

Polyhedral Compilation: Applications, Approximations and GPU-specific Optimizations

Abhishek A. Patwardhan

A Thesis Submitted to
Indian Institute of Technology Hyderabad
In Partial Fulfillment of the Requirements for
The Degree of Master of Technology

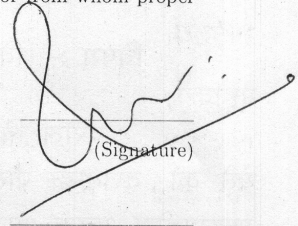


Department of Computer Science & Engineering

June 2018

Declaration

I declare that this written submission represents my ideas in my own words, and where ideas or words of others have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources that have thus not been properly cited, or from whom proper permission has not been taken when needed.



(Signature)

(Abhishek A. Patwardhan)

CS15MTECH11015

(Roll No.)

Approval Sheet

This Thesis entitled Polyhedral Compilation: Applications, Approximations and GPU-specific Optimizations by Abhishek A. Patwardhan is approved for the degree of Master of Technology from IIT Hyderabad

m.v. Pandey

PROF. M. V. P. RAO
() Examiner

Dept. of Computer Science & Engineering
Indian Institute of Technology Hyderabad

Subrahmanyan Kalyanasundaram

PROF. SUBRAHMANYAM KALYANASUNDARAM
() Examiner

Dept. of Computer Science & Engineering
Indian Institute of Technology Hyderabad

U. Ramakrishna

(Dr. Ramakrishna Upadrasta) Adviser
Dept. of Computer Science & Engineering
Indian Institute of Technology Hyderabad

() Co-Adviser

Dept. of Computer Science & Engineering
Indian Institute of Technology Hyderabad

Sparsh Mittal

PROF. SPARSH MITTAL
() Chairman

Dept. of Computer Science & Engineering
Indian Institute of Technology Hyderabad

Acknowledgements

According to me, Its the most important section from everyones thesis – acknowledgment section – where one can express sincere gratitude towards people who are always working for you in the background. I believe acknowledgment section really throws light over all of them, helps rest of the world to understand that the piece is actually a ”team-work” and not individual’s contribution !

First of all, I express my sincere gratitude towards Dr. Ramakrishna Upadrasta – my advisor – without whom, I could not even be at the stage of writing my masters thesis. His positive attitude, kind and down to earth nature, ability to enable his student to work at their maximum, all of these really worked well during my 3 years of journey. He gave complete freedom to pursue research. He was always keen to work on new, challenging problems. His ability to ask research students to get a holistic view, keen love and interest in education are worth appreciating. His research interests are truly aligned with mine and always revolve around The polyhedral model – A giant of giants! I feel myself to be lucky enough to work on (or rather improve upon) a small portion of his Ph.D. thesis. In spite of all this, we shared multiple research ups and downs (law of life !) in my three years of stay, but his positivity, generousness, caring nature literally drove me gently through all of those situations. He always encouraged me to target top-tier conferences irrespective of the final outcome. This attitude really opened my eyes to see how hard it is to even submit a paper in such conferences!. In addition to this, I really thank him for wonderful discussions on spiritual topics including Bhagwat-Geeta, Ganesh-Puran, Ramayana etc. Thank you very much Ramakrishna sir.

Next, I would like to thank Dr. Saurabh Joshi sir for serving on my thesis committee. Dr. Saurabh sir gave critical feedback, appreciated our ideas, suggested improvements (In particular, asked us to prove the soundness of static analysis on modern GPUs having weaker memory models). Dr. Saurabh also encouraged me to only re-submit to top-tier conferences, especially when I was in frustrated mode after multiple rejections. I also thank him for wonderful courses on verification, SAT solver. I thank Dr. Sparsh sir for his insightful as well as broad-coverage course on Advanced Architecture. I also thank Dr. N.R.Aravind sir for his great teaching skills during Advanced Data-Structures course.

I thank Dr. Vineeth sir for allowing me to use their GPU server for experimentation purposes. In addition to this, I must thank many people from VIGIL Lab (especially Debaditya Roy, Dinesh Singh, Adepur Ravi Shankar, Arghya Pal, Srinivas) for their timely help in resolving GPU-server related issues quickly. I was lucky enough to work with many bright B.Tech students: Hrishikesh Vaidya, Akilesh Badrinaarayan, Arasu Arun etc. I thank all of them.

I was fortunate enough to start my journey at IIT-Hyderabad by meeting Prof. Dr. Sanjay Rajopadhye – One of the "inventors" of the Polyhedral Model. Unfortunately, I did not get enough chance to work closely with him, but he visited IIT-Hyderabad twice during my stay. He gave honest feedback which includes: critical comments as well as highly appreciating our new ideas! I also thank Prof. Dr. Albert Cohen to whom I first discussed GPU-cache optimization idea. He motivated me to pursue that track. Essentially, our current work is an outcome from his motivation.

Research in compilers cannot happen unless you have efficient tools already developed, their developers are open to answer your (even) smallest doubts. I thank all the contributors of PLUTO, ISL, PPCG for production-friendly open source tools they have developed over many years. I thank Dr. Uday Reddy Bondhugula, Dr. Sven Verdoolaege, Dr. Michael Kruse, Dr. Riyadh Baghdadi, Dr. Prashant Singh Rawat for taking time to answer my doubts from their busy schedule. Thank you polyhedral compilation community!

Life in IIT cannot keep going well unless you have great friends around you. Great friendship with Shalini, Gayatri, Nandini, Aniruddh, Mukesh bhaiyya, Rasika, Santanu, Venkat Utpal, Tharun will be remembered and continued for a longer time. I also thank all the members of theory lab for making my stay pleasant.

I am grateful to Dr. Pushkar Joglekar Sir (VIT-Pune) for his gentle mentorship for my B.Tech project. His research-level thought process really enabled me to think of pursuing higher studies with a research assistantship. I also thank Ratna Patil Madam for the wonderful undergraduate course on Compilers – The hardest subject for most of the CS students. Her teaching style made me course easy and only because of that, I could pursue masters into the subject. I also thank Dr. Priyadarshan Dhabe sir for teaching CUDA in undergraduate which formed as a foundational basis in my masters project work.

Nothing could have been possible without family support. I thank Aai (Mother), Baba (Father) for their great support, motivation to pursue my studies. I also thank Ajoba (grand-father) for his keen enthusiasm in my studies. I thank Aaji (grand-mother) who taught me Mathematics, Geometry, and Sanskrit in my school days and made my basic concepts clear. I also thank Monika (cousin), Mrs. Vandana Sathe (Aunt) for their immense help towards family during my three years of stay at IIT Hyderabad. I thank my uncles (Mr. Subhash and Mr. Shrinkant Agashe) for timely help in career related decisions.

I thank IIT Hyderabad CSE faculty, CSE-staff, security-staff, mess staff, hostel-staff for their direct and indirect help in making my stay at IIT-Hyderabad good and secure.

I considered myself to be lucky enough to be a part of PATWARDHAN family; having the

tradition of spirituality and astrology. I consider myself to be blessed by teachings from two spiritual guides: (1) Rashichakraakar Sharad Upadhye: Who developed a keen interest in astrology and Dattasampradaya. Interaction with him for three days at Shri Narsinhawadi changed my life drastically. (2) Sadguru Shri Kaka Maharaj: For his spiritual guidance. I also thank Sadguru kaka Maharaj for asking me to pursue higher studies at the point of time, when I was about to give sweets for my undergraduate placement selection. *Pranam* to both of them.

Thanking you ALL very much.

! Dhanyosmi !

! Idam Na Mama !

! Tatsat Brahma-ArpanM-Astou !

ShriGaneshCharanRaj

= Abhishek patwardhan

Dedication

Dedicated to *charanKamal* (Lotus Feet) of:

Shri Sharada-Vignahar Ganesh

Shree Dattatreya

Sadguru Kaka Mahraj

Aai (Mother)

Baba (Father)

Publications based on this Thesis

- Poster: Texturizing PPCG: Supporting Texture Memory in a Polyhedral Compiler
Abhishek Patwardhan and Ramakrishna Upadrasta
IEEE International Conference on High Performance Computing (HiPC), 2016
(Best Poster award)
(Also appeared at NVIDIA-GTCx-Mumbai, 2016)
- Poster: When Polyhedral Optimizations Meet Deep Learning Kernels
Hrshikesh Vaidya, Akilesh B, Abhishek Patwardhan and Ramakrishna Upadrasta
IEEE International Conference on High Performance Computing (HiPC), 2017
(Finalist for Best Poster award)
- Polyhedral Model Guided Automatic GPU Cache Exploitation Framework
Abhishek Patwardhan and Ramakrishna Upadrasta
Under revision.
- Some Efficient Algorithms for Polyhedral Over-approximations problem and it's applications
Abhishek Patwardhan and Ramakrishna Upadrasta
Under preparation.

This page is intentionally left blank.

Abstract

Polyhedral compilation has been successful in analyzing, optimizing, automatically parallelizing affine computations for modern heterogenous target architectures. Many of the tools have been developed to automate the process of program analysis and transformations for affine control parts of programs including widely used open-source and production compilers such as GCC, LLVM, IBM/XL. This thesis makes contribution to the polyhedral model in three orthogonal dimensions as follows:

- **Applications:** Applies polyhedral loop transformations on Deep learning computation kernel to demonstrate the effectiveness of complex loop transformations on these kernels.
- **Approximations:** Develops *two* efficient algorithms to over-approximate convex polyhedra into U-TVPI polyhedra having applications in polyhedral compilation as well as automated program verification.
- **GPU-Specific Optimizations:** Builds end-to-end fully automatic compiler framework to generate cache optimized CUDA code beginning from sequential C program by using polyhedral modelling techniques.

Contents

Acknowledgements	iv
Publications based on this Thesis	viii
Abstract	x
Nomenclature	xii
1 Polyhedral Optimizations for Deep learning kernels	1
1.1 Introduction	2
1.2 Motivation	2
1.3 Polyhedral Compilation	3
1.4 CNN	5
1.5 Max Pooling	5
1.6 RNN	7
1.7 LSTM	8
1.8 PolyBench/NN in Julia	9
1.9 Performance analysis	10
1.10 Conclusions And Future Work	12
2 Exploiting GPU caches by Polyhedral compilation	13
2.1 Introduction	14
2.2 A Motivating Example: LSTM layer	15
2.3 Related Work	18
2.4 Polyhedral Model and PPCG	18
2.5 GPUs: Memory Hierarchy	19
2.5.1 Read-Write incoherency	20
2.6 Automatic Framework for Cache Exploitation	21
2.6.1 Static Analysis	22
2.6.2 Cache selection	24
2.7 Cost Model	24
2.7.1 Cost model for Constant cache	25
2.7.2 Unified Cost Model for Texture/ Surface caches	25
2.8 Code-generation	27
2.9 Performance Evaluation	27
2.9.1 Experimental setup	27
2.9.2 Experimental Results	28

2.10	Real-World Use cases	31
2.11	Conclusions and Future work	33
3	Efficient Algorithms for Polyhedral Over-Approximation	35
3.1	Background	36
3.2	Limitations of state-of-art Mine's algorithm	36
3.3	Algorithm#1: The Farkas lemma based OA algorithm	37
3.3.1	Enabling usage of Farkas lemma	37
3.3.2	Joint search space and Cost function	37
3.3.3	Iteratively finding the pairs of U-TVPI hyperplanes	38
3.4	The Insight	38
3.5	Algorithm#2: Fourier-Motzkin based OA Algorithm	38
3.6	Implementation	39
3.7	Conclusions	39
	References	40

Chapter 1

Polyhedral Optimizations for Deep learning kernels

Deep Neural Networks (DNN) are well understood to be one of the largest consumers of HPC resources and efficiently running their training and inference phases on modern heterogeneous architectures (and accelerators) poses an important challenge for the compilation community. Currently, DNNs are actively being studied by the automatic parallelization and polyhedral compilation communities for the same purpose. In this (initial) paper, we study the kernels of four varieties of DNN layers with the goal of applying automatic parallelization techniques for latest architectures. We show the *affine (Polyhedral) nature* of these kernels thereby showing that they are amenable to well known polyhedral compilation techniques. For benchmarking purposes, we implemented forward and backward kernels for four varieties of layers namely convolutional, pooling, recurrent and long short term memory in `PolyBench/C`, A well known polyhedral benchmarking suite. We also evaluated our kernels on the state-of-art `Pluto` polyhedral compiler in order to highlight the speedups obtained by automatic loop transformations.

1.1 Introduction

Machine Learning (ML) techniques are being extensively used for solving real world problems in various domains. In applications from Computer Vision and Natural Language Processing (NLP), Neural Network (NN) models are trained in order to learn a pattern, after which the model can be used for an unseen input. Due to extensive usage of high resolution graphics and large textual datasets, the real-world HPC requirements of DNNs is quite large. This makes the application of compiler optimizations, parallelization (and tuning) strategies to the training and inference phase as a vital key to effectively parallelize and optimize these computations.

In this paper, we discuss the issues that arise from a *per-layer* implementation of the main classes of DNNs as a *specific variety* of (affine) Polyhedral loop programs. Our implementation focuses on following the PolyBench/C [1] structure, a widely used benchmark for Polyhedral compilation tools. While two of DNN implementations are perfectly polyhedral codes, presence of *stride parameters* makes the remaining two non-polyhedral. We describe a practical way in which they can be turned into polyhedral programs. As a proof of concept of our implementation, as well as to show the potential of polyhedral compilation framework on DNNs, we optimize our kernels using a well known polyhedral compiler Pluto [2], to study the speedups obtained by applying a complex sequence of loop transformations.

We see our work as the first step in automatically generating accelerator specific kernels using advanced polyhedral compilation techniques. The larger goal of this study is aimed at applying polyhedral techniques to automatically generate accelerator specific efficient programs on various architectures.

1.2 Motivation

Artificial Neural Networks (ANNs) are biologically inspired from interactions of neurons in the human brain. ANNs consist of several layers with nodes in each layer obtaining input from nodes in the previous layer via interconnections between layers. The activation of a node is determined by the input values and the weights on the connections between the inputs and the nodes. The training phase involves updating the weights on interconnections so that expected results are obtained at the output layer. The forward pass involves executing the network on new training inputs, while backward pass updates the weights to reduce the deviation from the expected output. The training data is partitioned into chunks where each chunk is referred as a mini-batch. The input mini-batch is presented to the network and is forward propagated through the layers to obtain an output at the final layer. Error is computed between the obtained output and the desired output with respect to a certain loss function (square loss, cross-entropy loss etc.). During backward pass, the weight values of every layer get adjusted so as to be able to predict correct labels of input samples. After all the mini-batches are presented, the process starts over again. Deep NNs have been successful at solving many tasks of utmost importance. NN models are implemented as programs that perform various array operations (for instance multiplication, reductions etc.) possibly lying within a deep loop nest, and therefore are best candidates for polyhedral loop transformations. Deep NNs typically comprise of a large number of layers, and their design is crucial to improve the accuracy of the network.

We now briefly discuss some widely used classes of deep NNs and state their broad application

domain. Convolutional Neural Networks (CNNs) are a class of NNs widely employed for image and video data. In the recent past, CNNs have achieved tremendous improvement in accuracies for several computer vision tasks [3, 4]. Convolutional network consists of one or more (convolutional) layers often accompanied with a subsampling layer and fully connected layers; the convolutional layers account for roughly 80% of the computation time. Pooling layer is a type of layer within a deep CNN, which summarizes the input presented to it. Since CNNs are compute intensive, pooling helps to compress the data as it flows through the deep net. Recurrent Neural Networks (RNNs) have become a de facto for modeling sequential dependencies in discrete time series, useful in context driven tasks (NLPs). Long Short Term Memory (LSTM) network is another variant of RNN specialized for improving accuracy during learning phase.

Deep learning workloads are computationally intensive and manually optimizing their kernels on a variety of modern parallel architectures (and accelerators) is a challenging task. In this paper, we try to explore potential of automatic parallelization for deep learning kernels.

Further we observe that deep learning kernels involve deep loop nest and hence polyhedral optimizations

Algorithm 1 Convolutional layer: Forward pass

Require: $N, K, C, H, W, R, S, U, V$

Require: $\text{inp}[N][C][H][W]$: Input data

Require: $W[K][C][R][S]$: Weight matrix for a given layer

1: $P \leftarrow (H - R)/U + 1$

2: $Q \leftarrow (W - S)/V + 1$

3: $\forall (n \in N, k \in K, p \in P, q \in Q, c \in C, r \in R, s \in S) \text{ do } \{$

4: $\text{out}[n][k][p][q] += W[k][c][r][s] * \text{inp}[n][c][u*p+R-r-1][u*q+S-s-1]$

5: $\}$

The rest of this paper is organized as follows: In Section 1.3, we give a quick overview of polyhedral compilation. Then, in Sections 1.4–1.7, we present the computational kernels corresponding to forward and backward phases of various deep NNs. In Sec. 1.9, we discuss the performance improvements obtained by loop transformations. Finally, in Sec. 1.10, we state conclusions and some future work.

1.3 Polyhedral Compilation

The Polyhedral model focuses on optimizing and parallelizing the loop nests. It is a powerful formalism to analyze and transform the input affine programs so as to run them on varieties of modern heterogeneous architectures. It can statically analyze programs which involve affine loop bounds and affine array access functions. Typically, a polyhedral compiler first creates a model of input loop nest. A statement nested within a d deep loop nest is represented as d -dimensional polyhedron where each integral point represents dynamic instance of that statement. After extracting such a representation, data dependence analysis—a well studied problem that boils down to solving an integer linear programming problem [5]—is performed. This analysis finds the (dynamic) instances of two possibly different statements which access the same array location, and at least one of the accesses is a write. Such an analysis is important to preserve the semantics of original program. The second step of polyhedral compilation is the affine scheduling problem [6], that involves finding a

Deep Neural Network layers	BLAS / HPC kernels
Convolutional layer	Stencils, tensor multiplication
Recurrent layer	Stencil with varying time steps
LSTM	Set of Matrix vector products
Max, sum Pooling	Max/Sum reductions

Table 1.1: Correspondance among DNN layers and HPC kernels

Table 1.2: Program Parameters for CNN

N	Number of Input Images in batch
C	Number of Input feature maps
K	Number of Output feature maps
$P \times Q$	Size of output feature map
$R \times S$	Size of filter kernel
U,V	Stride parameters

complex sequence of classical loop transformations (such as loop tiling, permutation, skewing) which expose the parallelism and improve the data locality. A number of approaches exist to find a good program transformations from a large search space, and one practical approach involves scheduling two dependent statement instances as much closer (in time space) as possible. The above approach was first implemented in the Pluto source to source compiler [2] which we use in this work. The final step of polyhedral compilation involves generating a loop nest which scans all valid integer points in polyhedra [7]. The parallel loops is be marked with apt pragmas (like OpenMP, OpenACC) during code-generation.

In recent past, polyhedral compilation has shown to be effective in accelerating various linear algebra kernels, tensor contractions, stencils, image processing applications etc. We make the *crucial observation* that many of the layers used in deep learning pipelines perform computations that are similar to the ones polyhedral compilation has been successful in optimizing. It is known that entries in the column 2 of above table are well optimized by polyhedral compilers. In this paper, we try to explore how polyhedral model optimizes various deep learning layers given the close correspondence as depicted in table 1.1. We also release the NN kernels. Earlier researchers who worked only on CNN [8] did not release their code (Neither HDL nor HLS/C code) to open-source. To the best of our knowledge, there is no known implementation of DNNs (**as (affine) Polyhedral programs**) for benchmarking purposes. With this paper we overcome the above limitation.

```

for (n = 0; n < _PB_NN; n++)
  for (k = 0; k < _PB_NK; k++)
    for (p = 0; p < _PB_NP; p++)
      for (q = 0; q < _PB_NQ; q++)
        for (c = 0; c < _PB_NC; c++)
          for (r = 0; r < _PB_NR; r++)
            for (s = 0; s < _PB_NS; s++)
              out_F[n][k][p][q] += W[k][c][r][s] *
                inp_F[n][c][u*p+NR-r-1][u*q+NS-s-1];

```

Figure 1.1: C Code : CNN forward pass

Table 1.3: Program Parameters for Pooling

N	Number of Input images
D	Number of feature maps
(IH,IW)	Size of input feature map
(OH,OW)	Size of output feature map
(DH,DW)	Size of Pooling kernel
(SH,SW)	Horizontal and vertical stride values

1.4 CNN

The program parameters of a CNN are described in Table 1.2. For parallelizing purposes, the CNN program can be thought of as a stencil (with uniform dependences) defined over a loop nest of depth seven, with the loop body computing convolution. A quick study of the dependences of the code shows that all four outer dimensions, namely n, k, p, q , are completely parallel. The array index expression for *inp* array accesses the appropriate location in the input feature map accounting for inverting and striding. Though our current implementation assumes absence of padding in the input filters, though it can be added at a later time. The array access is clearly non-affine (due to the

```

for(n = 0; n < _PB_NN; n++)
  for (c = 0; c < _PB_NC; c++)
    for (h = 0; h < _PB_NH; h++)
      for (w = 0; w < _PB_NW; w++)
        for (k = 0; k < _PB_NK; k++)
          for (r = 0; r < _PB_NR; r++)
            for (s = 0; s < _PB_NS; s++)
              for (p = 0; p < _PB_NP; p++)
                for (q = 0; q < _PB_NQ; q++)
                  if((u*p - (h - NR + r + 1) == 0) && (u*q - (w - NS + s + 1) == 0))
                    err_in[n][c][h][w] += W[k][c][r][s] * err_out[n][k][p][q];

```

Figure 1.2: C Code : CNN backward pass

multiplication of the stride parameter with the corresponding indices). The reader is referred to Section 1.5 for a note on affinity of CNN and MaxPool. In the backward pass, the error information is propagated from output of a layer to its input. *err_out* contains the error derivative with respect to the output of the layer. To compute the error derivative with respect to the input, *err_out* is multiplied with values from weight matrix W to accumulate values into *err_in* matrix.

1.5 Max Pooling

Pooling is a form of layer usually added after convolutional layer in CNN to reduce the spatial size of the representation in the network. The program parameters for max pooling operation are provided in Table 1.3. In MaxPooling, the maximum input value within the window is termed as the output of the operation as shown in Algo. 2. Only the maximum value of input window contributes to the output value. During backpropagation phase (shown in Algo. 3), the error derivative with respect to output is added only to the input pixels which have contributed to the output value.

Affinity of CNN and MaxPool: A central operation in CNN is *convolution* which accesses the array index expression by multiplying the *stride parameter* with the loop dimension to get the required offset in the input image. Though this makes the array access function non-affine, as these

Algorithm 2 Max pooling layer: Forward pass

Require: $N, D, IH, IW, DH, DW, SH, SW, \text{inp}[N][D][IH][IW]$: Input

```
1:  $OH \leftarrow (IH - DH) / SH + 1$ 
2:  $OW \leftarrow (IW - DW) / SW + 1$ 
3:  $\forall (n \in N, d \in D, r \in OH, c \in OW)$  do {
4:    $\text{val} \leftarrow \text{MIN\_INT}$  foreach  $h \in [SH * r, \min(SH * r + dh, ih))$  do
       end
        $w \in [SW * c, \min(SW * c + dw, iw))$ 
5:    $\text{val} \leftarrow \text{MAX}(\text{val}, \text{inp}[n][d][h][w])$ 
6:
7:
8:    $\text{out}[n][d][r][c] \leftarrow \text{val}$ 
9: }
```

Algorithm 3 Max pooling layer: Backward pass

Require: $N, D, IH, IW, DH, DW, SH, SW$

Require: $\text{inp}[N][D][IH][IW], \text{err_out}[N][D][OH][OW]$: Input data

```
1:  $OH \leftarrow (IH - DH) / SH + 1$ 
2:  $OW \leftarrow (IW - DW) / SW + 1$ 
3:  $\forall (n \in N, d \in D, r \in OH, c \in OW)$  do { foreach  $h \in [SH * r, \min(SH * r + dh, ih))$  do
       end
        $w \in [SW * c, \min(SW * c + dw, iw))$  if  $\text{out}[n][d][r][c] == \text{inp}[n][d][h][w]$  then
4:
       end
        $\text{err\_in}[n][d][h][w] += \text{err\_out}[n][d][r][c]$ 
5:
6:
7:
8: }
```

Table 1.4: Program Parameters for RNN

T	Number of time steps
P	Size of input vector
Q	Size of output vector
S	Size of hidden vector
BPTT	Truncated Unroll factor

stride parameters are constant integer literals for each individual layer within a DNN, and are fixed while designing the network, we fix them statically. A similar strategy was used by Zang et al. [8] who used Polyhedral techniques for FPGA code generation. The same argument applies MaxPool layer as well.

1.6 RNN

The unique aspect of RNNs is the feedback loop where the output of the neuron is passed as input to the same neuron leading to a recurrence in time dimension. The presence of feedback loop introduces a set of dependences during both forward and backward phases of the network. The layer has three weight matrices namely U, V, W which are *learnt* during back-propagation phase. During back-propagation, the error of output neuron is propagated T steps back in time. The kernel parameters for a typical RNN is shown in Table 1.4.

Algorithm 4 RNN layer: Forward pass

Require: BT, T, P, Q, S
Require: $U[S][P]$, $W[S][S]$, $V[Q][S]$: Weight matrices
Require: $state(t)$, $input(t)$: Vector of size S, P resp.
1: $state(0) \leftarrow U * input(0)$
2: $output(0) \leftarrow V * state(0)$ **foreach** $t \in [1, T]$ **do**
3: **end**
 $state(t) \leftarrow U * input(t) + W * state(t-1)$
4: $output(t) \leftarrow V * state(t)$
5:
6:

As described in Algo. 4, in the forward pass of RNN, $state(t)$ denotes hidden state vector at each time step and similarly $output(t)$ is the output at timestep t . The hidden state ($state(t)$) computation at time step t uses information of current input vector and hidden state vector of previous time step. U and W are multiplied with $input(t)$ and $state(t - 1)$ respectively and the quantities are added to get the final result. The output vector is obtained by computing an inner product of V and current hidden state i.e. $state(t)$.

We describe the backward pass in Algo. 5, where the error derivatives are summed up for each time step t . This computes the error accumulation of gradient using chain rule. The err_S^B acts as an intermediate vector during the back-propagation step to store the error derivative with respect to the hidden state vector $state(t)$ represented as err_S^A in the Algorithm.

Algorithm 5 RNN layer: Backward pass

Require: BT, T, P, Q, S, (U[S][P], W[S][S], V[Q][S]): Weights

Require: $err_{out}(t)$, state(t), input(t): Vector of size Q, S, P resp. **foreach** $t \in [T - 1, 1]$ **do**

 ℓ:
 end
 $err_V+ = err_{out}(t) * state(t)$
 2: $err_S^A[1 : r] = V * err_{out}(t)$ **foreach** $step \in [t + 1, max(0, t - BT)]$ **do**
 B:
 end
 if $step > 0$ $err_W+ = err_S^A[1 : r] * state(step - 1)$
 4: $err_U+ = err_S^A[1 : r] * input(step)$
 5: $err_S^B+ = err_S^A[1 : r] * W$
 6: $err_S^A[1 : r] = err_S^B[1 : r]$
 7:
 8: =0

Table 1.5: Program Parameters for LSTM

T	Number of time steps
P	Size of input vector
Q	Size of output vector
S	Size of hidden vector

1.7 LSTM

LSTM is a special kind of RNN, designed to combat vanishing gradients [9] through a gating mechanism. A typical LSTM layer is comprised of a forget gate, an input gate and an output gate. Each gate masks some information (from the stream of data flowing through the network) propagating through itself, or its previous layers depending on the type of gate. The parameters required to describe LSTM are given in Table 1.5.

In Algo. 6, $input_{gate}$, $forget_{gate}$, $output_{gate}$ represent the input, forget and output gates respectively, and work like masks. The $input_{gate}$ decides to what extent the current input contributes to the newly computed state $memory(t)$. The $forget_{gate}$, defines the factor of the previous state which is retained in the current state. Finally, the $output_{gate}$, defines how much of the internal state is exposed to the external network (that is, to the subsequent layers and to the next time step as well).

Algorithm 6 LSTM Neural Network layer: Forward pass

Require: T, P, Q, S, (input(t), state(t)): Vectors of size P, S resp.

Require: $W_i[S][S]$, $W_f[S][S]$, $W_o[S][S]$, $W_g[S][S]$: Weight matrices for hidden state. Suffix represent gate type (input, forget, output, hidden)

Require: $U_i[S][P]$, $U_f[S][P]$, $U_o[S][P]$, $U_g[S][P]$: Weight matrices for input state. **foreach** $t \in [1, T)$ **do**

 ℓ:
 end
 $input_{gate}[1:S] \leftarrow input(t) * U_i + state(t-1) * W_i$
 2: $forget_{gate}[1:S] \leftarrow input(t) * U_f + state(t-1) * W_f$
 3: $output_{gate}[1:S] \leftarrow input(t) * U_o + state(t-1) * W_o$
 4: $cand_{state}[1:S] \leftarrow input(t) * U_g + state(t-1) * W_g$
 5: $memory(t) \leftarrow memory(t-1) * forget_{gate} + cand_{state} * input(t)$
 6: $state(t) \leftarrow memory(t) * output_{gate}$
 7:

$cand_{state}$ is a *candidate* hidden state that is computed based on the current input and the previous

hidden states. The method of computing $cand_{state}$ is the same as that of computing $state(t)$ in a RNN, except that the parameters U and W are replaced with U_g and W_g . $memory(t)$ can be considered as internal memory of the unit, which is a sum of two components: a) $memory(t - 1)$ multiplied by the forget gate $forget_{gate}$, b) newly computed candidate hidden state $cand_{state}$ multiplied by the input gate $input_{gate}$. In other words, it is a combination of how we want to combine the new input with previous memory. Given the $memory(t)$, the output $state(t)$ is computed by multiplying the $memory(t)$ with the output gate $output_{gate}$.

The back-propagation phase for LSTM(Alg. 7) consists of computing errors for vectors representing various gates(input/output/forget/cand.state). Using these, errors for weight matrices are computed. Notice that, while computing errors for U_i, U_g, U_f, U_o , $input(t)$ gets multiplied with the error values for a gate. While, error computation of W_i, W_g, W_f, W_o requires $state(t)$. This is so because during forward phase U_i, U_f, U_o, U_g represents weight matrices for $input(t)$ and W_i, W_f, W_o, W_g represents weight matrices for candidate hidden state.

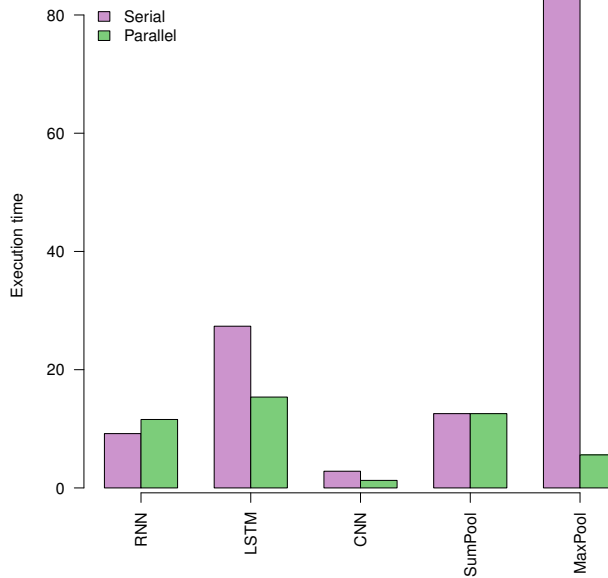


Figure 1.3: Execution times for forward pass

1.8 PolyBench/NN in Julia

Julia [10] is a high-level language suited for scientific applications. Julia code gets translated into LLVM-IR through its JIT compiler so as to facilitate varieties of compiler optimizations implemented in LLVM. Recently during Google Summer of Code-2016, polyhedral transformations were enabled into Julia via LLVM-Polly [11]. Hence, we also ported our PolyBench/NN programs into PolyBench.jl framework [12].

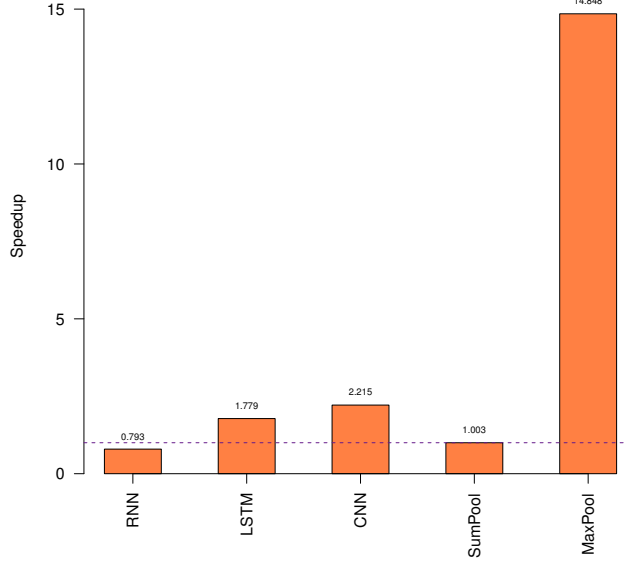


Figure 1.4: Execution times for backward pass

1.9 Performance analysis

To study speedups obtained by applying polyhedral transformations, we use Pluto [2] (version 0.11.4), a widely used source-to-source polyhedral optimizer with `--tile --parallel` flags. All experiments were performed with the data set sizes set to the PolyBench variable `EXTRA-LARGE`. We compiled the parallel codes generated by Pluto using GCC-7.0.0, and OpenMP-4.5 for execution. The experiments were performed on Intel(R) Xeon(R) CPU E5-2630 v3@2.40GHz cluster having two processors with each processor having 8 hardware cores. We ran each program three times by using benchmarking script bundled within PolyBench, which internally runs it five times. We selected median of three trials as the execution time. We separately recorded the execution times for serial and parallel versions for both forward and backward phases.

Algorithm 7 LSTM Backward pass

Require: $T, P, Q, S, W_i[S][S], W_f[S][S], W_o[S][S], W_g[S][S], U_i[S][P], U_f[S][P], U_o[S][P], U_g[S][P],$
 $output_gate[1:S], input_gate[1:S], forget_gate[1:S], cand_state[1:S]$

Require: $memory(t), input(t)$ Vector of size S, P resp **foreach** $t \in [T - 1, 1)$ **do**

```

l:
    end
     $err_{output}^g = memory(t) * err_{state}(t)$ 
2:  $err_{memory}(t) += output\_gate[1:S] * err_{state}(t)$ 
3:  $err_{forget}^g = memory(t-1) * err_{memory}(t)$ 
4:  $err_{memory}(t-1) += forget\_gate[1:S] * err_{memory}(t)$ 
5:  $err_{input}^g = cand\_state[1:S] * err_{memory}(t)$ 
6:  $err_{cand\_state}^g = input\_gate[1:S] * err_{memory}(t)$ 
7:  $err_{U_i/g/f/o} += input(t) * (err_{input/cand\_state/forget/output}^g)$ 
8:  $err_{W_i/g/f/o} += state(t) * (err_{input/cand\_state/forget/output}^g)$ 
9:  $err_{state}(t-1) += W_i * err_{input}^g + W_f * err_{forget}^g + W_o * err_{output}^g + W_g * err_{cand\_state}^g$ 
10:

```

11: \triangleright Note: Line 8,9 defines 4 statements, with one to one correspondence between LHS and RHS alternatives

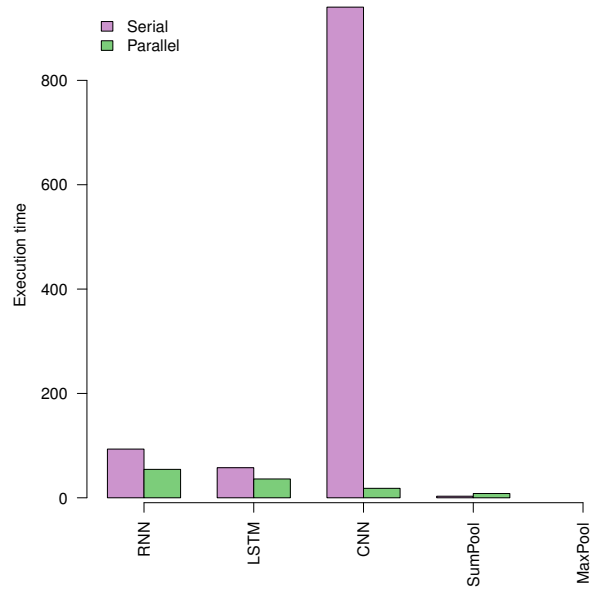


Figure 1.5: Execution times for backward pass

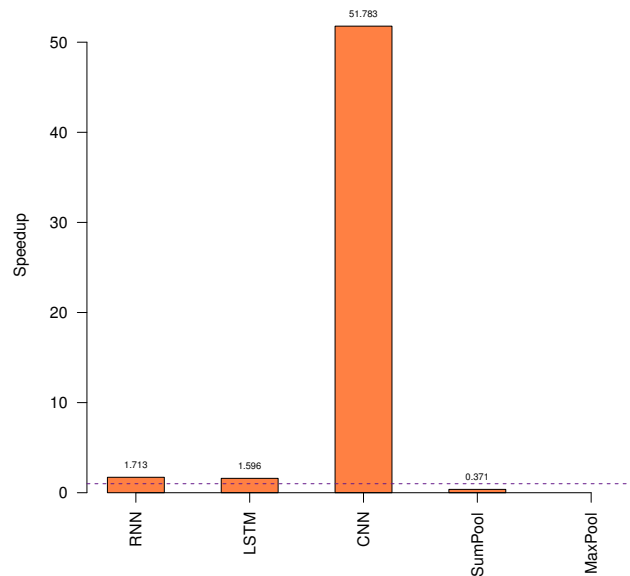


Figure 1.6: Execution times for backward pass

The plots showing execution times for forward and backward phases are given in 1.3 and 1.6 respectively. We make following observations from plots: 1) Backward phase is more compute intensive than forward phase for all layers except SumPooling. 2) RNN, LSTM consist of Polyhedral loops that are successfully parallelized by Pluto. 3) For CNN and Maxpool, we were forced to replace the stride parameters with integer constants defined in our header file. This made CNN and Maxpool (forward pass) analyzable for Pluto. 4) The backward phase of MaxPooling kernel

Type	Forward	Backward
CNN	2.21x	51.78x
RNN	0.79x	1.71x
LSTM	1.77x	1.59x
MaxPool	14.84x	NA

Table 1.6: Speed-up over serial execution

consists of a data dependent condition which Pluto's dependence analysis is unable to analyze. 5) No speedups were observed for forward phase of RNN and backward phase of SumPooling. 6) Average speedups observed for forward and backward phases are 2.15 and 2.69 respectively.

1.10 Conclusions And Future Work

We implemented the four varieties of neural network layers as loop-programs in the PolyBench framework. While RNN and LSTM strictly adhere to Polyhedral framework's affinity conditions, CNN and MaxPool do not and we had to fix the stride parameters of their codes manually. The programs show *significant speedups* after applying polyhedral transformations. We released our PolyBench/NN C implementation <https://github.com/hrishikeshv/polybench/tree/master/polyNN> so that other researchers can work on advanced optimizations on these kernels. Though our work is *preliminary*, we believe it will form a basis to apply automatic loop transformations that expose parallelization as well as data locality optimization opportunities for different DNN architectures on various heterogeneous architectures and accelerators.

Chapter 2

Exploiting GPU caches by Polyhedral compilation

We propose a compiler driven method by which parallel computations can be accelerated on GPUs by exploiting the various special varieties of caches (such as texture, surface and constant for NVIDIA GPUs) available on them. We show that our method obtains superior performance, when compared with earlier methods of accelerating these classes of computations that use on-chip shared memory. We provide an end-to-end solution by developing a *fully automatic* framework within a state-of-art source-to-source Polyhedral compiler (PPCG) to exploit these varieties of GPU caches. Using techniques from the formalism that Polyhedral model provides, we reason about the profitability of using each of the particular variety of GPU caches. We evaluate our implementation on kernels from PolyBench/C benchmark and report *up to 1.5x* speedups over the existing memory mapping strategy used by PPCG compiler. As the usage of particular variety of cache is highly application specific, we also consider five representative computations namely: PageRank, Recurrent Neural Network, Long Short Term Memory layer, Poisson Solver and DWE-FDTD stencil and show that usage of special GPU caches in these programs results in up to 2.6x speedup over a standard shared memory based implementation. With these use cases, we show the general purpose computing usage of these special GPU caches that were originally designed for image processing applications. With increasing interest in mapping general purpose algorithms on GPUs, we believe that our contribution is towards automatic exploitation of GPU cache/memory hierarchy.

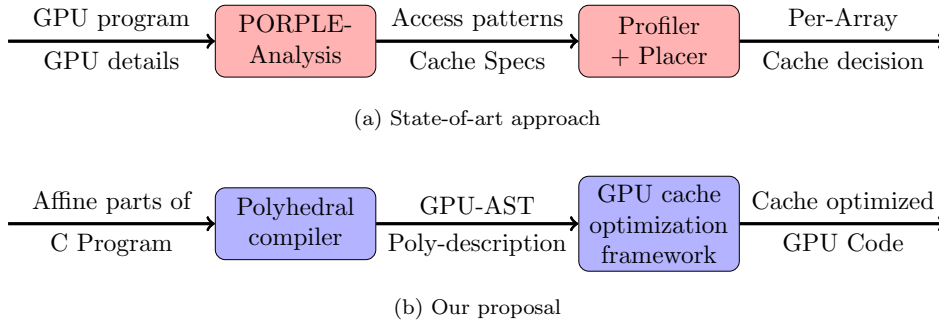


Figure 2.1: Profiler based approaches employ less-precise analytical models to guide cache decisions. We use polyhedral representation to precisely model profitability criterion as specified by GPU vendors thereby resulting into end-to-end fully automatic framework.

2.1 Introduction

The advancements made from the traditional processor architecture to multi-core, many-core machines after the collapse of Moore's law are greatly improving the execution times of various applications. Also, special hardware accelerators like GPUs are being used to speedup highly parallel SIMD programs. However, it is well understood that correctly programming such parallel accelerators is not only difficult but also highly error prone. Ever since the beginning of compilers, optimizing compilers have long been shown themselves to be effective in making programmers free of the challenges involved in writing correct and efficient parallel programs. The primary motivation for many of the optimizing compilers is to design new optimization strategies so as to expose and exploit the parallelism present in the input program.

Such an approach of auto-parallelization has multiple benefits. Firstly, by relying on static/dynamic analysis implemented within the compiler, the correctness of the transformed program can be guaranteed, as long as the compiler is free of bugs. Secondly, supporting upcoming architectures reduces to the problem of writing a new backend, which is a well understood engineering challenge. Finally, and most importantly, with extra target-specific information, the compiler can exploit the available hardware resources to the maximum possible extent.

It is this last benefit that our work focuses on: by performing rigorous static and dynamic analyses, optimizing compilers can generate efficient parallel/vector programs that can exploit the complete power of multi/many core machines along with their hardware intricacies.

Polyhedral model is a powerful formalism to automatically parallelize the affine programs. In recent past, using polyhedral model, many tools have been developed to optimize programs for CPUs and various accelerators. PPCG [13] is one such state-of-art polyhedral compiler that analyzes affine parts of programs written in C, and automatically generates CUDA/OpenCL code, while incorporating several GPU specific transformations.

Parallelizing and optimizing compilers employ a variety of affine transformations to effectively utilize the memory hierarchy with multiple level of caches. For CPUs, such transformations result in effective utilization of on-chip L1, L2 and shared L3 level caches. In case of GPUs, the situation is a little different; large number of cores share common GPU global memory. As GPU global memory has much higher access latency [14], effective utilization of global memory bandwidth by reducing the global memory accesses is critical to accelerate computations on GPUs. Thanks to architecture level advancements, GPU architectures support a variety of software managed caches. These include

shared memory (software managed portion of L1 cache), *constant* cache, *texture* cache and *surface* cache (Table 2.2).¹ Beyond software managed caches, GPU architectures also support large register files, and fast on-chip L1 and L2 level caches.

With this work, we make a case that, for real world computations that can be mapped to GPUs, exploitation of shared memory incurs extra overhead. This can be avoided by effectively utilizing special varieties of GPU caches, namely, surface/texture/constant memory in NVIDIA GPUs and image/constant memory in AMD GPUs.² Thereby, we show the *general purpose computing usage of these special GPU caches that were originally designed for image processing applications*.

2.2 A Motivating Example: LSTM layer

For real-world computations in general, GPU shared memory may not be a good choice to access arrays. To substantiate this claim, consider the computational kernel shown in Fig. 2.2. This kernel depicts the working of forget gate included in Long short term Memory layer. LSTM layer is widely used by deep learning community for various learning tasks in Natural language processing (NLP). Even though we show only one portion of the LSTM code (representing forget gate only), discussion is applicable to other types of gates involved in LSTM layer (such as input gate, output gate). In LSTM, the outermost (time) loop captures the context information. At every time step, the kernel computes the vector \mathbf{f} by simply taking weighted combination of (i) input at current time step $\text{inp_F}[t]$, and (ii) context provided by previous time step ($\text{s_F}[t-1][\dots]$). In the above Figure, the U_f , W_f arrays correspond to the weight matrices learned by the LSTM layer during training phase. Finally, vector \mathbf{f} depicts the output for the forget gate of LSTM layer at given time step t .

At each time step, the kernel simply performs sequence of array-multiplication followed by a *reduction* operation with sum as operator. It can however be noticed that the above computation uses a large number of arrays. Thanks to the affine nature of the above computation (affine array access functions, affine loop bounds), GPU code can be generated for it by using the PPCG compiler. PPCG employs a variety of analyses and transformations on the input program to effectively manage data in the register files, perform array privatization, exploit shared memory by performing array reuse analysis, and optimize the data transfers from CPU to GPU global memory. The relevant sections of the output produced by PPCG is shown in Figure 2.2.

Table 2.1: Execution times of PPCG generated CUDA code for LSTM layer on NVIDIA GPUs (Problem size: T=400, S=2850, P=3000)

	PPCG:Global	PPCG:Shared	PPCG:Caches
Tesla K20X	9.504 sec	3.882 sec	1.785 sec
Tesla P100	2.434 sec	1.968 sec	1.429 sec

In the transformed program, the GPU kernel `kernel1` is invoked for each iteration of a sequential loop (with iterator `c0`) which essentially executes on the host (CPU). Also, the GPU kernel uses low-latency shared (also termed as scratchpad) memory and therefore, achieves good amount of

¹GPU programming literature refers texture/surface/constant as memories rather caches. This is because, as per GPU programming APIs, these are treated as objects which encapsulates the program data (arrays) that are supposed to be accessed through special varieties of caches. The special caching techniques allow to access these objects effectively.

²Though our framework equally applies to special caches available on both NVIDIA and AMD GPUs, our code generator currently supports only CUDA. So, our discussion is mostly with respect to NVIDIA GPU caches.

Figure 2.2: LSTM: Original C-code fragment and PPCG Generated CUDA Code

```

1  for (t = 0; t < _PB_T; t++)
2  for (s1 = 0; s1 < _PB_S; s1++) {
3      f[s1] = 0.0;
4      for (p = 0; p < _PB_P; p++)
5          f[s1] += U_f[s1][p] * inp_F[t][p];
6      if (t > 0)
7          for (s2 = 0; s2 < _PB_S; s2++)
8              f[s1] += W_f[s1][s2] * s_F[t-1][s2];
9  }
```

```

1  --global-- void kernel( ... ){
2      ...
3      shared_f[...] = f[...];
4      shared_U_f[...] = U_f[...];
5          shared_inp_F[...] = inp_F[...];
6      --syncthreads();
7      for (int c4 = 0; c4 <= ppcg_min(31, c2 + 2999); c4 += 1)
8          shared_f[t0] += (shared_U_f[t0][c4] * shared_inp_F[0][c4]);
9          --syncthreads();
10     f[...] = shared_f[...];
11     }
12 }
13 --host-- void LSTM_layer(...) {
14     ...
15     // Copy data from CPU to GPU
16     for (int c0 = 0; c0 <= 400; c0 += 1){
17         dim3 dimBlock(32), dimGrid(63);
18         kernel <<<dimGrid, dimBlock>>> (...);
19         //synchronize and launch other kernels
20     }
21     // Launch next kernel
22     ...
23 }
```

speedup over the original naïve global memory based implementation. But, the following *crucial* observations can be made on these programs:

1. Arrays `U_f`, `s_F`, `W_f`, `inp_F` are read-only; both in the original, as well as the transformed program.
2. Before starting computations within a GPU kernel, small portion of arrays need to be copied from global memory to the shared memory. Note that this need to happen for each iteration of the outer-loop executing on host `c0`.
3. All the threads within a `cudaBlock` must synchronize once the data has been loaded to shared memory. This contributes to extra latency; again, this is incurred for each iteration of outer-loop `c0`.

So, in the above computation, read-only texture cache³ is an apt choice for accessing arrays `U_f`, `W_f`. Using texture cache eliminates each of the latencies mentioned in observations 2 and 3, and therefore results in superior performance. All these observations create a motivation for having a unified automatic framework to support the entire GPU cache/memory hierarchy.

Contributions: We make the following contributions:

- We show theoretically as well as empirically the limitations of usage of shared memory on real-world kernels of iterative nature. As a solution, we design a compile-time, fully automatic GPU cache exploitation framework in the state-of-art source-to-source polyhedral compiler. Our framework supports *all GPU caches*, including surface caches, an architectural advancement of texture caches.
- We design a sound (on weaker memory models) static analysis to ensure program correctness after transformation. We formalize the profitability guidelines provided by GPU-vendors (like NVIDIA) for varieties of caches using integer set arithmetic and integer linear programming.
- On the PolyBench/C benchmarks we report *up to 1.5x speedups over existing PPCG*. As the speedups after cache exploitation vary from program to program, we show that our framework can accelerate some additional (representative) *real-world iterative computations*: PageRank, RNN, LSTM, PoissonSolver and FDTD kernels. On these important programs from different domains, we show up to 2.6x performance improvements over auto-generated codes (using PPCG) that use *shared memory*, and up to 1.27x over manually annotated codes (using PGI). We thereby show that GPU special caches could be effectively used in domains beyond than what they were originally designed for.
- Considering that PPCG already exploits local, shared, and global memory effectively, with our new framework added into PPCG, it is now able to support the **complete GPU cache and memory hierarchy automatically**. To the best of our knowledge, **none** of the existing (fully/semi) automatic optimizing compilers support code-generation for the complete GPU cache/memory hierarchy.

³Even constant cache could be used, in case the array size is small.

The remaining part of this section is organized as follows: First, in Sec. 2.3, we summarize the related work. In Sec. 2.4, we introduce polyhedral model and the PPCG compiler. In Sec. 2.5, we discuss GPU memory hierarchy with a focus on special caches. In Sec. 2.6, we discuss static analysis and cache decision algorithms. In Sec. 2.7, we describe the cost model, while in Sec. 2.8, we discuss code-generation. In Sec. 2.9, we describe the evaluation results on PolyBench/C suite. In Sec. 2.10, we show how our framework accelerates some real-world kernels (PageRank, RNN, LSTM, PoissonSolver, FDTD) and finally in Sec. 2.11, we state conclusions and future work.

2.3 Related Work

There has been a significant work on automatic and semi-automatic compilation strategies to map general purpose and affine programs on GPUs. The polyhedral model has been successful in effectively parallelizing the sequential programs for GPUs. One such first experimental prototype tool namely CToCUDA-C was by Baskaran et al. [15, 16] and uses PLUTO compiler to obtain program transformations; while it claims to support constant memory, we were unable to obtain constant memory optimized CUDA code for some simple kernels. The tool does not support texture caches. (We used pluto-0.6.2-cuda [17].)

LLVM/Polly-ACC [18] supports PTX code generation from LLVM/Polly [19], but lacks support for GPU caches. R-Stream [20] a proprietary compiler from Reservoir Labs that uses polyhedral framework to map SCoPs to GPUs. Due to its unavailability, we could not use it for any comparison; their paper [20] does not discuss about GPU caches.

While our framework is *fully-automatic*, there are many semi-automatic approaches where the user can specify the affine transformations to the compiler to generate parallel code. CUDA-Chill [21] is an example; it lacks support for GPU caches. PGI compiler from Portland group and NVIDIA accepts C/Fortran programs with OpenACC annotations for loops and array-annotations (only for texture/constant and not surface caches) that are interpreted by the compiler to generate PTX code. Even among PGI compilers, texture/constant cache support seems to be limited to the PGI CUDA Fortran compiler [22].

Another popular semi-automatic tool OpenMPC [23] is developed on the top of Cetus infrastructure which extends OpenMP so that the user can provide annotations. Like in the PGI compiler, OpenMPC allows annotations to access arrays, but that too only from texture caches.

2.4 Polyhedral Model and PPCG

The Polyhedral model works on compute-intensive parts of a program and hence targets loop nests. The specific variety of loops that Polyhedral Model operates on are called Static-Control Parts of program (SCoP). Essentially, SCoPs are statically analyzable because of usage of affine-expressions for loop bounds, conditionals, array accesses. Polyhedral model represents statements nested within a d -deep loop nest as a d -dimensional polyhedron in the Euclidean space, henceforth referenced as iteration domain. A valid integer point inside the iteration domain corresponds to a dynamic instance of that statement. In the polyhedral model, array data-flow analysis [5, 24] is performed to detect two dynamic instances of statements, both of which access the same array element, and at least one of them is write. In order to preserve program semantics, dependences are used to obtain

Table 2.2: GPU cache varieties

GPU vendor	Variety of Cache	
	Optimized for spatial access	Optimized for broadcast access
NVIDIA	Texture (Read only) Surface (Read + Write)	Constant (Read only)
AMD	Image (Read + Write)	Constant (Read only)

constraints on coefficients of the statement wise transformation matrices, thanks to affine form of Farkas lemma [6, 24]. By the procedure, an affine transformation search space is obtained, where each (integral) point represents a valid program restructuring [25]. Of the several approaches of scheduling, the approach by PLUTO [2] has been shown to be practical as it gives a transformation with maximum permutable schedule dimensions. This makes loop tiling legal over permutable band of the schedule. Finally, the original program that is modelled as a union of polyhedra and a schedule (a bijective map from original space to transformed space) are given as input to code generation algorithm [7] which generates the loop nests that scan the transformed union of polyhedra. Parallel schedule dimensions can be detected and marked as `doall` loops.

PPCG PPCG is an open-source C to CUDA-C/OpenCL automatic parallelizer based on polyhedral model. PPCG detects SCoPs from input C programs using Polyhedral Extraction Tool (PET) [26]. PPCG performs dependence analysis by using Integer Set Library (ISL) [27], a library for manipulating integer sets bounded by affine constraints. ISL supports algorithms useful in analyzing and transforming SCoPs, like Feautrier's dependence analysis [5].

To compute a schedule, PPCG uses the ISL scheduler—a variation of PLUTO's algorithm [2]—which allows to set several options so as to tailor the scheduling process. PPCG finds a schedule by maximizing permutable dimensions. PPCG generates GPU code only if each permutable band present in the schedule contains at least one parallel schedule dimension. This allows to generate GPU code for each permutable band by tiling it. Since GPU architectures support two levels of parallelism (block and thread level), outermost parallel dimensions within each permutable band are tiled to support these two levels of parallelism. The tile loops are then mapped to blocks, and the point loops are mapped to threads. Based upon the number of parallel schedule dimensions in each band, PPCG generates 2D or 3D kernel grids and thread blocks. PPCG uses ISL-AST code generator [28] to obtain kernel AST. The generated ISL-AST is transformed into CUDA or OpenCL kernel function. PPCG also generates corresponding host code.

PPCG supports some GPU specific optimizations of which, the prominent are: (1) *Array linearization*: To enable coalesced access to GPU global memory. (2) *Array privatization*: Avoid accessing global memory by promoting arrays to scalars. (3) *Usage of shared memory*: By creating a small tile of the array and storing that tile in shared memory.

2.5 GPUs: Memory Hierarchy

As per Flynn's taxonomy, GPUs comes under the SIMD category. A GPU architecture involves collection of streaming multiprocessors (SMs) each having 32 cores. A GPU application is executed by launching large number of threads. All the threads are grouped into fixed sized (user-specified) blocks. Each block is scheduled to execute on some SM. SM scheduler selects consecutive 32 threads

(termed as a warp) from a block and executes it in SIMD fashion. GPUs hide memory latency, pipeline stalls by switching to ready-warps.

A GPU architecture comprises of DRAM where all the address spaces are mapped. By default, all the program arrays are stored in global memory space. GPU supports a low latency, software managed shared memory which is essentially a part of L1 cache. GPUs typically support large register files (to store private variables) facilitating parallel access by threads. Also, in order to optimize accesses to read only data, GPUs support a special type of cache, namely **constant** cache. Historically, GPUs were used for accelerating image/video processing applications and **texture** cache was designed to accelerate such applications. Texture cache is supported by a sophisticated caching hardware: *texturizing hardware*. Considering the topic of this section, we discuss these special caches in detail.

Constant Cache: The constant cache is additional to, and physically different from regular on-chip L1, L2 level caches. It is optimized for read-only broadcast access patterns, and is useful in cases when all the threads within a warp need the same (read-only) data. For applications involving large number of arrays, accessing the suitable arrays through constant cache helps to improve the utilization of L1, L2 level caches.

Texture and Surface caches: The Texture/Surface caches are specially designed for spatial array access patterns. While accessing 2-D or 3-D arrays⁴ using texture/surface caches, the array linearization is done in a way that preserves the spatial neighborhood of array elements, as it would be in 2-D or 3-D space respectively. By doing so, when a particular array element is accessed, its spatial neighbor elements are *cached*. In contrast, for normal caches, linearizing 2-D or 3-D arrays (by row-major, or column-major order) destroys the spatial neighbourhood.

This special type of array flattening—linearization which preserves spatial neighbourhood—is typically achieved by using space-filling curves (mainly Z-curves). The uniqueness of Z-curves lies inside their simplicity of interpolation function that is required for address translation mechanism. The interpolation for Z-curves involves application of bitwise operations (like *and* and *shift*), that can be implemented as a hardware circuits, thereby making address translation mechanism to be hardwired. It is clear that for a normal L1/L2 level cache to take care of such special types of array flattening, it would have to be supported by sophisticated cache mapping techniques along with an address translation mechanism. Hakura et al. [29] and Doggett et al. [30] discuss these techniques in detail.

When to use Texture/Surface Caches? Texture/surface caches are promising for array accesses patterns that exhibit spatial locality, like, in case a program accesses both $B[i][j-1]$ and $B[i-1][j]$ values. Then, no matter how array flattening is done (row/column major order), it is very likely that at least one cache miss will occur.⁵ In case, if any other thread within the same warp also accesses one of those two locations, then one global memory access gets saved.

2.5.1 Read-Write incoherency

Texture cache only supports read-only arrays. The workaround to access writable arrays via texture cache is as follows: create two copies of the array with one copy residing on texture cache while other on global memory. All the reads from that array can be altered programmatically so as to

⁴Note that array dimensionality should be strictly less than 4 so as to access it through texture/surface caches.

⁵A subtle assumption here is that: array A is large enough, and no data locality transformations are applied.

access from the copy residing on texture cache. All the writes to that array must be performed to the copy residing on global memory. However, this is guaranteed to work only if the update (written) array value is not read by any of the thread inside a kernel. This is because the updated value resides on the global memory array copy and array is read from the copy bound to texture cache. This is termed as *Read-Write incoherency* and should be avoided to ensure correct semantics. The workaround also suffers with a limitation of requiring two copies of array; increasing memory footprint of the application. To eliminate this limitation, NVIDIA GPU architectures starting from Fermi (with Compute capability ≥ 2.0) support a writable variant of texture cache, namely surface cache. We note that read write incoherence should be avoided to ensure correct semantics if program uses surface cache.

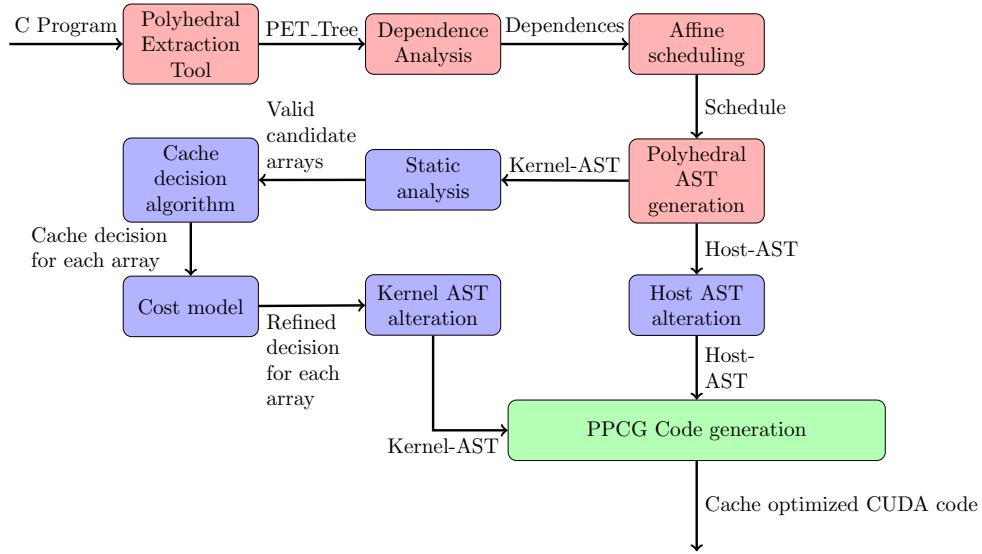


Figure 2.3: Overview of our Framework (Colour Encoding: Red \equiv PPCG module, Blue \equiv Proposed module, Green \equiv Modified PPCG module.)

2.6 Automatic Framework for Cache Exploitation

In this section, we describe our automatic cache exploitation framework. By default, we assume that the surface cache is supported. We also exposed the `-no-surface-memory` flag so as to have flexibility and backward compatibility for target GPUs having compute capability ≥ 2 .

Our schema is explained in Figure 2.3. First we allow PPCG to analyze, transform the input program. Once PPCG generates the AST, our algorithm performs static analysis so as to ensure that Read-Write incoherency is avoided. We discard the arrays which result in incoherency and access them using global memory. Then, we use the cost model to reason about profitability of accessing an array via candidate GPU caches. After making the appropriate cache decision for each array, we alter the kernel-AST and host-AST.

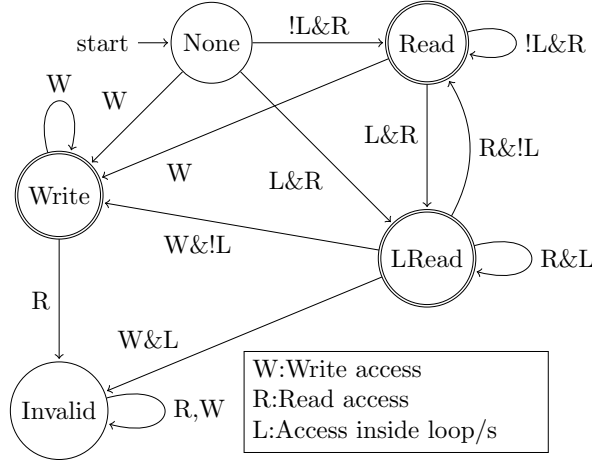


Figure 2.4: Automata showing transitions of `new_state`

2.6.1 Static Analysis

In order to preserve semantics, arrays resulting in read-write incoherence must be discarded from candidate list of surface cache. In other words, during kernel execution, updated array value (by any of the thread within a kernel) must not be read (by same or other thread). Since, GPU programs are Single Instructions Multiple Data (SIMD), it suffices to make sure that for a given array, kernel-AST is free of write followed by Read access (W-R access sequence) for a candidate array. Notice that, the analysis is conservative in the sense that information about accessed location is discarded. Our analysis traverses the entire GPU-AST visiting kernel statements, and detects potential W-R access sequence for all the arrays in the program. A subtle point while traversing tree is that children of a node must be processed in right to left order. This ensures correct analysis for assignment statements of following form: where A is free from read-write incoherence: $A[i] = A[i] + b[i]$

Another small caveat in static analysis is that, if a loop contains Read followed by Write access (R-W sequence) for some array, then it is likely to cause R-W incoherency. Because of implicit back-edge for a loop, (R-W) sequence embedded within a loop can effectively be: R-W-R-W-R-W,... Hence, static analysis must look for R-W sequence as well; when array access is surrounded within a loop nest. If the transformed-AST comprises of if-else constructs, we analyze each branch separately and perform a conservative analysis. This means, presence of R-W incoherence in either of the branches implies presence of R-W incoherence in the entire if-else construct. A pseudo-code which performs static analysis is summarized in Algorithm 1. The corresponding automata which decides how `new_state` gets computed is shown in Figure 2.4.

The surface cache is refreshed each time a new kernel is launched [31]; thereby, values updated by the previous kernel would be read in the next kernel launch while preserving coherency. Therefore, we do not need to look for W-R incoherence across kernels. Hence, we force all the candidate arrays in NONE state before analyzing next kernel.

We note that detecting read only arrays for a sequence of GPU kernels (which are extracted from a single SCoP) turns out to be a special case of static analysis. The arrays accepted via `Read` or `LRead` state of the automata are guaranteed to be read only arrays. We consider such arrays as candidates for texture and constant cache.

Completeness

The input program to PPCG is affine. Polyhedral transformations are essentially iteration reordering transformations. Hence transformed AST also has regular control flow involving only loops, if-else constructs, statements. Our static analysis considers all of these constructs.

Soundness

Our static analysis assumes relative ordering among memory access is same as in program text. However, due to architecture level optimizations (such as load/store buffering), such ordering may not necessarily get respected at runtime. This is especially true in case of weaker memory models. It has been experimentally observed by Algave et al. [32] that GPU memory models are believed to have weaker memory models.⁶

```
1
2  __device__ kernel (...) {
3     int i = threadIdx.x + blockDim.x*blockIdx.x
4     B[i] = A[i] + A[i+1] + A[i+2];
5 }
6 .visible .entry _kernel (...) {
7     ...
8     suld {%r28}, [surfRef_A, {...}]; //Read Surface values
9     suld {%r29}, [surfRef_A, {...}];
10    suld {%r31}, [surfRef_A, {...}];
11    ...
12    mul.f32 %r42, %r29, 0f3E4CCCCD; //computations
13    ...
14    sust [surfRef_B, {...}], {%r42}; //Write Surface values
15 }
```

Listing 2.1: Sample kernel and PTX assembly after using surface cache. Due to intermediate computations, reads are guaranteed to finish before initiating surface writes; thereby making memory view consistent.

We make the important observation to ensure the validity of static analysis at run-time even on weaker memory models. Every GPU kernel generated by PPCG is free of data-races due to exact data-dependence analysis. Hence, there are no observers (read/write from other threads) for a memory location written by a particular thread. Single thread always has a consistent view of memory. In spite of the above observations, two memory accesses (to different addresses) within a thread might get reordered (because of architectural latencies). Due to such re-ordering, it may appear that static analysis may get invalidated at execution time.⁷ However, we make the following argument to rule out such possibility: (1) Arrays accessed through texture and constant caches are read-only, and notice that re-ordering of two read-accesses does not invalidate the static analysis during execution time. (2) For surface caches, our static analysis rejects candidate array with W-R

⁶Exact characterization of memory consistency model needs accurate architecture details (eg memory latency) that are not revealed by GPU vendors.

⁷R-W sequence may effectively be W-R thereby violating semantics.

access sequence. Furthermore, write only arrays are not good candidates for surface cache because surface cache is a writable variant of read-only texture cache. This implies that the candidate array is guaranteed to be accessed *only* by R-W access sequence within a kernel. At the start of the kernel, all the threads read required values from reference bound to surface cache, then perform computation, and finally at the end of the kernel writes values through the reference. Observe that surface reads must be completed before starting the actual computations otherwise it leads to incorrect results. Also, newly computed values will be written to surface cache implying writes must be initiated after computations. Transitively, we can infer that surface reads are guaranteed to be finished before starting surface writes thereby making static analysis valid (Refer 2.1). Finally, reordering of surface-reads (correspondingly surface-writes) among themselves still makes the static analysis to be true at run-time. Even though our static analysis might appear to be designed considering a sequential consistency of memory operations, it is sound for weaker memory models (like GPUs) due to specialty in access pattern.

2.6.2 Cache selection

Static analysis aids in analyzing the candidate arrays, so that semantics are preserved even after accessing each candidate array through appropriate caches. For cache selection, first we analytically decide potential caches, then we query cost model (discussed in the next section) for profitability checks. The analytical cache decision tree 2.7 decides potential caches for arrays. The decision is made at per-SCoP granularity. This avoids array setup time in-between two successive kernel launches which are extracted from the same SCoP. Depending upon `no-surface-memory` flag, algorithm decides whether to use surface cache or not. Read only arrays are candidates for texture, constant cache. Algorithm chooses global memory for: (1) Candidate arrays end-up being in invalid state (2) Write-only arrays.

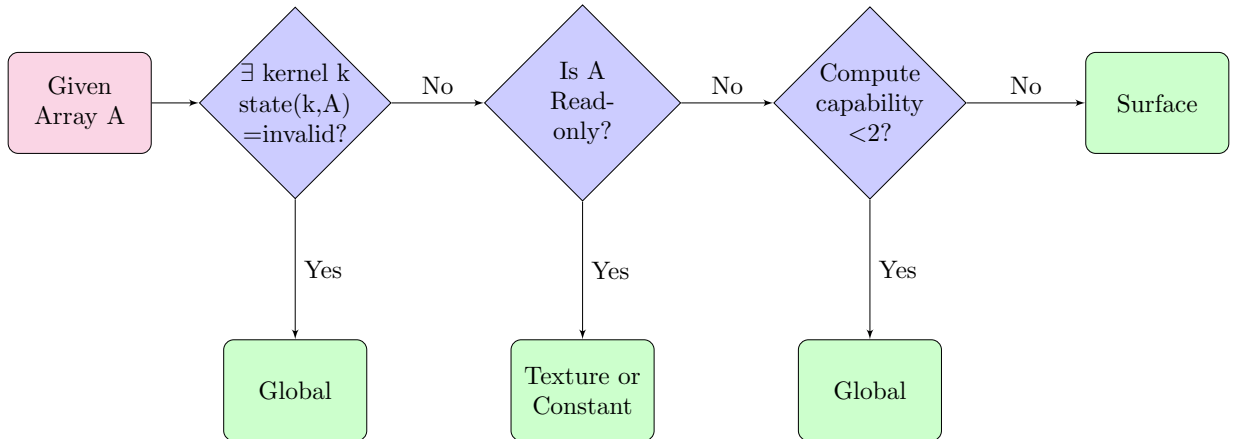


Figure 2.5: GPU-cache decision tree

2.7 Cost Model

In this section, we use the power of polyhedral framework to design a cost model for each variety of cache. We analyze each array access present in the kernel and reason about its profitability.

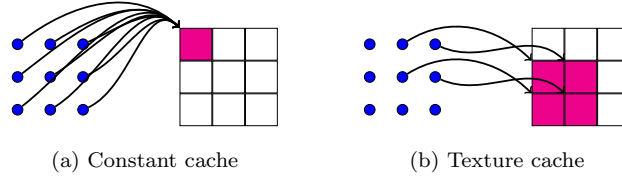


Figure 2.6: Profitable access pattern for a 2-D array

In the polyhedral model, array accesses are modelled as a functions from iteration domain(I) to array access location(L). Formally, $F_{original} : I \rightarrow L$. Transformed array access function for array A is derived using original access function and newly computed schedule ($F_{transformed}(A) = F_{original}(A) \circ schedule^{-1}$). Our cost model analyzes transformed array access functions for all candidate arrays. For notational convenience, we assume $F_{transformed}(A)$ denotes transformed array access functions for all the references to array A .

2.7.1 Cost model for Constant cache

Constant cache is well suited if all the threads within a same warp access the same memory element. In other words, the array access expression should be independent of thread identifiers (`threadIdx.x`, `threadIdx.y`, `threadIdx.z`). Recall that PPCG maps tile loops to blocks, and point loops to threads. Therefore, array access function should be free from the point loop iterators($iter_{point}$) [Fig. 2.6]. Formally, for a candidate array A , if following formula evaluates to *True*, we decide to access it using constant cache.

$$\bigwedge_{a \in F_{transformed}(A)} \bigwedge_{p \in iter_{point}} a.isIndependent(p) \quad (2.1)$$

2.7.2 Unified Cost Model for Texture/ Surface caches

Recall that usage of texture and surface cache is profitable if threads within a same warp access neighbouring locations in 2-D or 3-D data space (See Fig. 2.6). Our cost model captures this situation by operating over integer sets.

We assume the following notation: let A be the array to be analyzed and n be the dimensionality of A . Recall that $1 \leq n \leq 3$. Let f be function with which array A is accessed for some statement S present in GPU kernel AST. Let I be the transformed iteration domain for statement S .

First we construct a set D which captures the neighbouring elements of Array A . Since $D \subset A$, it follows $\dim(D) = \dim(A)$. We construct universal set of dimension n . Then for each dimension $d \in D$ we add a constraint of form: $0 \leq d \leq 2$. Operating on parametric integer sets leads to un-scalability. Since we need to analyze every candidate array access appearing in the transformed program, we purposefully fix the starting location of set D within array A . Such a design allows to efficiently estimate (as opposed to exact calculation of) the spatial locality. Next, we compute the set $T \subset I$ which comprises of iterations accessing the elements in set D . We use pre-image operation to compute T .

$$T = PreImage(f, D)$$

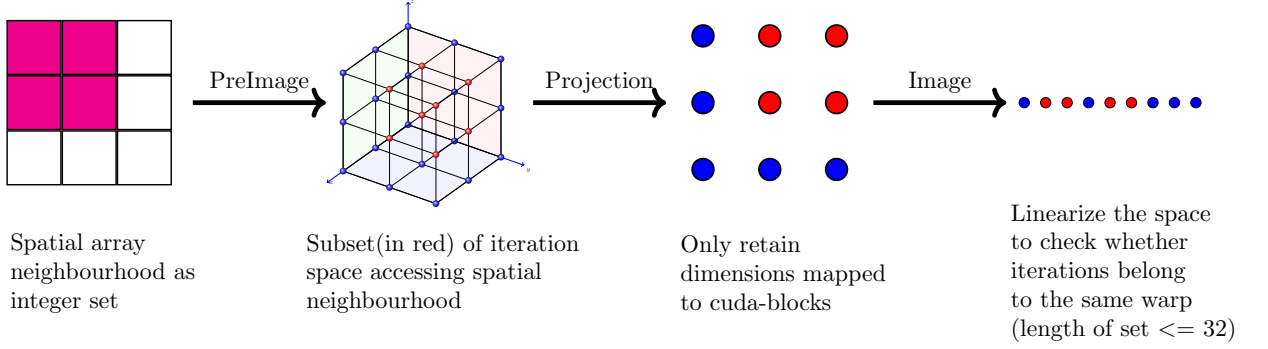


Figure 2.7: Statically modelling spatial access pattern within a GPU-warp through polyhedral operations

Note that, T may include following types of loop dimensions. (1) Outer loop dimensions which corresponds to loops mapped to host code. (2) Block dimensions corresponds to the loops mapped to blocks (3) Thread dimensions represents loops mapped to threads. (4) Inner loop dimensions represents the loops appearing within a thread. We are interested in estimating spatial locality available within a warp. Hence, we project out the all but loop dimensions which are mapped to threads.

$$T' = Project_out(T, dim_{outer}, dim_{inner}, dim_{block})$$

GPU architectures supports upto 3 thread dimensions (`threadIdx.x`, `threadIdx.y`, `threadIdx.z`). Hence, $\dim(T') \leq 3$. GPU runtime linearizes 2-D or 3-D thread dimensions [31] by using following mapping where B_x, B_y represents the kernel block dimensions.

$$L : t_{linearized} = t_x + t_y * B_x + t_z * B_x * B_y$$

PPCG uses tile sizes as kernel block dimensions. We use tile sizes in place of B_x and B_y . We refer to this function as linearization map L . Using L we can linearize T' to obtain 1-D set W .

$$W = Image(L, T')$$

W exactly captures the linearized iterations that access neighbouring elements in array A . We can compute length of W as follows:

$$W_{length} = lexmax(W) - lexmin(W)$$

Observe that computing W_{length} involves solving *two* integer linear programming problems. W_{length} can be interpreted as fictitious warp size which accesses neighbouring elements in Array A through given access functions f . For each array A we compute average fictitious warp size W_{avg} by computing average of W_{length} computed for each access function f . In ideal case W_{avg} should be close to (or less than) 32 (actual warp size on GPUs). Due to thread linearization, it may not necessarily be the case. In order to make a decision on whether to use texture or surface cache we use criterion shown in table 2.3

Spatial locality for 1-D array is implicitly captured by normal (L1, L2 level) caches. Therefore cost model refuses to access such arrays from texture/ surface cache. The upper bounding threshold value for W_{avg} for texture cache is set to twice of actual warp size on GPU. While for surface cache,

Table 2.3: Profitability parameters for texture, surface caches

Cache/ Memory choice	Metrics of profitability		
	Array accesses	Estimate of spatial locality (W_{avg})	Array Dimension
Global	Write-Only	$W_{avg} \geq 65$	1-D
Texture	Read-Only	$W_{avg} \leq 64$	2-D/3-D
Surface	Read+Write	$5 < W_{avg} \leq 50$	2-D/3-D

upper bound for W_{avg} is set to 1.5 times of actual warp size. We empirically observed that usage of surface cache introduces little overhead (larger setup time) as opposed to texture cache, therefore bounds on W_{avg} are made tighter for surface cache. The threshold value setup is based on empirical observations and tuning of cost model may be necessary, but we do not consider this aspect as of now and leave it for future work.

2.8 Code-generation

The code-generation phase consists of augmenting the PPCG-CUDA code-generator so that arrays are accessed from references bound to appropriate caches. We alter the host and kernel code generation phases of PPCG to setup constant, textures, surfaces references and also to access arrays from apt reference.

Host Code-generation: For each array selected to be accessed via texture/surface cache, we generate code to: (1) Declare a unique texture/surface reference. (2) Declare a channel format. (3) Declare and allocate `cudaArray`. (4) Generate a code to Copy array from CPU memory to `cudaArray`. (5) Bind `cudaArray` to texture/surface reference. At the end of SCoP we generate code to: (1) Unbind references. (2) Copy live-out arrays from surface cache to CPU memory. For constant cache candidate arrays we generate code to: (1) Declare a device array qualified by `__constant__` (2) Copy CPU array to constant array.

Kernel Code-generation: The main objective during kernel code generation is to transform the AST node representing candidate array accesses to the appropriate CUDA function calls. We transform each constant cache candidate array access so as to read from the array qualified with `__constant__`.

2.9 Performance Evaluation

This section describes our experimental results. We implemented our approach within PPCG-0.06 using pet-0.09, Clang-5.0.0 and ISL library (isl-0.17.1-GMP).

2.9.1 Experimental setup

We evaluate our implementation on PolyBench/C 4.2.1-beta [33] a widely used benchmark in Polyhedral community with 30 kernels spanning domains from linear algebra, stencils etc.

Our machine comprises of two chips of Intel(R) Xeon CPU E5-2670 with 2.60GHz clock frequency. Each chip comprises of 8 cores (with hyperthreading) thereby making total of 16 virtual cores. We use NVIDIA Tesla K20Xm GPU having Kepler microarchitecture. The compute capability of Tesla

Table 2.4: PolyBench/C: Various GPU caches automatically exploited by the framework

PolyBench/C	GPU caches		
	Constant	Texture	Surface
JAC-1D, JAC-2D, HEAT-3D, GRAMSCHMIDT	✗	✗	✓
FDTD-2D, DURBIN, GEMVER, LUDCMP	✓	✗	✗
GEMM, 2MM, ATAX, BICG, MVT, GESUMMV, TRMM, TRISOLV	✓	✓	✗
SYRK, SYR2K, 3MM, DOITGEN	✗	✓	✗
SYMM, DERICHIE, NUSSINOV, ADI, CHOLSKY, FLOYD-WARSHALL, SEIDEL, LU, CORRELATION, COVARIANCE	✗	✗	✗
Total = 30 Programs			

K20X is 3.5 and hence supports surface caches. The GPU comprises of 14 Streaming multiprocessors having 192 cores per SM thereby making total core count to 2688. Maximum clock rate of our GPU is 0.73 GHz. The GPU comprises of 5.7 GB of global memory. The version of CUDA compilation tools is 8.0 (V8.0.61). NVCC compiler uses GNU C++ compiler for host code compilation. We use nvprof (release version 8.0.61 (21)) to collect various performance metrics.

PolyBench supports various dataset granularities in order to scale array sizes proportionally. Typically, constant cache size is much smaller than texture/surface cache size, hence we use `SMALL_DATASET` granularity during benchmarking constant cache. In contrast, we use `LARGE_DATASET` (default granularity) for texture/surface cache evaluations. Furthermore, for all experimentation we **disabled** array linearization transformation in PPCG so as to retain the spatial locality of arrays which we are interested to exploit via GPU caches. We use float as a data-type.

PPCG generated code is compiled using the NVCC compiler. For recording the execution times, we use following procedure: (1) Run each benchmark five times. Record execution time. (2) Eliminate two extremal execution times, and verify that deviation for remaining is less than 5%. If not repeat the experiment. (3) Repeat above steps thrice. (4) Take *geometric mean* of these three values. With this procedure, we also record execution times for following two cases which serves as a baseline: (1) PPCG generated CUDA code which uses only global memory. (2) PPCG generated CUDA code which uses shared memory. The table 2.4 depicts varieties of GPU caches automatically exploited on PolyBench/C kernels.

2.9.2 Experimental Results

In this section, we discuss the results of performance evaluation. First we discuss the results for constant cache optimizations and then, the texture, surface cache optimizations.

Constant Cache Results: The baseline for evaluation is PPCG generated global memory based CUDA code. The table 2.5 shows the speedups obtained when suitable arrays are accessed using constant cache or shared memory. Out of 30 programs in PolyBench/C, 11 are successfully transformed through our framework. We observe that 7 programs show small but significant speedups. We also observe that except for 2MM and MVT kernels, constant cache performs better over shared cache. It is well known that usage of shared memory results in better speedups compared to global memory based implementation. But due to small dataset sizes, significant amount of kernel exe-

Benchmarks	Performance in MFLOPS/Sec					
	K20			P100		
	Global	Shared	Constant	Global	Shared	Constant
GEMM	12316.77	49707.91	5082.06	20104.85	74330.59	11358.75
2MM	6047.92	21365.82	2294.12	10100.75	30685.17	5060.15
ATAX	339.31	778.41	340.36	1015.50	2041.65	993.98
BICG	351.15	800.36	354.07	998.14	1997.28	956.38
MVT	345.57	781.99	350.18	988.28	1977.30	957.50
GEMVER	835.47	1739.46	817.36	2307.91	3942.59	2236.71
GESUMMV	187.35	214.61	188.58	560.44	492.94	561.57
LUDCMP	12.65	12.68	12.66	27.81	33.09	27.83
TRMM	382.65	391.09	385.14	960.09	1584.38	1278.81
DURBIN	11.083	10.87	11.40	13.72	13.31	13.89
FDTD-2D	3849.35	4369.06	3835.57	4711.53	4868.49	4743.19

Table 2.5: Effectiveness of constant cache on PolyBench/C with problem size=SMALL DATASET

cution time is spent in loading arrays to shared memory and synchronization barrier following this data copy. Both of these overheads are completely avoided by using constant cache for some arrays, thereby resulting in a (slightly) better performance. To further investigate the potential of constant cache, we record (using `nvprof`) the percentage reduction (with PPCG-Global as a baseline) in global memory loads after exploiting constant cache.

Texture/Surface Results: Out of 30 benchmarks in PolyBench/C, 15 programs are transformed through our framework. We observe speedups for 11 programs compared to global memory based CUDA code. The geometric mean of the speedups is 1.05. We notice maximum speedup of 1.5x for heat-3d kernel due to 3-D spatial locality present in the kernel.

Benchmarks	Performance in GFLOPS/Sec					
	K20			P100		
	Global	Shared	Texture/ Surface	Global	Shared	Texture/ Surface
GEMM	99.082	343.366	77.987	161.204	1417.003	119.575
2MM	87.526	197.509	68.442	198.097	1107.626	122.044
3MM	64.091	230.902	53.297	109.107	991.159	87.749
ATAX	5.128	13.787	10.346	11.182	23.789	33.714
BICG	5.203	13.510	10.371	10.700	22.908	31.429
DOITGEN	0.947	2.687	2.028	1.124	4.646	3.201
MVT	5.139	13.633	10.405	11.883	22.967	31.314
GESUMMV	2.687	4.729	6.928	7.076	5.724	8.752
SYR2K	28.126	41.486	252.742	253.715	482.404	183.062
SYRK	27.222	107.757	244.236	241.343	332.702	181.327
TRMM	6.257	6.504	5.833	17.003	24.977	8.470
GRAMSCHMIDT	0.409	0.409	0.319	1.125	1.125	1.370
TRISOLV	0.278	0.246	0.290	0.334	0.293	0.339
HEAT-3D	21.669	N/A	42.168	94.798	N/A	111.864
Jacobi-2D	46.679	N/A	42.908	155.818	N/A	359.958

Table 2.6: Effectiveness of texture, surface caches on PolyBench/C with problem size=DEFAULT DATASET

The Table 2.6 also shows speedups obtained by PPCG when shared memory is enabled. For stencils, the PPCG-shared memory heuristic fails to find array tiles to store into shared memory,

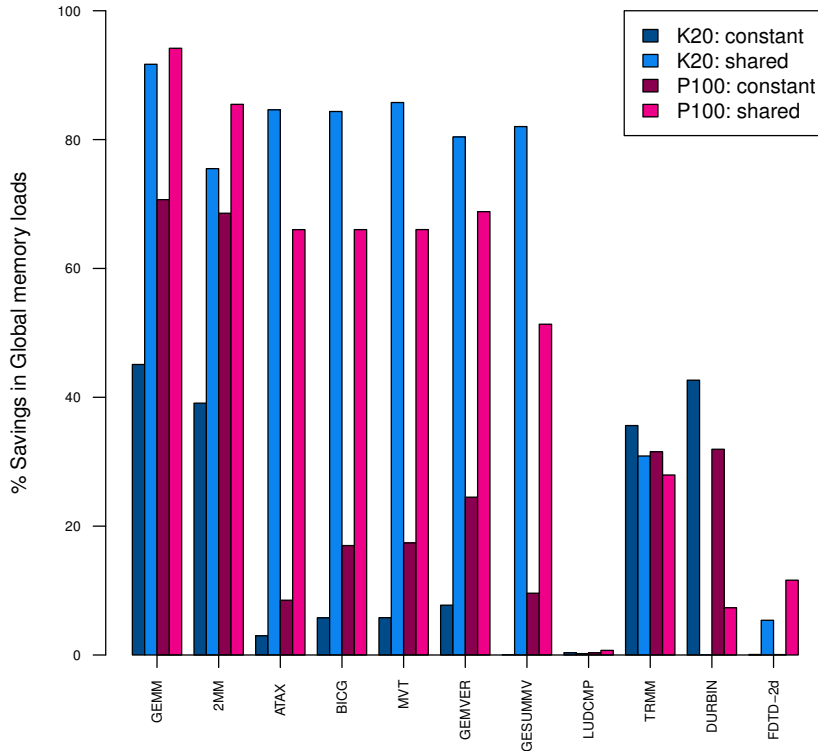


Figure 2.8: Reduction in global memory read requests after exploiting constant cache and shared memory

and so it cannot generate shared memory based code. In contrast, our new framework is able to generate surface cache optimized CUDA code for stencils.

We also observe that access patterns present in SYR2K, SYRK, BICG are more suitable for texture cache, resulting in better performance. In case of GEMM (matrix multiplication kernel), PPCG uses shared memory for both of the read-only matrices. On the other hand, our cost model rejects one of the array to access from texture cache. Therefore, shared memory based code performs better. The argument equally applies to 2MM and 3MM.

We study percentage reduction in global memory accesses after usage of texture/surface caches. The Figure 2.9 shows the result of our study. The maximal degradation in performance for gram-schmidt kernel can be justified by referring to the plot. In spite of spatial locality in jacobi-2d, it fails to achieve good speedup (as opposed to heat-3D kernel). For jacobi-2d kernel, reduction in global memory reads is quite small. jacobi-2d kernel performs only 5 reads (while heat-3d performs 9 reads) per thread.

For this subsection, we do not present comparative results of PPCG with other tools over Poly-Bench/C suite. The reader may refer to the PPCG paper [13] to see how it gives impressive speedups when compared to other manually annotated tools like PGI, CToCUDA-C, OpenMPC etc., including hand-optimized library like CUBLAS.

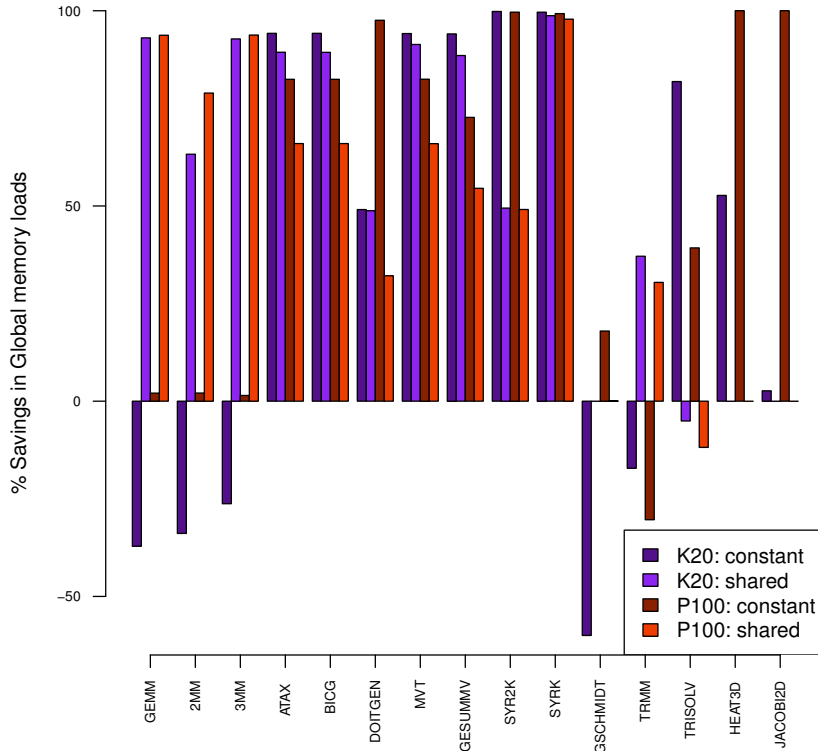


Figure 2.9: Reduction in global memory read requests after exploiting texture cache and shared memory

Benchmark	Tesla K20		Tesla P100	
	Shared	Surface	Shared	Surface
DOITGEN	98.148%	93.827%	98.148%	96.296%
GRAMSCHMIDT	0.002%	0.324%	0%	0.302%
HEAT-3D	N/A	66.494%	N/A	100%
Jacobi-2D	N/A	27.767%	N/A	100%

Table 2.7: Reduction in global memory write requests after exploiting surface cache and shared memory

2.10 Real-World Use cases

In this section, we present some representative real world computations which benefit by our new framework. In addition to our motivating example (RNN), we consider the following computations:- PageRank algorithm [34]: used widely to score web-pages by finding eigenvectors for Google's stochastic matrix; LSTM [9]: A widely used layer in natural language learning tasks; PoissonSol [35]: Poisson's PDE solver over 3D grid; and DWE-FDTD [36, 37]: A finite difference discretization kernel which is the basic building block in Reverse Time Migration (Seismic imaging).

To provide comparative study with other state-of-art tools, we consider two most representative optimizing compilers namely: Intel C++ compiler (version 17.0.4) and PGI compiler (version pgcc 17.4-0). While Intel Compiler is fully-automatic, the PGI compiler requires precise user-annotations for good performance. In comparing our framework with ICC and PGI, we use the same methodology followed by Verdoolaege et al.[13]. The method involves usage of state-of-art source-to-source

PLUTO compiler [2] (v0.11.4) so as to extract coarse grain parallelism by applying complex loop transformations. We use PLUTO transformed OpenMP code to measure performance numbers with ICC. Then, we interpret PLUTO generated OpenMP pragmas to OpenACC. As PLUTO only annotates outermost parallel loops with OpenMP pragmas, we manually annotate the inner parallel loops. We also manage CPU-GPU data transfers by appropriate pragmas, enable usage of shared memory by using cache directives in OpenACC. We explicitly annotate reductions. As a baseline for this part of experimentation, we use PLUTO + ICC -O3 configuration (without OpenMP), representing exploitation of parallel vector lanes on CPU⁸. In

Fig. 2.10, we compare the performance of various PPCG configurations⁹ with PLUTO+PGI).

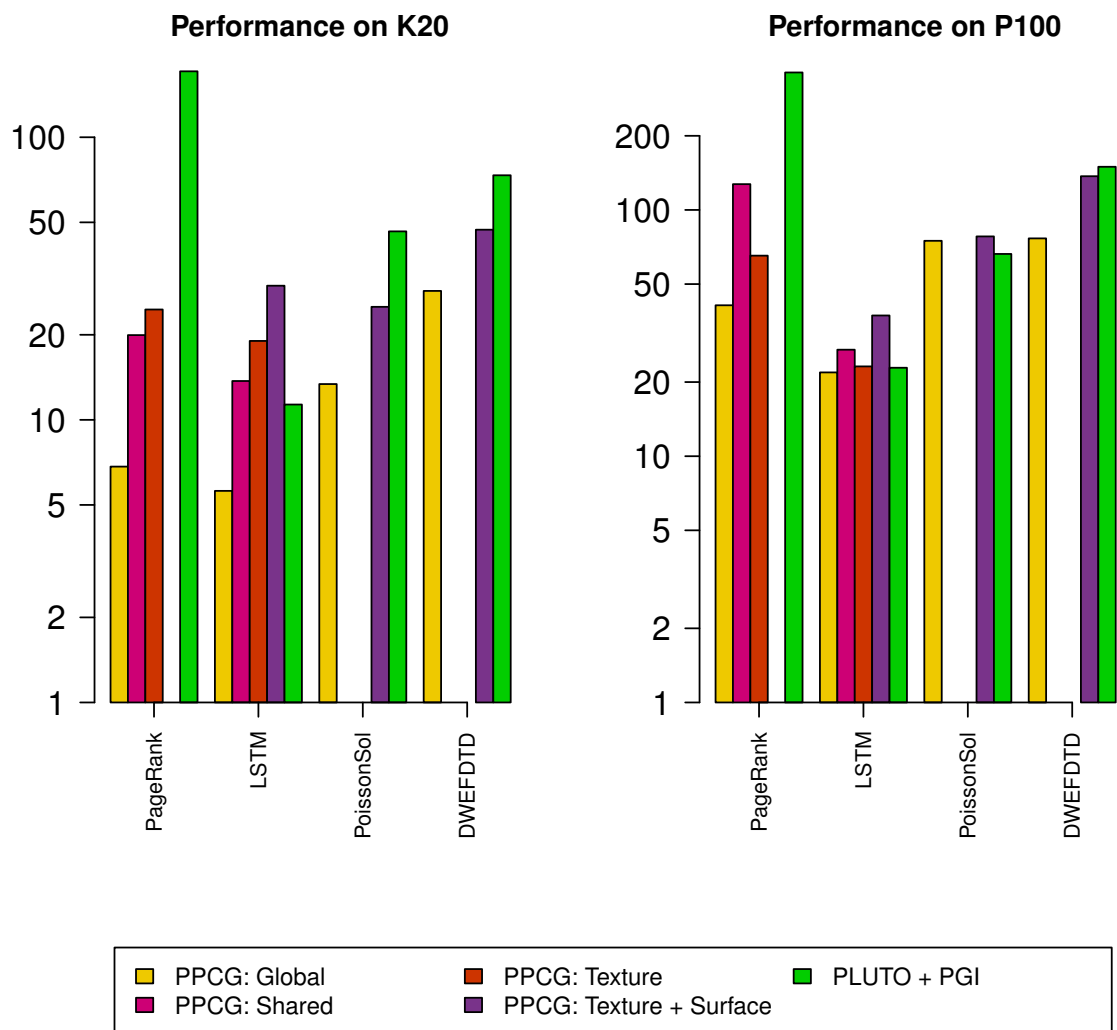


Figure 2.10: Various PPCG configurations against manually annotated code using openACC (Log scale for y-axis)

⁸In this case, PLUTO also improves data locality.

⁹Due to large array sizes, we do not consider PPCG-Constant configuration.

Some observations:

Among all our use-cases, (only) LSTM, due to the nature of its computation, is amenable for simultaneous exploitation of both texture and surface caches; thereby resulting in a further (1.3x) speedup over PPCG: Texture.

In case of both PoissonSol and DWE-FDTD, our framework *outperforms* PPCG-Global, by a factor of 1.8x and 1.5x respectively. In case of both these kernels, PPCG’s heuristic is unable find profitable array-tile to load into shared memory. In contrast, our framework (cost-model) is successful in exploiting spatial locality for efficient use of surface caches for both of these kernels. And, due to presence of Read/Write array accesses, it is not possible to use (read-only) texture caches for these kernels.

The PGI compiler when annotated with precise OpenACC pragmas is able to generate efficient target GPU-architecture-specific code by performing low level optimizations on PTX code. However, the performance of the code produced by PGI is *highly dependent* on the user-placement of pragmas, and hence needs (expert) user intervention. On the other hand, all the PPCG configurations (including our framework) are *fully automatic*, and hence no extra annotations are required. Our framework *automatically achieves comparable results* to user-annotated code. In fact, our framework outperforms PLUTO+PGI configuration in case of (1) LSTM kernels on both platforms. (2) PoissonSol on P100. In all other cases, our framework helps PPCG to obtain better speedups against PLUTO+PGI without sacrificing its fully-automatic nature.

We obtain better numeric speedups (than for the ones in PolyBench) for *all our use-cases*¹⁰. The observations drawn from our motivating example in Sec.2.2 help explain this. PageRank and LSTM suffer from bottleneck-latencies of shared-memory which are eliminated after exploiting caches, while PoissonSol and DWE-FDTD benefit from 3-D spatial locality aware caching technique of surface caches.

2.11 Conclusions and Future work

We proposed a complete end to end framework to support varieties of GPU caches in the polyhedral compiler PPCG. To the best of our knowledge, this is the *first attempt to automatically generate CUDA code that exploits various GPU caches*, beginning from a C program. In order to generate cache optimized CUDA program, our framework performs static analysis to preserve the program semantics. We use the power of polyhedral model to design a cost model which helps to decide the type of cache to be used for accessing candidate arrays. We present a comparative study of various caches vs. shared memory based CUDA codes. We have shown the importance of our framework in accelerating five important real-world computational kernels. Traditionally, texture caches are designed for image processing pipelines. Our work shows the *general purpose computing use of these special caches*. With increasing interest in mapping general purpose algorithms on GPUs, we believe that our contribution will make a strong case towards automatic exploitation of complete GPU memory hierarchy, along with improving the scope of usage of texture, constant and surface caches.

In future we would like to support code-generation for OpenCL. We also believe that more precise static analysis is possible instead of our conservative one. We believe that our new infrastructure

¹⁰Except PageRank on P100. Due to unification of texture and L1 cache in microarchitecture, texture cache gets exploited by CUDA run-time in presence of shared memory based code as well.

provides an ideal platform for exploring many related questions.

Chapter 3

Efficient Algorithms for Polyhedral Over-Approximation

Many of the analysis and verification problems in formal methods, static analysis are casted as solving mathematical data-flow equations over various abstract domains. Examples of the above are Convex-polyhedra, TVPI polyhedra, UTVPI polyhedra (Octagons) and Interval (Box) polyhedra.

The convex-polyhedral abstract domain is useful to capture the affine relationship among program variables. However, majority of the algorithms (like feasibility/optimization) developed for convex-polyhedra fail to scale for larger problem size due to their worst-case exponential complexity. In particular, both the feasibility and optimization problems on general convex polyhedra are not strongly polynomial in complexity.

The (Unit-)Two Variable Per Inequality Polyhedra (also called as Octagons) has been proven to be a very useful abstract domain due to its improved worst-case polynomial time complexity [38] and ease of implementation.

In this work, we propose two new algorithms for Over Approximating a given (arbitrary) convex polyhedron into its tightest UTVPI Over Approximation. We show the limitations of the existing Over-Approximation algorithm, and then discuss our new algorithms for Over Approximation (OA). Our first algorithm is based on linear programming, while our second algorithm is based on Fourier-Motzkin elimination (projections). Both of our algorithms return the tightest over-approximation of the given convex polyhedron.

Both the algorithms are very simple to reason as they fully exploit the Octagonal nature of the OA that they aim to obtain. The LP based algorithm is based on a series of linear programming calls. The Fourier-Motzkin algorithm is based on a series of rotations and projections of the polyhedron. Because of the above nature, they are also very easy to implement.

Furthermore, insight obtained from this algorithm, is used to design alternative algorithm which is free from (Integer) Linear Programming Problem.

Our two algorithms are implemented in the Integer Set Library (ISL) [27];, a open-source polyhedral compilation library.

3.1 Background

We now formally introduce some basic terminologies required for further sections.

Polyhedra: A Polyhedra is a space enclosed within n-dimensional vector-space by finitely many linear (in)-equalities.

Dual Representations of Polyhedra: A Polyhedron in n-dimensional space can be represented in two alternate ways, and these representations are considered duals of each other.

Hyperplane (H) representation: The Polyhedron is expressed as an intersection of finitely many affine inequalities.

Generator (V) representation: The Polyhedron is expressed as a convex combination of its extremal vertices, conical combination of its rays and a linear combination of its lines. A classic algorithm to convert either of these representations to the other is the algorithm by Chernikova.

U-TVPI Polyhedra (Octagon) A U-TVPI polyhedron is a special case of convex polyhedra where every affine constraint is restricted to the form: $\pm X_i \pm X_j \leq C_{ij}$. As the name suggests, every constraint should involve at most two variables and having coefficients be one of the following +1, 0, -1 [38] [39].

Difference Bound Matrix (DBM) A U-TVPI polyhedron can be represented in a compact representation namely Difference Bound Matrix (DBM); A NxN matrix representing the constant-values for each possible U-TVPI constraint.

Diameter (Fatness) of a polyhedron: A diameter (or Fatness) of a polyhedron corresponds to the the width of the polyhedron in given direction. A direction can be one of the canonical axes.

Affine form of Farkas lemma: An affine function is positive over a polyhedron P if and only if it is positive affine combination of hyperplanes describing polyhedron P. Essentially, affine form of farkas lemma is corollary to farkas lemma (A lemma followed from principle of duality in Linear Programming)

3.2 Limitations of state-of-art Mine's algorithm

While proposing the Octagonal abstract domain [39], Mine presented an algorithm, the first-ever, that Over-Approximates a convex polyhedron into a UTVPI polyhedron. His algorithm first converts the input polyhedron from H-form to its V-form. Once the generators of polyhedra have been enumerated, the algorithm iteratively tightens every possible U-TVPI constraint in n-dimensional space (which is of form: $\pm X_i \pm X_j$). The optimal (constant) value obtained by optimizing each such possible constraint with respect to all the vertices of the original polyhedra is then used to obtain U-TVPI over-approximating constraint. After this iterative procedure, the algorithm post-process the constraints in the direction of rays, so as to set upper (or correspondingly lower) bounding constraint to infinity value.

It is well known that enumerating generators of a convex polyhedra is never a polynomial time process: an n-dimensional hypercube can be represented with at most $2n$ halfspaces while it has 2^n many vertices.

3.3 Algorithm#1: The Farkas lemma based OA algorithm

Feautrier [40] in his seminal work on affine scheduling proposed to use the affine form of Farkas lemma as a means of avoiding the transformation of the polyhedron from H representation to V representation. The result of the application is a polyhedron that encodes all semantic preserving transformations of the input program.

In this section, we propose a new algorithm that uses the same lemma to construct a search space for (over-approximating) U-TVPI hyperplanes. The algorithm involves a series of linear programming calls ($\binom{N}{2}$ in total), which find the tightest over-approximation effectively minimized.

3.3.1 Enabling usage of Farkas lemma

Consider a U-TVPI constraint of form: $\pm X_i \pm X_j \leq C_{ij}$. We are interested in finding good numeric (or parametric) value for C_{ij} such that the resulting constraint is satisfied by every (Integer) point belonging to original convex polyhedron. Note that, such a constraint once formed, essentially ensures that the constraint can be part of over-approximating U-TVPI polyhedron. In other words, $C_{ij} \pm X_i \pm X_j \geq 0$ must be positive over original convex polyhedra. Now, farkas lemma can be applied to obtain bounds on C_{ij} . That is,

$$C_{ij} \pm X_i \pm X_j \equiv \lambda_0 + \sum_i (\lambda_i * (A_i x + B_i))$$

where

λ_i : farkas multiplier

$(A_i x + B_i)$: Hyperplane describing convex polyhedron P.

Now, we can equate the coefficients of the variables appearing from both of the sides; and project out the farkas multipliers. The resulting in-equality captures all feasible values for C_{ij} . This essentially represents a search space for over-approximating octagonal (U-TVPI) hyperplanes.

3.3.2 Joint search space and Cost function

By applying affine form of farkas lemma, we obtain a set of feasible values (or search space) for every possible over-approximating U-TVPI constraint. However, we are interested in finding tightest U-TVPI over-approximation. For that purpose, we construct joint search space for two opposite U-TVPI over-approximating constraints. For example, we construct a 2-dimensional search space (c_{ij}, C'_{ij}) obtained by applying farkas lemma to $X_i + X_j \leq C_{ij}$ and $-X_i - X_j \leq C'_{ij}$.

Consider a linear function $f : C_{ij} - C'_{ij}$. It represents the distance among two resulting over-approximating hyperplanes. Minimizing this cost function with respect to the search space (C_{ij}, C'_{ij}) essentially results in finding tightest over-approximation.

Formally, we use below objective function over the joint search space constructed: (C_{ij}, C'_{ij}) :

$$\text{lexmin} (C_{ij} - C'_{ij}, C_{ij}, C'_{ij})$$

That is $C_{ij} - C'_{ij}$ is minimized with highest priority, to find tightest over-approximation. In case of existence of two possible sets of values for (C_{ij}, C'_{ij}) with equal separating distance among them, the one with lower numeric value is preferred.

3.3.3 Iteratively finding the pairs of U-TVPI hyperplanes

Optimizing the cost function over the joint search space constructed gives two U-TVPI hyperplanes. For n-dimensional (bounded) convex polyhedra, we need to find $8^* \binom{N}{2}$ many hyperplanes. However, due to construction of joint search space, we iterative above procedure $4^* \binom{N}{2}$ times. The intersection of all such constraints found gives U-TVPI over-approximation of original convex polyhedra.

3.4 The Insight

We now investigate the relation between tightness of over-approximation with respect to the diameter (or fatness) of a over-approximating convex polyhedra. In above algorithm, we minimize the function: $C_{ij} - C'_{ij}$ to find tightest over-approximation. Notice that this function can be minimized only till a value, which essentially represents the diameter (or fatness) of a polyhedron along that direction. In other words, by estimating the diameter (or fatness) of original convex-polyhedron along some particular direction, enables to find the required constant values (namely C_{ij} and C'_{ij}) thereby providing the equations (or constraints) of U-TVPI over-approximating polyhedron. We use this insight to formulate another algorithm which is free from Integer-Linear-Programming. The core idea is to estimate the diameter of original convex polyhedron in various directions by using Fourier-Motzkin projection algorithm.

3.5 Algorithm#2: Fourier-Motzkin based OA Algorithm

Given a convex polyhedron P in n-dimensional space, we first project the polyhedron on all possible 2-d planes. In other words, we need to project polyhedron P onto $\binom{N}{2}$ many planes. We iteratively process each such projection (referenced as shadow henceforth) obtained.

Given a shadow S of a polyhedron P on say X_i and X_j plane, we further project it onto each axis (i.e. X_i and X_j). Notice that, this projection essentially results in finding a diameter (or fatness) a polyhedron in that particular dimension. The constraints obtained after projecting a 2-d shadow (of original convex polyhedron) on each individual axis provides a (lower and upper) bounds on the values allowed for that particular dimension. In this way, a 2-D rectangular bounding box can be constructed for a given shadow of polyhedron. This 2-D bounding box gives 4 constraints for over-approximating U-TVPI polyhedron. However, (bounded) 2-D octagon needs to be described using 8 constraints. The remaining four constraints essentially specify bounds on X_i+X_j and X_i-X_j

In order to find, the diameter in those direction, we use simple trick of rotation. A 2-D shadow can be rotated by applying following linear transformation:

$$\boxed{(X_i, X_j) \rightarrow (X_i + X_j, X_i - X_j)}$$

The above linear transformation rotates the canonical axes of a shadow by 45 degrees. After rotation, we again project the (rotated) shadow onto (new) canonical axes to obtain (lower and upper) bounds onto $(X_i + X_j, X_i - X_j)$. This gives rise to 4 more required constraints. Intuitively, this corresponds to finding the rectangular 2-D bounding box for a rotated shadow.

By intersecting the constraints for bounding box of original shadow with the bounding box for rotated shadow, we obtain 2-D octagonal (U-TVPI) over-approximation for a shadow.

Finally, iteratively finding such 2-D octagonal over-approximation for each of the $\binom{N}{2}$ shadow and intersecting the resulting constraints gives rise to the U-TVPI over-approximation for original convex-polyhedra.

3.6 Implementation

We have implemented both of the algorithms in the latest version of Integer Set Library [27]; A library for manipulating integer sets bounded by affine constraints. We use ISL's implementation of Affine form of farkas lemma to obtain the search space of U-TVPI hyperplanes. We implemented our cost model using ISL's Parametric Integer Programming solver. For Fourier Motzkin based Algorithm, we rebase our implementation using Integer FM algorithm which ISL supports currently. Our future work involves using rational FM and rational LP to achieve further scalability due to the usage of integer FM and integer LP.

3.7 Conclusions

We show limitations of state-of-art algorithm (which is used extensively in program verification libraries). We address these limitations by proposing two new algorithms, which solely uses original hyperplane representation of convex polyhedra and thereby avoiding usage of vertex enumeration algorithm. Both of our algorithms are designed in a way which guarantees computation of tightest over-approximation provided it exists. Our algorithms effectively handles cases involving parametrized and unbounded polyhedra. We provide preliminary implementation for both of these algorithms using Integer Set Library.

References

- [1] L.-N. Pouchet et al. PolyBench Benchmarks. <https://sourceforge.net/projects/polybench/>.
- [2] U. Bondhugula et al. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In PLDI. ACM, 2008 <http://pluto-compiler.sourceforge.net/>.
- [3] A. Krizhevsky et al. ImageNet Classification with Deep Convolutional Neural Networks. In Advances in Neural Information Processing Systems 25. 2012.
- [4] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* .
- [5] P. Feautrier. Dataflow analysis of array and scalar references. *IJPP* .
- [6] P. Feautrier. Efficient solutions to the affine scheduling problem. *IJPP* .
- [7] C. Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In 13th International Conference on PACT. 2004 .
- [8] C. Zhang et al. Optimizing FPGA-based Accelerator Design for Deep CNNs. In ACM/SIGDA International Symposium on FPGAs. 2015 .
- [9] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Comput.* 9, (1997) 1735–1780.
- [10] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A Fresh Approach to Numerical Computing. *CoRR* abs/1411.1607.
- [11] T. Grosser, A. Groesslinger, and C. Lengauer. Polly Performing Polyhedral Optimizations On A Low-Level Intermediate Representation. *Parallel Processing Letters* 22, (2012) 1250,010.
- [12] M. Reisinger. PolyBench Benchmarks in Julia. <https://github.com/MatthiasJReisinger/PolyBench.jl>.
- [13] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, (2013) 54:1–54:23.
- [14] H. Wong, M. M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In 2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS). 2010 235–246.

- [15] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In Proceedings of the 22Nd Annual International Conference on Supercomputing, ICS '08. ACM, New York, NY, USA, 2008 225–234.
- [16] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine Programs. In Proc. of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction, CC'10/ETAPS'10. Springer-Verlag, Berlin, 2010 244–263.
- [17] U. Bondhugula et al. PLUTO Automatic Parallelizer for CUDA, Version 0.6.2. <https://sourceforge.net/projects/pluto-compiler/files/pluto-0.6.2-cuda.tar.gz/download> 2011.
- [18] T. Grosser and T. Hoefler. Polly-ACC Transparent Compilation to Heterogeneous Hardware. In Proc. of 2016 International Conference on Supercomputing, ICS '16. ACM, USA, 2016 1–13.
- [19] T. Grosser, A. Groesslinger, and C. Lengauer. POLLY: Performing Polyhedral Optimizations On a Low-level Intermediate Representation. In Parallel Process. Lett. 22. 2012 .
- [20] A. Leung, N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin. A Mapping Path for multi-GPGPU Accelerated Computers from a Portable High Level Programming Abstraction. In Proc. of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU-3. ACM, USA, 2010 51–61.
- [21] G. Rudy, M. M. Khan, M. Hall, C. Chen, and J. Chame. A Programming Language Interface to Describe Transformations and Code Generation. In Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing, LCPC'10. Springer-Verlag, Berlin, 2011 136–150.
- [22] The Portland Group. PGI 2013 Release Notes Version 13.10. The Portland Group, 2013.
- [23] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In Proc. of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10. IEEE, USA, 2010 1–11.
- [24] A. Darte, Y. Robert, and F. Vivien. Scheduling and Automatic Parallelization. 1st edition. Birkhauser Boston, 2000.
- [25] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative Optimization in the Polyhedral Model: Part II, Multidimensional Time. In Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '08. ACM, New York, NY, USA, 2008 90–100.
- [26] S. Verdoolaege and T. Grosser. Polyhedral Extraction Tool. In Second Int. Workshop on Polyhedral Compilation Techniques (IMPACT'12). Paris, France, 2012 .
- [27] S. Verdoolaege. isl: An Integer Set Library for the Polyhedral Model. In K. Fukuda, J. Hoeven, M. Joswig, and N. Takayama, eds., Mathematical Software - ICMS 2010, volume 6327 of *Lecture Notes in Computer Science*, 299–302. Springer, 2010.

- [28] T. Grosser, S. Verdoolaege, and A. Cohen. Polyhedral AST Generation Is More Than Scanning Polyhedra. *ACM Trans. Program. Lang. Syst.* 37, (2015) 12:1–12:50.
- [29] Z. Hakura and A. Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. In Proc. of 24th Annual International Symposium on Computer Architecture, ISCA '97. ACM, USA, 1997 108–120.
- [30] M. Doggett. Texture Caches. *IEEE Micro* 32, (2012) 136–141.
- [31] NVIDIA Corporation. NVIDIA CUDA Compute Unified Device Architecture Programming Guide. NVIDIA Corporation, 2016.
- [32] J. Alglave, M. Batty, A. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson. GPU Concurrency: Weak Behaviours and Programming Assumptions. In Proc. of the 20th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15. ACM, NY, USA, 2015 577–591.
- [33] L.-N. Pouchet et al. PolyBench/C. <https://sourceforge.net/projects/polybench/> 2013.
- [34] L. Page, S. Brin, R. Motwani, and T. Winograd. PageRank Citation Ranking: Bringing Order to the Web 1999.
- [35] W.-m. W. Hwu. GPU Computing Gems Emerald Edition. 1st edition. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.
- [36] P. Micikevicius. 3D Finite Difference Computation on GPUs Using CUDA. In Proc. of 2nd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-2. ACM, USA, 2009 79–84.
- [37] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-performance Code Generation for Stencil Computations on GPU Architectures. In Proc. of the 26th ACM International Conference on Supercomputing, ICS '12. ACM, NY, USA, 2012 311–320.
- [38] R. Upadrasta and A. Cohen. Sub-polyhedral Scheduling Using (Unit-)Two-variable-per-inequality Polyhedra. In Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13. ACM, New York, NY, USA, 2013 483–496.
- [39] A. Mine. The octagon abstract domain. In Proceedings Eighth Working Conference on Reverse Engineering. 2001 310–319.
- [40] P. Feautrier. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *International Journal of Parallel Programming* 21, (1992) 313–347.