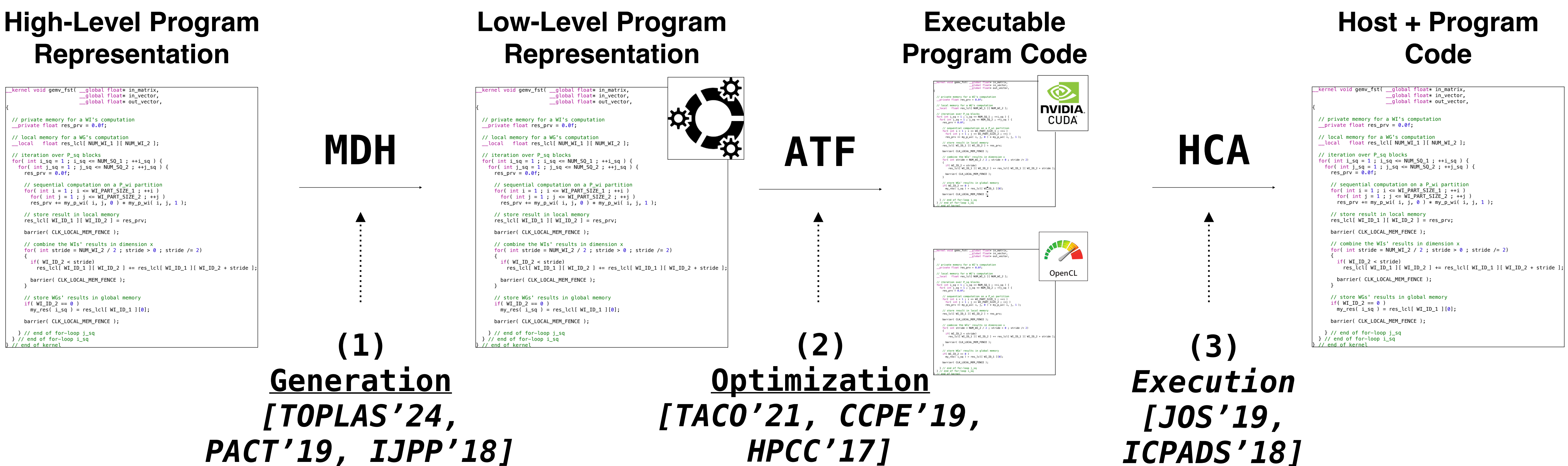
<u>This thesis introduces a novel, holistic approach to ***Generating*** & ***Optimizing*** & ***Executing*** code:</u>



The ultimate goal of **MDH+ATF+HCA** is to simultaneously achieve **<u>Performance</u>** & **<u>Portability</u>** & **<u>Productivity</u>**

Outline

<u>This talk(/thesis) is structured into three main parts:</u>



1. Part: How to *generate* automatically optimizable (auto-tunable) code?

2. Part: How to automatically *optimize* (auto-tune) code?

3. Part: How to *execute* code on (distr.) multi-dev. systems?

Code Generation via **MDH**

Overview Getting Started Code Examples Publications Citations Contact

MDH

Multi-Dimensional Homomorphisms (MDH)

An Algebraic Approach Toward Performance & Portability & Productivity for Data-Parallel Computations

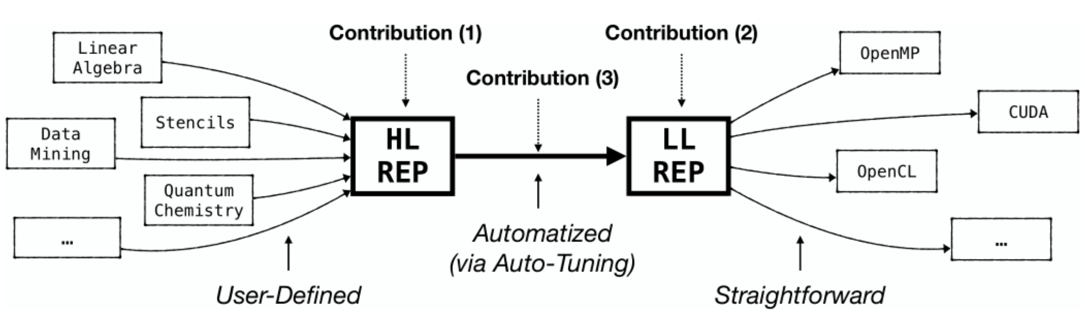
Overview

The approach of Multi-Dimensional Homomorphisms (MDH) is an algebraic formalism for systematically reasoning about *de-composition* and *re-composition* strategies of data-parallel computations (such as linear algebra routines and stencil computations) for the memory and core hierarchies of state-of-the-art parallel architectures (GPUs, multi-core CPU, multi-device and multi-node systems, etc).

The MDH approach (formally) introduces:

1. *High-Level Program Representation* *(Contribution 1)* that enables the user conveniently implementing data-parallel computations, agnostic from hardware and optimization details;
2. *Low-Level Program Representation* *(Contribution 2)* that expresses device- and data-optimized de- and re-composition strategies of computations;
3. *Lowering Process* *(Contribution 3)* that fully automatically lowers a data-parallel computation expressed in its high-level program representation to an optimized instance in its low-level representation, based on concepts from automatic performance optimization (a.k.a. *auto-tuning*), using the Auto-Tuning Framework (ATF).

The MDH's low-level representation is designed such that Code Generation from it (e.g., in OpenMP for CPUs, CUDA for NVIDIA GPUs, or OpenCL for multiple kinds of architectures) becomes straightforward.



Our Experiments report encouraging results on GPUs and CPUs for MDH as compared to state-of-practice approaches, including NVIDIA cuBLAS/cuDNN and Intel oneMKL/oneDNN which are hand-optimized libraries provided by vendors.

https://mdh-lang.org

ACM TOPLAS 2024

(De/Re)-Composition of Data-Parallel Computations via Multi-Dimensional Homomorphisms

ARI RASCH, University of Muenster, Germany

Data-parallel computations, such as linear algebra routines and stencil computations, constitute one of the most relevant classes in parallel computing, e.g., due to their importance for deep learning. Efficiently de-composing such computations for the memory and core hierarchies of modern architectures and re-composing the computed intermediate results back to the final result—we say *(de/re)-composition* for short—is key to achieve high performance for these computations on, e.g., GPU and CPU. Current high-level approaches to generating data-parallel code are often restricted to a particular subclass of data-parallel computations and architectures (e.g., only linear algebra routines on only GPU or only stencil computations), and/or the approaches rely on a user-guided optimization process for a well-performing (de/re)-composition of computations, which is complex and error prone for the user.

We formally introduce a systematic (de/re)-composition approach, based on the algebraic formalism of Multi-Dimensional Homomorphisms (MDHs). Our approach is designed as general enough to be applicable to a wide range of data-parallel computations and for various kinds of target parallel architectures. To efficiently target the deep and complex memory and core hierarchies of contemporary architectures, we exploit our introduced (de/re)-composition approach for a correct-by-construction, parametrized cache blocking, and parallelization strategy. We show that our approach is powerful enough to express, in the same formalism, the (de/re)-composition strategies of different classes of state-of-the-art approaches (scheduling-based, polyhedral, etc.), and we demonstrate that the parameters of our strategies enable systematically generating code that can be fully automatically optimized (auto-tuned) for the particular target architecture and characteristics of the input and output data (e.g., their sizes and memory layouts). Particularly, our experiments confirm that via auto-tuning, we achieve higher performance than state-of-the-art approaches, including hand-optimized solutions provided by vendors (such as NVIDIA cuBLAS/cuDNN and Intel oneMKL/oneDNN), on real-world datasets and for a variety of data-parallel computations, including linear algebra routines, stencil and quantum chemistry computations, data mining algorithms, and computations that recently gained high attention due to their relevance for deep learning.

CCS Concepts: • **Computing methodologies → Parallel computing methodologies**; *Machine learning*; • **Theory of computation → Program semantics**; • **Software and its engineering → Compilers**;

Additional Key Words and Phrases: Code generation, data parallelism, auto-tuning, GPU, CPU, OpenMP, CUDA, OpenCL, linear algebra, stencils computation, quantum chemistry, data mining, deep learning

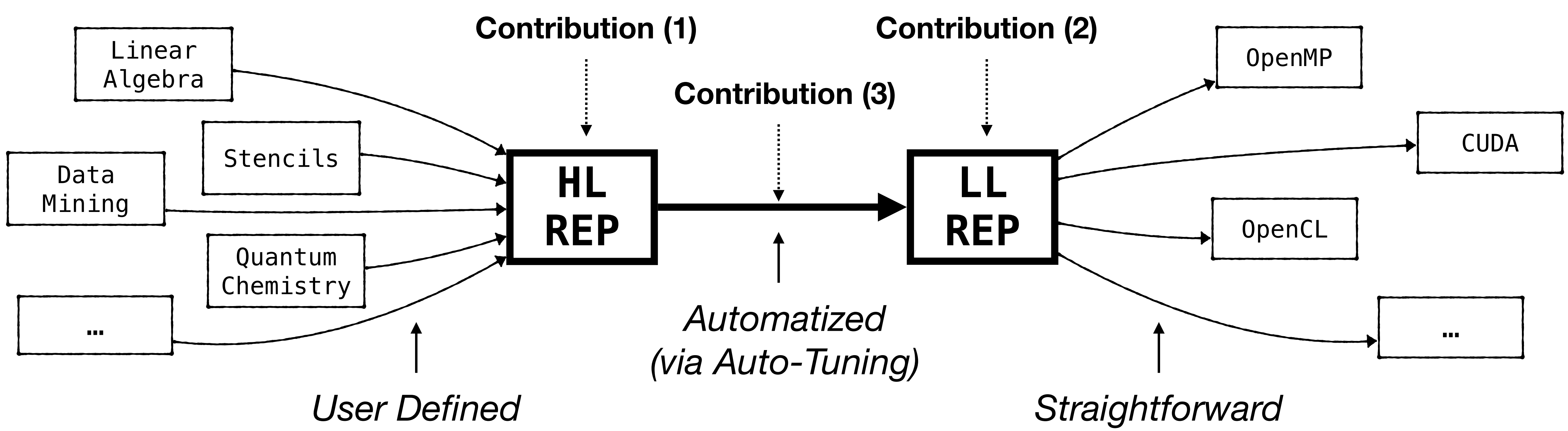
A full version of this article is provided by Rasch [2024], which presents our novel concepts with all of their formal details. In contrast to the full version, this article relies on a simplified formal foundation for better illustration and easier understanding. We often refer the interested reader to Rasch [2024] for formal details that should not be required for understanding the basic ideas and concepts of our approach.

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)—project PPP-DL (470527619).

Author's Contact Information: Ari Rasch (Corresponding author), University of Muenster, Muenster, Germany; e-mail: a.rasch@uni-muenster.de.

Code Generation via **MDH**

MDH is a (formal) framework for expressing & optimizing data-parallel computations:



1. ***Contribution 1 (HL-REP):*** defines *data parallelism* & introduces *higher-order functions* for expressing data-parallel computations, agnostic from hardware and optimization details while still capturing high-level information relevant for generating high-performing code

2. ***Contribution 2 (LL-REP):*** allows *expressing and reasoning about optimizations* for the memory and core hierarchies of contemporary parallel architectures & generalizes these optimizations to apply to arbitrary combinations of data-parallel computations and architectures

3. ***Contribution 3 (→):*** introduces a *structured optimization process* — for arbitrary combinations of data-parallel computations and parallel architectures — that allows *fully automatic* optimization (auto-tuning)

Code Generation via **MDH**

Example: `MatVec` expressed in MDH

```
MatVec<T∈TYPE|I,K∈ℕ>  :=  out_view<T>( w:(i,k)↦(i) ) ∘
                              md_hom<I,K>( *, (++,+) ) ∘
                                  inp_view<T,T>( M:(i,k)↦(i,k) , v:(i,k)↦(k) )
```

High-Level Representation of MatVec

What is happening here:

- `inp_view` captures the accesses to input data
- `md_hom` expresses the data-parallel computation
- `out_view` captures the accesses to output data

[1]*We can generate such MDH expressions also automatically from straightforward (annotated) code in Python or C*

Code Generation via **MDH**

md_hom	f	$\circledast_1$	$\circledast_2$	$\circledast_3$	$\circledast_4$
Dot	*	+	/	/	/
MatVec	*	++	+	/	/
MatMul	*	++	++	+	/
MatMul^T	*	++	++	+	/
bMatMul	*	++	++	++	+

Views	inp_view		out_view
	A	B	C
Dot	(k) ↦ (k)	(k) ↦ (k)	(k) ↦ ()
MatVec	(i,k) ↦ (i,k)	(i,k) ↦ (k)	(i,k) ↦ (i)
MatMul	(i,j,k) ↦ (i,k)	(i,j,k) ↦ (k,j)	(i,j,k) ↦ (i,j)
MatMul^T	(i,j,k) ↦ (k,i)	(i,j,k) ↦ (j,k)	(i,j,k) ↦ (j,i)
bMatMul	(b,i,j,k) ↦ (b,i,k)	(b,i,j,k) ↦ (b,k,j)	(b,i,j,k) ↦ (b,i,j)

1) Linear Algebra Routines

md_hom	f	$\circledast_1$	$\circledast_2$	$\circledast_3$	$\circledast_4$	$\circledast_5$	$\circledast_6$	$\circledast_7$	$\circledast_8$	$\circledast_9$	$\circledast_{10}$
Conv2D	*	++	++	+	+	/	/	/	/	/	/
MCC	*	++	++	++	++	+	+	+	/	/	/
MCC_Capsule	*	++	++	++	++	+	+	+	++	++	+

Views	inp_view		out_view
	I	F	O
Conv2D	(p,q,r,s) ↦ (p+r,q+s)	(p,q,r,s) ↦ (r,s)	(p,q,r,s) ↦ (p,q)
MCC	(n,p,...) ↦ (n,p+r,q+s,c)	(n,p,...) ↦ (k,r,s,c)	(n,p,...) ↦ (n,p,q,k)
MCC_Capsule	(n,p,...) ↦ (n,p+r,q+s,c,cm,ck)	(n,p,...) ↦ (k,r,s,c,ck,cn)	(n,p,...) ↦ (n,p,q,k,cm,cn)

2) Convolution Stencils

md_hom	f	$\circledast_1$	$\circledast_2$
MBBS	id	$++_{\texttt{prefix-sum}}(+)$	+

Views	inp_view	out_view
	A	Out
MBBS	(i,j) ↦ (i,j)	(i) ↦ (i)

8) Maximum Bottom Box Sum

md_hom	f	$\circledast_1$	$\circledast_2$
Jacobi1D	J_{1D}	++	/
Jacobi2D	J_{2D}	++	++

Views	inp_view	out_view
	I	O
Jacobi1D	(i) ↦ (i+0) , (i) ↦ (i+1) , (i) ↦ (i+2)	(i) ↦ (i)
Jacobi2D	(i,j) ↦ (i,j+1) , (i,j) ↦ (i+1,j) , ...	(i,j) ↦ (i,j)

3) Jacobi Stencils

md_hom	f	$\circledast_1$	$\circledast_2$
PRL	wght	++	$\texttt{max}_{\texttt{PRL}}$

Views	inp_view		out_view
	N	E	M
PRL	(i,j) ↦ (i)	(i,j) ↦ (j)	(i,j) ↦ (i)

4) Probabilistic Record Linkage

md_hom	f	$\circledast_1$
map(f)	f	++
reduce(⊕)	id	⊕
reduce(⊕,⊗)	(x) ↦ (x,x)	(⊕,⊗)

Views	inp_view	out_view	
	I	O_1	O_2
map(f)	(i) ↦ (i)	(i) ↦ (i)	/
reduce(⊕)	(i) ↦ (i)	(i) ↦ ()	/
reduce(⊕,⊗)	(i) ↦ (i)	(i) ↦ ()	(i) ↦ ()

6) Map/Reduce Patterns

md_hom	f	$\circledast_1$
scan(⊕)	id	$++_{\texttt{prefix-sum}}(\oplus)$

Views	inp_view	out_view
	I	O
scan(⊕)	(i) ↦ (i)	(i) ↦ (i)

7) Scan Pattern

md_hom	f	$\circledast_1$	$\circledast_2$
Histo	$f_{\texttt{Histo}}$	++	+

Views	inp_view		out_view
	Bins	Elems	Out
Histo	(b,e) ↦ (b)	(b,e) ↦ (e)	(b,e) ↦ (b)

5) Histogram

MDH is capable of **expressing a wide range of data-parallel computations** from **popular domains**

Code Generation via **MDH**

Performance Evaluation: (via runtime comparison)

Deep Learning	NVIDIA Ampere GPU									
	ResNet-50				VGG-16				MobileNet	
	Training		Inference		Training		Inference		Training	Inference
	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MCC
TVM+Ansor	1.00	1.26	1.05	2.22	0.93	1.42	0.88	1.14	0.94	1.00
PPCG	3456.16	8.26	–	7.89	1661.14	7.06	5.77	5.08	2254.67	7.55
PPCG+ATF	3.28	2.58	13.76	5.44	4.26	3.92	9.46	3.73	3.31	10.71
cuDNN	0.92	–	1.85	–	1.22	–	1.94	–	1.81	2.14
cuBLAS	–	1.58	–	2.67	–	0.93	–	1.04	–	–
cuBLASEx	–	1.47	–	2.56	–	0.92	–	1.02	–	–
cuBLASLt	–	1.26	–	1.22	–	0.91	–	1.01	–	–

Highlights only

MDH speedup over

- TVM: 0.88x – 2.22x
- PPCG: 2.58x – 13.76x
- *(cuBLAS/cuDNN: 0.91x – 2.67x)*

Deep Learning	Intel Skylake CPU									
	ResNet-50				VGG-16				MobileNet	
	Training		Inference		Training		Inference		Training	Inference
	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MatMul	MCC	MCC
TVM+Ansor	1.53	1.05	1.14	1.20	1.97	1.14	2.38	1.27	3.01	1.40
Pluto	355.81	49.57	364.43	13.93	130.80	93.21	186.25	36.30	152.14	75.37
Pluto+ATF	13.08	19.70	170.69	6.57	3.11	6.29	53.61	8.29	3.50	25.41
oneDNN	0.39	–	5.07	–	1.22	–	9.01	–	1.05	4.20
oneMKL	–	0.44	–	1.09	–	0.88	–	0.53	–	–
oneMKL(JIT)	–	6.43	–	8.33	–	27.09	–	9.78	–	–

MDH speedup over

- TVM: 1.05 – 3.01x
- Pluto: 6.29x – 364.43x
- *(oneMKL/oneDNN: 0.39x – 9.01x)*

Case Study "Deep Learning" for which most competitors are highly optimized (most challenging for us!)

Significantly higher speedups for other case studies,
e.g., >170x over TVM on GPU already for straightforward dot products

Code Optimization via **ATF**

Auto-Tuning Framework (ATF)

Efficient Auto-Tuning of Parallel Programs with Constrained Tuning Parameters

Overview

The Auto-Tuning Framework (ATF) is a general-purpose auto-tuning approach: given a program that is implemented as generic in performance-critical program parameters (a.k.a. *tuning parameters*), such as sizes of tiles and numbers of threads, ATF fully automatically determines a hardware- and data-optimized configuration of such parameters.

Key Feature of ATF

A key feature of ATF is its support for *Tuning Parameter Constraints*. Parameter constraints allow auto-tuning programs whose tuning parameters have so-called *interdependencies* among them, e.g., the value of one tuning parameter has to evenly divide the value of another tuning parameter.

ATF's support for parameter constraints is important: modern parallel programs target novel parallel architectures, and such architectures typically have deep memory and core hierarchies thus requiring constraints on tuning parameters, e.g., the value of a tile size tuning parameter on an upper memory layer has to be a multiple of a tile size value on a lower memory layer.

For such parameters, ATF introduces novel concepts for *Generating* & *Storing* & *Exploring* the search spaces of constrained tuning parameters, thereby contributing to a substantially more efficient overall auto-tuning process for such parameters, as confirmed in our *Experiments*.

Generality of ATF

For wide applicability, ATF is designed as generic in:

1. The target program's Programming Language, e.g., *C/C++*, *CUDA*, *OpenMP*, or *OpenCL*. ATF offers *pre-implemented cost functions* for conveniently auto-tuning C/C++ programs, as well as CUDA and OpenCL kernels which require host code for their execution which is automatically generated and executed by ATF's pre-implemented CUDA and OpenCL cost functions. ATF also offers a pre-implemented *generic* cost function that can be used for conveniently auto-tuning programs in any other programming language different from C/C++, CUDA, and OpenCL.

https://atf-tuner.org

ACM TACO 2021

Efficient Auto-Tuning of Parallel Programs with Interdependent Tuning Parameters via Auto-Tuning Framework (ATF)

ARI RASCH and RICHARD SCHULZE, University of Muenster, Germany
MICHEL STEUWER, University of Edinburgh, United Kingdom
SERGEI GORLATCH, University of Muenster, Germany

Auto-tuning is a popular approach to program optimization: it automatically finds good configurations of a program's so-called tuning parameters whose values are crucial for achieving high performance for a particular parallel architecture and characteristics of input/output data. We present three new contributions of the Auto-Tuning Framework (ATF), which enable a key advantage in *general-purpose auto-tuning*: efficiently optimizing programs whose tuning parameters have *interdependencies* among them. We make the following contributions to the three main phases of general-purpose auto-tuning: (1) ATF *generates* the search space of interdependent tuning parameters with high performance by efficiently exploiting parameter constraints; (2) ATF *stores* such search spaces efficiently in memory, based on a novel chain-of-trees search space structure; (3) ATF *explores* these search spaces faster, by employing a multi-dimensional search strategy on its chain-of-trees search space representation. Our experiments demonstrate that, compared to the state-of-the-art, general-purpose auto-tuning frameworks, ATF substantially improves generating, storing, and exploring the search space of interdependent tuning parameters, thereby enabling an efficient overall auto-tuning process for important applications from popular domains, including stencil computations, linear algebra routines, quantum chemistry computations, and data mining algorithms.

CCS Concepts: • **General and reference** → **Performance**; • **Computer systems organization** → **Parallel architectures**; • **Software and its engineering** → **Parallel programming languages**;

Additional Key Words and Phrases: Auto-tuning, parallel programs, interdependent tuning parameters

ACM Reference format:
Ari Rasch, Richard Schulze, Michel Steuwer, and Sergei Gorlatch. 2021. Efficient Auto-Tuning of Parallel Programs with Interdependent Tuning Parameters via Auto-Tuning Framework (ATF). *ACM Trans. Archit. Code Optim.* 18, 1, Article 1 (January 2021), 26 pages.
https://doi.org/10.1145/3427093

This is a new paper, not an extension of a conference paper.
Authors' addresses: A. Rasch, R. Schulze, and S. Gorlatch, University of Muenster, Muenster, Germany; emails: {a.rasch, r.schulze, gorlatch}@uni-muenster.de; M. Steuwer, University of Edinburgh, Edinburgh, United Kingdom; email: michel.steuwer@glasgow.ac.uk.

1544-3566/2021/01-ART1

Code Optimization via ATF

<u>Advantage of ATF over state-of-the-art general-purpose AT approaches:</u>

ATF finds values of performance-critical parameters with

***<u>interdependencies</u>* <u>among them</u>**

via optimized processes to

<u>generating</u> & *<u>storing</u>* & *<u>exploring</u>*

the spaces of interdependent parameters

→ We illustrate ATF by comparing it to MIT's *OpenTuner* [PACT'14] & *CLTune* [MCSoC'15] which is the foundation of many related approaches (e.g., KernelTuner & KTT).

<u>Note:</u> BaCO [ASPLOS'23] & KTT recently adopted the ATF techniques to also efficiently handle interdependent tuning parameters.

Code Optimization via ATF

How does ATF achieve its efficiency for *interdependent tuning parameters*:

ATF introduces *parameter constraints*

```
tuner.addParameter( “tp_1”, T1 );
tuner.addParameter( “tp_2", T2 );
// ...

tuner.addConstraint(
  [](T1 tp_1, T2 tp_2, … ) -> bool
  { /* ... */ }
```

CLTune constraints

Defined on:
search space **(CLTune)**
vs. *parameters* (ATF)

```
tuner.addParameter( “tp_1”, R1, [](T1 tp_1) -> bool { /* … */ } );
tuner.addParameter( “tp_2”, R2, [](T2 tp_2) -> bool { /* … */ } );
```

ATF constraints

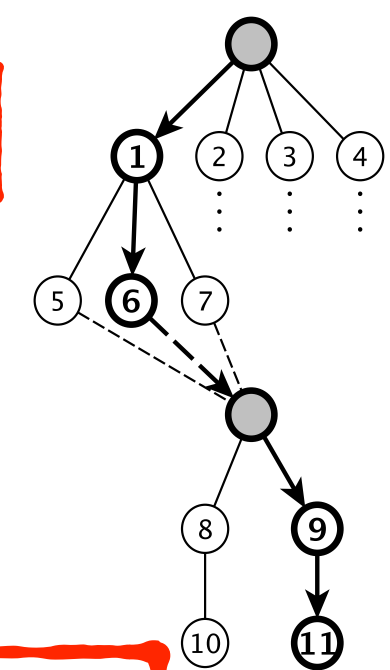
ATF introduces the
CoT (*Chain-of-Trees) space structure*)

```
SP :=[ (1,1) | (2,1) | (2,2) | … ]
```

CLTune search space

verbose & 1D **(CLTune)**
vs. *compact & nD* (ATF)

ATF
CoT search space



ATF introduces a novel
constraint design and search **space structure**
to efficiently **generate** & **store** & **explore** constrained search spaces

Highlights only

Code Optimization via ATF

ATF is able to auto-tune **modern** parallel computations, e.g., for ***GPUs & CPUs***:

`Stencil`

ATF is able to auto-tune `CONV` to:

>40x higher performance than CONV+**CLTune** on CPU

>10^4x higher performance than CONV+**CLTune** on GPU

>3x higher performance than **Intel MKL-DNN** on CPU

>15x higher performance than **NVIDIA cuDNN** on GPU

`Linear Algebra`

ATF is able to auto-tune `GEMM` to:

>2x higher performance than GEMM+**CLTune** on CPU

>120x higher performance than GEMM+**CLTune** on GPU

>2x higher performance than **Intel MKL** on CPU

>2x higher performance than **NVIDIA cuBLAS** on GPU

`Quantum Chemistry`

ATF is able to auto-tune `CCSD(T)` to:

>2x higher performance than **Facebook TC** on GPU

CLTune **fails**! (too high search space generation time)

`Data Mining`

ATF is able to auto-tune PRL to:

>1.6x higher performance than PRL+**CLTune** on CPU

>1.07x higher perform. than PRL+**CLTune** on GPU

`OpenTuner fails for all studies`

Code Execution via **HCA**

Overview Contact

HCA

Host Code Abstraction (HCA)

A High-Level Abstraction for Host Code Programming
Designed for Distributed, Heterogeneous Systems

Overview

The Host Code Abstraction (HCA) is a high-level programming abstraction that simplifies implementing and optimizing so-called host code which is required in modern parallel programming approaches (e.g., CUDA and OpenCL) to execute code on the devices of distributed, heterogeneous systems.

More details will follow soon!

Contact

https://hca-project.org

Skipped for brevity

Journal of Supercomputing 2019

The Journal of Supercomputing (2020) 76:5117–5138
https://doi.org/10.1007/s11227-019-02829-2

Check for updates

dOCAL: high-level distributed programming with OpenCL and CUDA

Ari Rasch[1] · Julian Bigge[1] · Martin Wrodarczyk[1] · Richard Schulze[1] · Sergei Gorlatch[1]

Published online: 30 March 2019

Abstract
In the state-of-the-art parallel programming approaches OpenCL and CUDA, so-called host code is required for program's execution. Efficiently implementing host code is often a cumbersome task, especially when executing OpenCL and CUDA programs on systems with multiple nodes, each comprising different devices, e.g., multi-core CPU and graphics processing units; the programmer is responsible for explicitly managing node's and device's memory, synchronizing computations with data transfers between devices of potentially different nodes and for optimizing data transfers between devices' memories and nodes' main memories, e.g., by using pinned main memory for accelerating data transfers and overlapping the transfers with computations. We develop distributed OpenCL/CUDA abstraction layer (dOCAL)—a novel high-level C++ library that simplifies the development of host code. dOCAL combines major advantages over the state-of-the-art high-level approaches: (1) it simplifies implementing both OpenCL and CUDA host code by providing a simple-to-use, high-level abstraction API; (2) it supports executing arbitrary OpenCL and CUDA programs; (3) it allows conveniently targeting the devices of different nodes by automatically managing node-to-node communications; (4) it simplifies implementing data transfer optimizations by providing different, specially allocated memory regions, e.g., pinned main memory for overlapping data transfers with computations; (5) it optimizes memory management by automatically avoiding unnecessary data transfers; (6) it enables interoperability between OpenCL and CUDA host code for systems with devices from different vendors. Our experiments show that dOCAL significantly simplifies the development of host code for heterogeneous and distributed systems, with a low runtime overhead.

Summary

<u>The **`MDH+ATF+HCA`** approach achieves ***Performance*** & ***Portability*** & ***Productivity*** for **data-parallel computations** targeting modern **parallel architectures**:</u>

COMPUTER SCIENCE

TOWARD
PERFORMANCE & PORTABILITY & PRODUCTIVITY
IN PARALLEL PROGRAMMING

A Holistic Code *Generation*, *Optimization*, and *Execution* Approach
for Data-Parallel Computations Targeting Modern Parallel Architectures

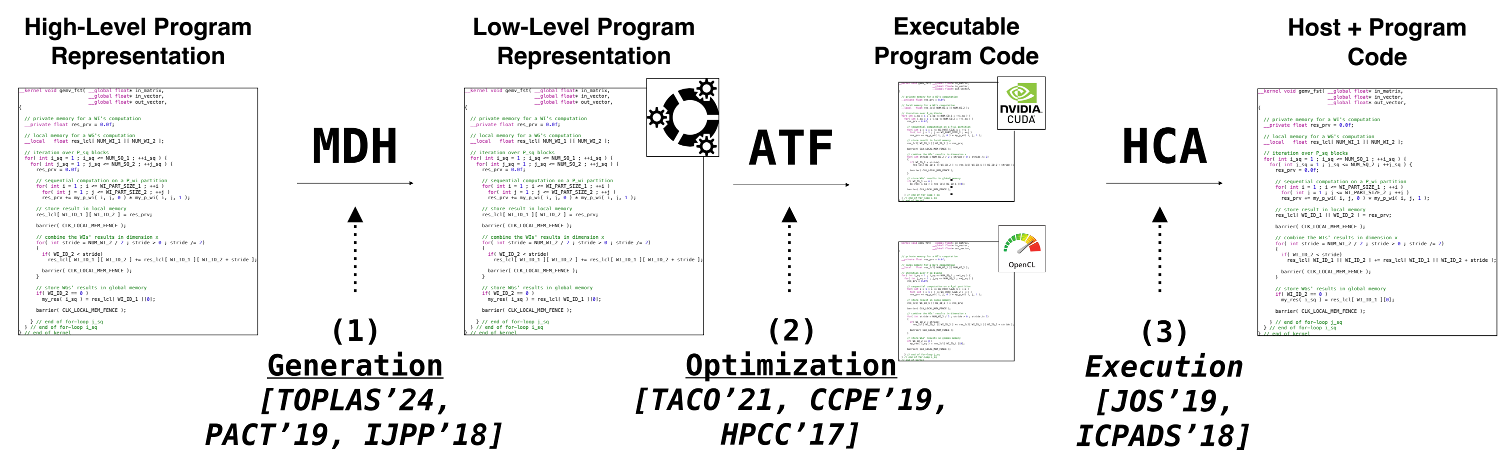
Inaugural Dissertation
for the Award of a Doctoral Degree
Dr. rer. nat.
in the Field of Mathematics and Computer Science
from the Faculty of Mathematics und Natural Sciences
of the University of Münster, Germany

submitted by

ARI RASCH

born in Essen, Germany
– 2024 –

<u>The MDH+ATF+HCA approach:</u>



- The three sub-projects — **`MDH`** & **`ATF`** & **`HCA`** — complement each other to a holistic code *Generation* & *Optimization* & *Execution* approach
- There are many (promising) future directions for MDH & ATF & HCA (one part of thesis dedicated to FW)

MDH **ATF** **HCA**