Ανάλυση Προτύπου Πρόσβασης στη Μνήμη χρησιμοποιώντας το Πολυεδρικό Μοντέλο

Memory Access Pattern Analysis using the Polyhedral Model

By

Vasiliadis Vasilis

University of Thessaly,
Department of Computer & Communication Engineering

Volos, 2012

Acknowledgments

I would like to thank my advisors Christos D. Antonopoulos and Nicolaos Bellas for their help and guidance during the development of my thesis. I would also like to thank Muhsen Owaida for providing his insight on the issues that arose during my work on the subject. Konstadinos Dalouka's provided the technical support I needed when I took over the development of SOpenCL's front-end and it was much appreciated. Finaly I would like to thank Tobias Grosser for the development of the Polyhedral Extraction Tool, if it hadn't been for it I would have to reinvent the wheel.

Memory Access Pattern Analysis using the Polyhedral Model

Vasiliadis Vasilis
vasiliad@inf.uth.gr

University of Thessaly,
Department of Computer & Communication Engineering,
Volos, Thessaly,
2012

ABSTRACT

Τα ετερογενή παράλληλα συστήματα γίνονται ολοένα πιο δημοφιλή, επειδή προσφέρουν υψηλότερη απόδοση με μικρότερο κόστος και λιγότερη κατανάλωση ισχύος. Αυτές οι αρχιτεκτονικές τυπικά συμπεριλαμβάνουν ένα σύστημα υποδοχής (κεντρικός επεξεργαστής) και ένα πλήθος επιταχυντών (hardware accelerators) στη μορφή των GPUs, DSPs, ακόμα και FPGAs. Βέβαια η μνήμη των επιταχυντών είναι εκ φύσεως κατανεμημένη· Το γεγονός αυτό απαιτεί ρητό διαμερισμό και μεταφορά της μνήμης μεταξύ των διαφορετικών στοιχείων του συστήματος. Συνήθως οι επιταχυντές έχουν περιορισμένη μνήμη σε σύγκριση με την RAM συστήματος. Επιπροσθέτως συχνά η ιεραρχία μνήμης των accelerators ελέγχεται από λογισμικό που γράφει ο εκάστοτε προγραμματιστής. Οι σημερινές ανάγκες της τεχνολογίας υπολογιστών οδηγούν σε συστήματα με αρχιτεκτονικές τύπου Non Uniform Cache Access (NUCA)· αρχιτεκτονικές οι οποίες απαιτούν προσεγμένη τοποθέτηση δεδομένων ώστε να μην βρίσκονται μακρυά από το σημείο στο οποίο αυτά χρησιμοποιούνται. Όμοια, στην περίπτωση των FPGAs, η μνήμη (BRAMs) βρίσκεται οργανωμένη ως πολλά και σχετικά μικρά νησιά δεδομένων διάσπαρτα σε διαφορετικές περιοχές του chip.

Τα παραπάνω αποτελούν ενδείξεις ότι είναι απαραίτητες εξελιγμένες μέθοδοι διαμοιρασμού και διαχείρισης της μνήμης για την βελτιστοποίηση της τοπικότητας, απόδοσης, χρήσης εύρους ζώνης, και εξοικονόμησης ενέργειας κατά την εκτέλεση ενός αλγορίθμου σε τέτοιες πλατφόρμες. Παρόλα αυτά πολλά προγραμματιστικά μοντέλα δεν εκθέτουν πλήρως το σχήμα επικοινωνίας μεταξύ των διαφορετικών τους στοιχείων. Τυπικά παραδείγματα αποτελούν τα OpenMP[Boa11], OpenCL[SGS10] καθώς σε αυτά είναι αδύνατον να περιγραφθεί η επικοινωνία μεταξύ των διαφορετικών πυρήνων (Kernels). Μάλιστα πολλοί προγραμματιστές εφαρμογών τείνουν να είναι 'άτσαλοι' στον τρόπο που χειρίζονται τα δεδομένα· προτιμούν να χρησιμοποιουν καθολικές (global) εις βάρος των τοπικών (local) buffers της OpenCL.

Η δική μας προσέγγιση στο πρόβλημα είναι η Ανάλυση Προτύπου Πρόσβασης στη Μνήμη χρησιμοποιώντας το Πολυεδρικό Μοντέλο σε εφαρμογές ανεπτυγμένες βάση του προγραμματιστικού μοντέλου OpenCL. Στατικά, κατά την διαδικασία της μετάφρασης συλλέγουμε πληροφορίες που αφορούν τα στοιχεία πινάκων τα οποία προσπελαύνονται κατά την εκτέλεση ενός πυρήνα (Kernel). Στη συνέχεια τα αποτελέσματα χρησιμοποιούνται ώστε να υποστηρίξουν την έξυπνη και έγκαιρη τοποθέτηση δεδομένων σε συστήματα που χρησιμοποιούν σταθερές ιεραρχίες μνήμης, όπως οι κάρτες γραφικών (GPUs). Για αρχιτεκτονικές που κάνουν χρήση ρευστών ιεραρχιών, όπως οι FPGAs, αυτές οι πληροφορίες

χρησιμοποιούνται από το SOpenCL [OBDA11]· ένα εργαλείο που παράγει Verilog έτοιμη προς σύνθεση έχοντας ως είσοδο πυρήνες OpenCL, χωρίς να απαιτείται αλλαγή στον κώδικά τους. Ειδικότερα, τα αποτελέσματα της ανάλυσης χρησιμοποιούνται κατά το σχεδιασμό του υποσυστήματος μνήμης και της διασύνδεσης μεταξύ των διαφορετικών πυρήνων που τοποθετούνται πάνω σε μια FPGA.

Contents

1 Introduction **1**

2 Background **2**
2.1 The Polyhedral Model 2
2.1.1 Polyhedron 3
2.1.2 Parameterized Polyhedron 3
2.1.3 Polyhedral Analysis 3
2.2 SOpenCL Framework 6
2.3 Alternatives to Polyhedral Analysis 13

3 Algorithm **14**
3.1 Usage scenaria of the Analysis Results 19

4 Future Work **20**

5 Conclusion **21**

Bibliography **22**

Chapter 1

Introduction

Heterogeneous parallel systems are becoming increasingly popular, as they offer high performance with relatively low cost and power dissipation. These architectures typically include a host system and a number of accelerators, in the form of GPUs, DSPs, or even FPGAs. Memory on such platforms is inherently distributed. This necessitates explicit data distribution and movement between system components. Typically accelerators have limited memory capacity, in comparison to system RAM. Quite often the memory hierarchy within the accelerator is also software controlled. Furthermore, massive multicore chips in the future will essentially be NUCA, necessitating careful data placement close to the compute units that will use them. Similarly, in the case of FPGAs, the on-chip memory (BRAMs) is organized as many, relatively small memory islands distributed on different areas of the chip.

The above are indicators that sophisticated data distribution and management is necessary for optimizing locality, performance, bandwidth exploitation and power efficiency. Many popular programming models, however, do not fully expose the communication pattern. Typical examples are OpenMP [Boa11], or even OpenCL [SGS10] which makes it impossible to express communication between kernels. Moreover application programmers tend to be "clumsy" in data management; for example they prefer using global over local buffers in OpenCL.

Our approach to this problem is Polyhedral Analysis for memory access pattern estimation, on applications developed with the OpenCL programming model. At compile time, we collect information about the elements of arrays that are accessed within a given kernel. The results are then used to assist the educated and timely placement of data on systems with fixed memory hierarchies, such as GPUs. On fluid architectures, such as FPGAs, this information is also exploited by SopenCL [OBDA11], a tool that produces synthesizable Verilog starting from OpenCL kernels, to assist in the design of the memory subsystem and of the interconnection network between different kernels instantiated on the FPGA.

Chapter 2

Background

2.1 The Polyhedral Model

Nested loops structures along with their program statements can be represented using the Polyhedral Model. Upon analysis of the source code Static Control Parts (*SCoP*) are identified; these are source code fragments which adhere to a certain set of rules [ACE08]:

- All variables present in the SCoP must be:
  (a) Parameter: The variable's value remains constant throughout the execution of the SCoP. or
  (b) Iterator: The variable is used as an iterator in one of the enclosing nested loops.
- Each loop's bounds must either be a) constant , or b) affine (a linear combination of loop iterators, parameters, and or constants).
  - This also applies to the conditions of if-statements.
- References to array elements should have affine combinations of parameters, iterators, and or constants.
- Data flow between statements in the loop must be explicit. Statements may not communicate with each other using shared variables invisible to the compiler.

Note: There are methods to support certain expressions like floor(), max(), min() in addition to affine combinations of variables/parameters.

For every SCoP identitied a geometric object can be constructed, we call such objects Polyhedra. Every integer point of a Polyhedron is mapped to the execution of a statement. This allows for manipulation of the programs structures through geometric transformations, while maintaining the original functionality as well as the correctness of the SCoP.

2.1.1 Polyhedron

There are two popular definitions of the Polyhedron [LW97].

(a) A Polyhedron is defined as the intersection of a finite set of closed linear half-spaces. This representation is specified by a system of equalities (matrix A, vector b) and inequalities (matrix C, vector d):

$$D : \{ \ x \in \mathbb{Q}^n \mid Ax = b, \quad Cx \geq d \ \}$$

(b) A Polyhedron can be represented, using the Minkowski representation, by a combination of lines (L), rays (R), and vertices (V):

$$D : \{ \ x \in \mathbb{Q}^n \mid x = L\lambda + R\mu + V\nu, \quad \mu, \nu \geq 0, \quad \sum_i \nu_i = 1 \ \}$$

D is defined as a linear combination of lines, a positive linear combination of unidirectional rays and a convex combination of vertices.

Both definitions are equivalent and can be constructed using their dual counterpart. However, each allows for different kinds of transformations so it is common practise to keep descriptions of a Polyhedron in both formats because it is considered expensive to go from one representation to the other.

In polyhedral analysis we claim that D is a geometric object with n dimensions, where n is the number of iterators that enclose the respective SCoP.

2.1.2 Parameterized Polyhedron

We can extend the above definitions by also supporting parameters as well as iterators and constants. A parameterized polyhedron can be described as a linear function of p, an m-vector of parameters:

$$D(p) : \{ \ x \in \mathbb{Q}^n \mid Ax = Bp + b, \quad Cx \geq Dp + d \ \}, \quad p \in \mathbb{Q}^m$$

or equivalently

$$D' : \{ \ \begin{pmatrix} x \\ p \end{pmatrix} \in \mathbb{Q}^{n+m} \quad | \quad A' \begin{pmatrix} x \\ p \end{pmatrix} = b', \quad C' \begin{pmatrix} x \\ p \end{pmatrix} \geq d' \ \}$$

Which means that the dimension of the Polyhedron D is increased by m, in order to accompany the addition of parameters.

2.1.3 Polyhedral Analysis

Polyhedral Analysis is the basis for powerful methods that are present in many popular compilers [SPJ+09], [GZA+11]. By constructing Polyhedra the compiler can identify dependencies between statements as well as reveal hidden parallelism in the source code. These can be exploited in order to produce better scheduling policies [Bas04] through compiler transformations that are based on the Polyhedral Model.

Example: Producing polyhedra starting from SCoPs

```
for( i=2; i<=n; i++ )
  for ( j=2; j<=min(m,-1+n+2)
      ; j++ )
    S1(i,j);
```

(a) SCoP

```
{
  2<=i<=n;
  2<=j<=min(m, -1+n+2)
}
```

(b) Initial set of constraints

```
{
  2<=i<=n;
  2<=j<=m;
  2<=j<=-1+n+2
}
```

(c) Final set of constraints

Optimization Passes using the Polyhedral Model

The optimization passes generally execute the following steps:

a) Identify SCoPS

This can be done by letting the developer specify which parts of the source code follow the restrictions implied by the Polyhedral Model (See Section 2.1). More advanced compilers automatically discover the SCoPS in a program by analyzing the statements present in the source code.

b) For every SCoP

Construct a Polyhedron for each statement present in the SCoP.

c) Perform transformations on the resulting Polyhedra.

d) Produce the optimized source code by extracting statements from the optimized Polyhedra.

By applying such a method automatic parallelization of an application becomes possible. When blocks of statements are identified as being able to execute in parallel the compiler modifies the scheduling of the program in order to exploit the newly discovered hidden parallelism. This leads to an increase of performance in the application without the overhead that comes with developing a multithreaded application. The resulting schedule is guaranteed to be correct because it is based on geometric transformations that are performed on the Polyhedra, whose integer points map directly to the execution of statements.

Even if parallel statements are not detected in a SCoP, it is still possible to perform optimization transformations to the source code. Overhead due to conditions present

in if/for/while statements can be eliminated at the expense of program size. Multiple copies of the statements present in the if/for/while bodies are constructed and placed in a way that respects the original schedule of the program.

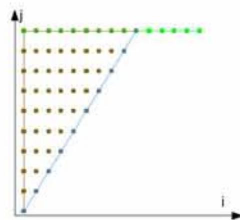
Optimization Example

For example, consider the following SCoP (code snippet 2.1).

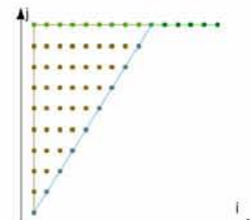
```
for ( i = 1 ; i<=max(n,m); i ++ ) {
 for ( j = 1; j<=n; j++ ) {
  if ( j == i )
   S1(i,j);
  else if ( i<=j )
   S2(i,j);

  if ( j==n )
   S3(i,j);
 }
}
```

Figure 2.1: Code Snippet

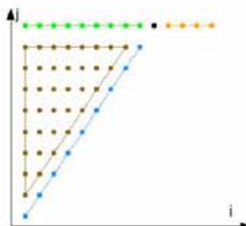
After the transformations of the Improved Quillere algorithm have been executed, the source code at code snippet 2.2 is produced. During the optimization pass the polyhedra presented at figure 2.5 are constructed.



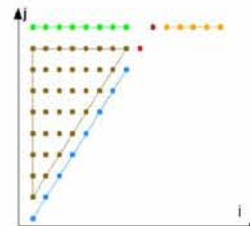
(a) Initial Polyhedra



(b) Result of the first recursion



(c) Result of the third recursion



(d) Final Polyhedra

```
for ( i=1; i<=n; i++ ) {
  S1(i,i);
  S2(i,i);

  for ( j=i+1; j<n; j++ ) {
    S2(i,j);
  }

  S2(i,n);
  S3(i,n);
}

S1(n-1, n-1);
S2(n-1, n-1);
S2(n-1, n);
S3(n-1, n);
S1(n, n);
S2(n, n);
S3(n, n;

for ( i=n+1; i<=m; i++ ) {
  S3(i, n);
}
```

Figure 2.2: Optimized Code Snippet

In the above code snippet, condition checking overhead in if-statements has been removed. In its place we find blocks of statements, this means that the resulting code size has increased but the execution time of the source code has been downsized. During Polyhedral Analysis the compiler may trade code size for code performance, most of the time this is desired.

However in cases where code size is an important factor the very same techniques can still be used, they just need to be tuned in order to avoid working to such great extents, which is a trivial task for the compiler developer.

2.2 SOpenCL Framework

The use of heterogeneous hybrid paraller platforms to accelerate the execution of algorithms is steadily increasing during the past few years. Because hardware accelerators can drastically improve the performance of the application while keeping the cost and power consumption rate low.

There are some facts which indicate that designing hardware is harder than devel-

oping a software implementation of the same algorithm. This claim is supported by the vast number of tools available to software engineers that assist the developing as well as the debugging process. Additionally the number of hardware designers is less than the number of software engineers.

The ideal scenario would be a programming model that provides the advantages of an algorithm implemented on hardware over software without the overhead that comes with designing hardware accelerators. A Programming Model that allows the developer to make use of multiple execution devices present on a platform is OpenCL; below we present a brief introduction to the OpenCL language.

OpenCL

OpenCL [SGS10] is a programming model designed in order to ease the development of task-parallel/data-parallel computations in a heterogeneous computing environment consisting of the host CPU and any attached OpenCL execution devices.

The devices may not share memory with the host CPU and typically have a different instruction set; due to these facts the OpenCL API assumes heterogeneity between the host and all devices present on the system. It provices functions to:

- enumerate the avalaible devices

    CPUs, GPUs, Hardware Accelerators

- manage contexts containint the devices to be used
- perform host-device memory transfers
- compile OpenCL programs and kernel functions to be executed on target devices
- launch kernels on target devices
- query execution process
- check for errors

The programming model abstracts CPUs, GPUs, and other accelerators as devices that contain one or more compute units composed of one or more SIMD processing elements that execute instructions in lock-step. For example, in the context of a device being a CPU a compute unit is a CPU core.

Four types of memory are available for the OpenCL devices:

a) Global
large memory with high latency.

b) Constant
low-latency but small and read-only memory.

c) Local
   accessible from multiple processing elements withing the same compute unit.

d) Private
   memory or device registers accessible within each processing element only.

When using the OpenCL programming model in order to achieve optimum results regarding the performance of the application the nature of each kernel should be taken into account before it is assigned for execution on a specific OpenCL device. For example, massively data-parallel kernels should be executed on GPUs.

An example C code for the Multiple Debye-Huckel method is presented in figure 2.3, the equivalent source using the OpenCL Programming Model is shown in figure 2.4.

```c
for ( int igrid=0; igrid<ngrid; igrid++ ) {
 float v=0.0f;

 for ( int jatom=0; jatom<natoms; jatom++ ) {
  dx = gx[igrid] - ax[jatom];
  dy = gy[igrid] - ay[jatom];
  dz = gz[igrid] - az[jatom];

  dist = sqrt(dx*dx + dy*dy + dz*dz);

  v += pre1 * (charge[jatom] / dist)
      * exp(-xkappa * (dist-size[jatom]))
      / (1.0f+xkappa*size[jatom]);
 }
 val[igrid]=v;
}
```

Figure 2.3: MDH source code written in C

```
__kernel void mdh(__global float *ax, __global float *ay,
        __global float *az,
        __global float *charge, __global float *size,
        __global float *gx, __global float *gy,
        __global float *gz,
        __global float *val, float pre1, float xkappa, int
           atoms ) {
  int igrind = get_global_id(0);

  float v = 0.0f;
  float dx, dy, dz, dist;

  for ( int jatom = 0; jatom < natoms; jatom++ ) {
    dx = gx[igrid] - ax[jatom];
    dy = gy[igrid] - ay[jatom];
    dz = gz[igrid] - az[jatom];

    dist = sqrt(dx*dx + dy*dy + dz*dz);

    v += pre1 * (charge[jatom] / dist)
         * exp(-xkappa * (dist-size[jatom]))
         / (1.0f+xkappa*size[jatom]);
  }
  val[igrid] = v;
}
```

Figure 2.4: MDH source code written in OpenCL

Silicon OpenCL [OBDA11] attempts to address the problem of harnessing the power of hardware accelerators without having to directly deal with hardware design. Its approach is to produce synthesizable Verilog starting from unmodified OpenCL kernels.

Figure 2.5: Visual representation of the SOpenCL transformation process.

In order to achieve this the framework is split into two tools, the Front-End and the Back-End.

Front-End

The Front-End produces a C-Function by performing code transformations on OpenCL kernels. The resulting code encloses the computations that will be exeuted by a single work-group. In order to execute the total workload of the original OpenCL kernel the C-Function has to be executed multiple times, in order to cover all the work-groups.

The transformations that directly change the form of the source code that is received as the front-end's output are the following:

a) Constructing Triple Nested Loops:
The front-end coarsens the parallelism of the kernel's body by enclosing it in a triple nested loop. This tripple nested loop represents the execution of a 3D partitioned set of threads, in other words a work-group.

   In the event of synchronization barriers, multiple triple nested loops may be present in the C-Function.

b) Variable Privatization:
Since the execution of work-items is "serialized" in the context of the output source code, when a barrier is present in the source code special action needs to be taken in order to ensure the correct execution of the algorithm.
After analysis of the statements new arrays are introduced to the code. They function as temporary storage in order to maintain the values for every work-item accross the barrier.

Back-End

The Back-End parses the Front-End's output and produces synthesizable Verilog. For a given Kernel multiple hardware accelerator instances will be instantiated on a single FPGA.

Before the actual process which generates Verilog, the back-end's input undergoes a series of LLVM Optimization passes:

a) Predication

b) Code Slicing

c) Swing Modulo Scheduling

Afterwards the template based back-end proceeds to generate the Hardware Accelerator, along with testbenches. The full extent of the process can be seen at figure 2.6.

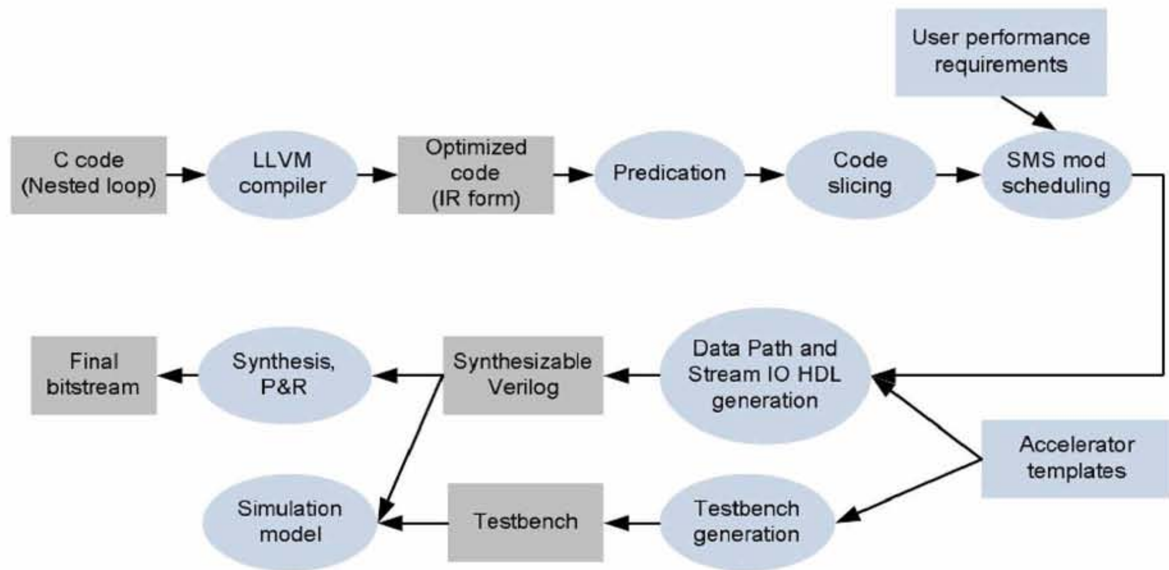


Figure 2.6: Hardware generation tool flow.

Why Polyhedral Analysis is a good choice

Below the typical output of the front-end is presented 2.7. The general format of the C source code is a series of triple nested loops. This greatly resembles a SCoP, as long as the statements enclosed in the loops are proven to be legal in the context of a SCoP, the resulting code is fit for polyhedral analysis.

```
void function(int __global_id_x, int __global_id_y,
          int __global_id_z,
          int __global_size_x, int __global_size_y,
          int __global_size_z,
          int __work_group_id_x, int __work_group_id_y,
          int __work_group_id_z,
          int __local_size_x, int __local_size_y,
          int __local_size_z)
{
 unsigned int __kernel_i, __kernel_j, __kernel_k;
 unsigned int *__privatized_array1, ... ;

 for (__kernel_k = 0; __kernel_k < __local_size_z;
    __kernel_k++) {
  for (__kernel_j = 0; __kernel_j < __local_size_y;
     __kernel_j++) {
   for (__kernel_i = 0; __kernel_i < __local_size_x;
      __kernel_i++) {
    < statement >
    < statement >
       ...
   }
  }
 }

 // barrier that has been removed

 for (__kernel_k = 0; __kernel_k < __local_size_z;
    __kernel_k++) {
  for (__kernel_j = 0; __kernel_j < __local_size_y;
     __kernel_j++) {
   for (__kernel_i = 0; __kernel_i < __local_size_x;
      __kernel_i++) {
    < statement >
    < statement >
       ...
   }
  }
 }


}
```

Figure 2.7: Typical Front-End output

2.3 Alternatives to Polyhedral Analysis

The Polyhedral Model imposes strict rules regarding which blocks of code can be categorized as fit for analysis. However, once we get around this obstacle Polyhedral Analysis is the prefered choice over the alternative.

a) **Profiling**
   Instead of using this mathematical set of rules one can use profiling tools to estimate the memory access pattern of an application. This requires a large ammount of test-runs and as a result takes a lot of time in order to get accurate results.

b) **Simulate the memory accesses**
   Another method of memory access pattern analysis is to parse the code and extract all possible combinations of variable values that are legal in the context of the application. Then the execution of the application is simulated and the resulting memory accesses are recorded. This is an accurate method but it is a really slow process because of its overly pedantic nature.

   When dealing with large source code files the compiler performance becomes an important issue. Thus, choosing the Memory Access Pattern analysis method should take into account the time it takes to complete.

After considering the above cases, as well as the fact that there are ways to create polyhedra out of (very specific) non affine combinations of variables, using the Polyhedral Model to create a Memory Access Analysis pass becomes an attractive choice.

Chapter 3

Algorithm

We are using the Polyhedral Extraction Tool [VG12] to create polyhedra starting from the output of the back-end, which is a pure C-Function. Below we present our method, followed by an example:

a) The front-end is executed, using an unmodified OpenCL kernel as input.

b) Its output, a pure C-Function, is parsed by the PET tool.

c) Throughout the Polyhedral Analysis step, we keep a record of which polyhedra represent memory accesses. We categorize them to read/writes.

   If a polyhedron cannot be constructed (due to non-affine constraints) for a specific read (write) of an array, all polyhedra regarding reads (writes) to that specific are dropped; this also includes any future polyhedra representing reads (writes) to that array's elements. We do not wish to have a partial view regarding the accesses to any array.

d) When all the polyhedra are generated, they are exported to a file.

e) The output file is read by the Memory Access Pattern analysis tool. Which performs the following steps:

   1) Generate as many copies of the polyhedra, as the number of the kernel instances that will be generated on the FPGA.
   2) For each different kernel instance inject additional constraints to the set of polyhedra related to that specific slice of the computation workload.
   3) Use the Loechner & Wilde [LW97] method to discover the parametric vertices of the polyhedra which were generated in the previous step.
   4) Finally, output the ranges of the elements which are being accessed to a file; include additional information regarding the type of the acces, read or write.

Memory Access Pattern Analysis Example

Consider the following source code (figure 3.1), written in OpenCL.

```
__kernel void matrixMultiply(__global float *c,
     __global const float* a, __global const float *b,
     unsigned int size)
{
 int x= get_global_id(0);
 int y = get_global_id(1);
 float element = 0;
 int i;

 for (i=0; i<size; i++)
   element += a[i + y*size] * b[x +i*size];

 c[x + y*size] = element;
}
```

Figure 3.1: OpenCL Kernel on which to perform Memory Access Pattern Analysis

The output of SOpenCL's front-end tool is the C-Function presented at figure 3.2.

```
__kernel void matrixMultiply( /*parameters are omitted on
   purpose*/ )
{
  unsigned int __kernel_i, __kernel_j, __kernel_k;

  unsigned int x, y,i;
  float element;

  for (__kernel_k = 0; __kernel_k < __local_size_z;
      __kernel_k++) {
    for (__kernel_j = 0; __kernel_j < __local_size_y;
        __kernel_j++) {
      for (__kernel_i = 0; __kernel_i < __local_size_x;
          __kernel_i++) {
        x = (__global_id_x + __kernel_i);
        y = (__global_id_y + __kernel_j);
        element = 0;

        for (i = 0; i < 1000; i++)
          element += a[i + y * 1000] * b[x + i * 1000];

        c[x + y * 1000] = element;
      }
    }
  }
}
```

Figure 3.2: Output of the front-end for the Kernel at figure 3.1

The polyhedra generated for this algorihm are presented in table 3.1.

Read/Write	Array	Constraints
R	a	$\{x = __global_id_x + __kernel_i,$ $y = __global_id_y + __kernel_j,$ $0 \leq __kernel_i < __local_size_x,$ $0 \leq __kernel_j < __local_size_y,$ $0 \leq __kernel_k < __local_size_z,$ $0 \leq __global_id_y \leq __global_size_y,$ $0 \leq i < 1000,$ $i + y * 1000 \geq 0\ \}$
R	b	$\{x = __global_id_x + __kernel_i,$ $y = __global_id_y + __kernel_j,$ $0 \leq __kernel_i < __local_size_x,$ $0 \leq __kernel_j < __local_size_y,$ $0 \leq __kernel_k < __local_size_z,$ $0 \leq __global_id_x < __global_size_x,$ $0 \leq i < 1000,$ $x + i * 1000 \geq 0\ \}$
W	c	$\{x = __global_id_x + __kernel_i,$ $y = __global_id_y + __kernel_j,$ $0 \leq __kernel_i < __local_size_x,$ $0 \leq __kernel_j < __local_size_y,$ $0 \leq __kernel_k < __local_size_z,$ $0 \leq __global_id_x < __global_size_x,$ $0 \leq __global_id_y \leq __global_size_y,$ $x + y * 1000 \geq 0\ \}$

Table 3.1: Polyhedra representing the memory accesses

The next step of the algorithm is to split the working data set to n smaller ones; one for each instance of the Kernel. The resulting polyhedra are presented in table 3.2.

Read/Write	Array	Constraints
R	a_j	$\{x = __global_id_x + __kernel_i,$ $y = __global_id_y + __kernel_j,$ $0 \leq __kernel_i < __local_size_x,$ $0 \leq __kernel_j < __local_size_y,$ $0 \leq __kernel_k < __local_size_z,$ $j * (__global_size_y/n) \leq __global_id_y\ ,$ $__global_id_y < (j+1) * (__global_size_y/n)\ ,$ $0 \leq i < 1000,$ $i + y * 1000 \geq 0\ \}$
R	b_j	$\{x = __global_id_x + __kernel_i,$ $y = __global_id_y + __kernel_j,$ $0 \leq __kernel_i < __local_size_x,$ $0 \leq __kernel_j < __local_size_y,$ $0 \leq __kernel_k < __local_size_z,$ $0 \leq __global_id_x < __global_size_x,$ $0 \leq i < 1000,$ $x + i * 1000 \geq 0\ \}$
W	c_j	$\{x = __global_id_x + __kernel_i,$ $y = __global_id_y + __kernel_j,$ $0 \leq __kernel_i < __local_size_x,$ $0 \leq __kernel_j < __local_size_y,$ $0 \leq __kernel_k < __local_size_z,$ $0 \leq __global_id_x < __global_size_x,$ $j * (__global_size_y/n) \leq __global_id_y,$ $__global_id_y < (j+1) * (__global_size_y/n),$ $x + y * 1000 \geq 0\ \}$

Table 3.2: Polyhedra split over n different kernel instances

The final results of the algorithm are presented in table 3.3. The format X_j indicates accesses to array X in the context of the j_{th} instance of the kernel on the FPGA.

Read/Write	Array	Range
R	a_j	$[\ (\frac{j}{n}) * (__global_size_y * 1000),\ (\frac{j+1}{n}) * (__global_size_y * 1000)\ )$
R	b_j	$[\ 0,\ __global_size_x + 999 * 1000\ )$
W	c_j	$[\ (\frac{j}{n}) * (__global_size_y * 1000),$ $((\frac{j+1}{n}) * __global_size_y - 1) * 1000) + __global_size_x\ ]$

Table 3.3: Ranges of the elements that are read/written by each kernel instance

If we merge all ranges X_j (where $0 <= j < n$) we end up with information regarding the accesses on array X throughout the kernel.

3.1 Usage scenaria of the Analysis Results

The output of the Memory Analysis is the array access pattern in the form of element ranges. For every range there it is also indicated whether the elements being accessed are read or written. Having such information about which parts of the arrays can be used during the hardware generation as well as during the execution of the application.

Synthesizing Memory Hierarchies based on Memory Access Pattern Analysis

Memory in FPGAs is distributed in the form of memory islands (Block RAMS), its size is much smaller than System RAM. These facts make FPGA memory a resource that has to dealt with very carefully. Since the cost of accessing memory that lies far from the source of the request is high it is desired to place the data close to the logic area that they are used/generated. Fortunately, the back-end can design the Memory Hierarchy of the FPGA by using insight which is derived from the Memory Access Pattern Analysis results.

Minimizing the overhead of transferring data between Execution Devices

The OpenCL Language enables the developer to access a number of execution devices on a platform. In order for this programming model to function the runtime system manages the data transfer between the different components of the platform. In the case of SOpenCL execution devices may not be always present on the same physical platform, in such circumstances transfering data over the network increases the cost of the operation. Using the analysis results the runtime may chose to only send fractions of the original dataset to each execution device thereby decreasing the memory transfer overhead.

Estimate the memory usage of Applications

Even though computer technology has evolved to the point where the average developer is not really limitted by the memory present on the system, it is still desired to know the memory footprint of an application. Using our Memory Analysis method the developer can receive a pretty accurate estimation of the amount of memory his/her algorithm works with.

Chapter 4

Future Work

Improve Accuracy of the Results

Currently our method functions under the assumption that all polyhedra generated are convex; that is between any two integer points of a polyhedron all integer points between are also included in the same polyhedron, in other words the polyhedron cannot contain any "holes". This may result in elements being reported as accessed just because they happen to exist in said "holes"; even though the method is not 100% accurate it will always report an element if an access to it exists.

We are currently investigating ways to either handle non-convex polyhedra or split the polyhedra to sets of convex polyhedra and then gather the results.

Timely Placement of Data

The Memory Access Pattern Analysis provides us a fairly accurate map of the areas in the memory that are being accessed by a given kernel instance. If we take into account the underlying hardware we could devise a method to automatically move memory to the device just before the data is actually required. This way an automatic profiling mechanism can be implemented to further improve the performance of the hardware.

Automatic Tiling Optimization Pass

Finaly since the access pattern can now be discovered we could implement an additional Clang pass to automatically modify the syntax tree of an algorithm in order to transform it to a tiling algorithm. This will improve the locality of memory and provide furthet improvement to the application performance.

Chapter 5

Conclusion

Using hardware generators to speed up the performance of applications is an ever increasing trend in modern high performance computing. SOpenCL attempts to automate the process of generating said hardware accelerators; our memory access pattern analysis algorithm facilitates the use of complex memory hierarchies in heterogeneous parallel systems with multiple accelerators, by

a) placing data on the chip close to where they are being used, and

b) reducing memory traffic overhead by being selective about the memory segments that are transferred between the different components of the system

Bibliography

[ACE08] ACE. Parallelization using Polyhedral Analysis. *ACE Associated Compiler Experts*, March 2008. 2

[Bas04] Cédric Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In *IEEE PACT*, pages 7–16. IEEE Computer Society, 2004. 3

[Boa11] OpenMP Architecture Review Board. OpenMP Specifications. 2011. iiiiii, 1

[GZA+11] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L. N. Pouchet. Polly - Polyhedral Optimization in LLVM. IMPACT, 2011. 3

[LW97] Vincent Loechner and Doran K. Wilde. Parameterized Polyhedra and Their Vertices. *International Journal of Parallel Programming*, 25(6):525–549, 1997. 3, 14

[OBDA11] Muhsen Owaida, Nikolaos Bellas, Konstantis Daloukas, and Christos D. Antonopoulos. Synthesis of Platform Architectures from OpenCL Programs. In Paul Chow and Michael J. Wirthlin, editors, *FCCM*, pages 186–193. IEEE Computer Society, 2011. iv, 1, 9

[SGS10] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science and Engineering*, 12(3):66–73, 2010. iiiiii, 1, 7

[SPJ+09] J. Sjödin, S. Pop, H. Jagasia, T. Grosser, and A. Pop. Design of Graphite and the Polyhedral Compilation Package. GCCSummit, 2009. 3

[VG12] Sven Verdoolaege and Tobias Grosser. Polyhedral Extraction Tool. IMPACT, January 2012. 14