Automating data management in heterogeneous systems using polyhedral analysis

Vassilis Vassiliadis
University of Thessaly, Greece
vasiliad@inf.uth.gr

Christos D. Antonopoulos
University of Thessaly, Greece
cda@inf.uth.gr

George Zindros
University of Thessaly, Greece
zindros@inf.uth.gr

ABSTRACT

In this paper we introduce a framework which automates the task of data management for OpenCL programs across multiple devices of a heterogeneous system. Our approach uses compile-time analysis, based on the polyhedral model, to associate computations with the data they consume / produce. The results of the analysis are then used by a runtime system which automates the task of data management. Beyond alleviating the programmer from the burden of data management, our framework enables partitioning computations to all computational devices of heterogeneous systems according to the computational power and memory capacity of each device, thus facilitating the exploitation of all computational and memory resources of the system.

We evaluate our approach on a system containing a multicore CPU and 4 GPUs, using a set of OpenCL applications and benchmarks. We find that our framework allows the transparent utilization of all heterogeneous resources with negligible overhead (1.24% on average over hand-mapped to the target system versions of the codes). At the same time, it enables the execution of problem sizes which could not be executed on homogeneous, or less complex heterogeneous systems, due to their high computational and memory requirements.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.4 [**Programming Languages**]: Processors - Code generation, Compilers,Optimization, Run-time environments

General Terms

Algorithms, Performance

Keywords

Polyhedral Analysis, Memory Access Pattern Estimation, OpenCL, Run-time

PCI2015 October 01 - 03 2015, Athens, Greece
.

1. INTRODUCTION

Heterogeneous systems gain popularity in high performance computing due to the performance they offer, combined with exceptional power efficiency. However, on such systems the programmer is not only responsible for expressing and exploiting parallelism, but has also to invest significant effort for memory management across devices, address spaces and often levels of the memory hierarchy within devices. To make things worse, heterogeneous systems can vastly differ in terms of configuration, often necessitating significant rewriting whenever applications need to be ported to another system.

Although sophisticated data distribution and management is necessary, it is clearly worth investing in automated techniques that alleviate this additional burden from programmers. In this paper we introduce a framework which automates the task of data management for OpenCL programs across multiple devices of a heterogeneous system. Our approach applies compile-time analysis, based on the polyhedral model, to associate computations with the data they consume / produce. The results of the analysis are then used by a runtime system which automates the task of data management. Beyond automating the task of data management, our framework enables partitioning computations to all computational devices of heterogeneous systems according to the computational power and memory capacity of each device, thus facilitating the exploitation of all computational and memory resources of the system.

Our contributions are (a) a combined compile- and runtime methodology to estimate the association of data to computations, (b) an implementation for OpenCL programs, automating data management and allowing kernel partitioning and execution on multiple devices, and (c) a performance evaluation of our automated approach, as well as a comparison against NVidia CUDA Unified Memory, an industrial solution to the problem of automating data management for GPU-based heterogeneous systems.

We find that our methodology introduces negligible overhead of 1.24% on average compared with programmer controlled data management. It significantly outperforms NVidia Unified Memory which, on the same set of codes, results to an average overhead of 614%. At the same time, it enables the execution of problem sizes that could not be tackled by homogeneous or less complex heterogeneous systems, due to their high computational and memory requirements.

The rest of the paper is organized as follows: We briefly introduce the polyhedral model in Section 2. In Section 3 we present our algorithm and in Section 4 we evaluate

our approach and compare it against NVidia CUDA Unified Memory (UM). Finally, we discuss related work in Section 5, followed by our conclusions in Section 6.

2. POLYHEDRAL MODEL

The polyhedral model (also called polytope model) is a mathematical framework mostly used for loop nest analysis in compiler optimization. The polyhedral method treats each loop iteration within nested loops as lattice points inside mathematical objects called polyhedra (polytopes). Transformations on the polyhedra translate to source code transformations of the represented source code tree. This characteristic makes polyhedral analysis a great toolkit for compiler optimizations, because it provides the means to easily manage and reform loops.

2.1 Polyhedron

A n-polyhedron (or polytope) is a geometric object with flat sides in the N-Dimensional space. Formally, a polyhedron domain (D) can be thought of as the intersection of a finite set of closed linear half-spaces. This representation is specified by a system of equalities and inequalities:

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

2.2 Polyhedral Analysis

Polyhedral analysis is based on representing nested loop structures, along with their program statements, in the form of polyhedra. It is the basis of powerful methods present in popular compilers like LLVM/CLANG [6], and GCC [11]. By constructing and transforming polyhedra the compiler can identify dependencies between statements, produce optimized instruction schedules and discover parallelism [6, 11, 1, 2].

Upon analysis of the input source code, Static Control Parts (SCoPs) are identified. SCoPs are source code snippets that hold a set of characteristics necessary in order for polyhedral analysis to be applicable:

- All variables in the SCoP are either *iterators* used in the underlying nested loops, (b) *parameters* to the SCoP, or (c) affine expressions of the previous two. *Parameters* maintain the same value throughout the execution of the SCoP.
- Loop bounds must be either (a) constants, or (b) affine combinations of loop iterators, parameters and constants. This applies to the conditions of conditional statements as well as expressions used to index arrays.
- All data flow between statements in the loop must be explicit. In other words, statements may not communicate via shared variables invisible to the compiler.

Each polyhedron point corresponds to the execution of a statement or, in the context of this paper, to the access of a variable. This characteristic enables the compiler to perform manipulation of program structures through polyhedral transformations, while maintaining the original functionality as well as guaranteed code equivalence.

2.3 Polyhedral analysis example

We show a simple example of applying polyhedral analysis to the small SCoP of Listing 1, which consists of two nested loops. This SCoP is translated to the *constraints*:
$2 \leq i \leq min(M, -1 + N + 2)$. These can be further refined to the following: $2 \leq i \leq M$, and $2 \leq i \leq N + 1$.

```
1 for ( i=2; i<=min(M,-1+N+2); ++i ) {
2     S1(i);
3 }
```

Listing 1: Simple SCoP

A polyhedron can be represented by a matrix with $(1 + Output + Input + Parameters + 1)$ columns and as many rows as the number of constraints which form the polyhedron. The first column indicates whether the row describes an equality or inequality (value 0 or 1 respectively). In the context of this paper output dimensions indicate array indexes, in the example below it indicates the value range of iterator i, the only output dimension is the actual value of i. Input dimensions are essentially iterators of the SCoP. A set of column vectors follow it, one for each iterator present in the polyhedron domain. The next set of column vectors represent the parameter coefficients. The final column vector stores the constant component of the affine constraints. The matrix representation of the polyhedron describing the possible values of iterator i in Listing 1 is:

$$\begin{bmatrix} 0 & -1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & -2 \\ 1 & 0 & -1 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 & 1 & 1 \end{bmatrix}$$

If, for example, one would like to add the additional constraint $2 * i + 3 * N + 4 * M - 1 \geq 0$, the matrix would be extended with the following row (*constraint*)

$$\begin{bmatrix} 1 & 0 & 2 & 3 & 4 & -1 \end{bmatrix}$$

3. MEMORY ACCESS PATTERN ANALYSIS AND BUFFER MANAGEMENT

In this Section we introduce a methodology for identifying the memory footprint of computations and automating the data transfers between devices of a heterogeneous system. This also enables semi-automatically partitioning parallel work at chunks of different – potentially finer – granularity than the one specified by the programmer, in order to fully exploit all available accelerators. Apart from enabling parallel execution of chunks on different devices, our approach identifies the exact subset of data used by each chunk, thus allowing execution on devices with limited memory. In the context of this paper we apply our approach on OpenCL kernels. We refer to kernels executing fractions of the total computation workload as sub-kernels.

Our methodology consists of two phases. Polyhedral analysis is used in the compile-time, offline phase, to produce parametric estimations of the data ranges accessed by each sub-kernel, as well as transformations to the original code. At the run-time phase, after the parameters are known, data are automatically partitioned and transferred – in an optimal, coalesced manner – to/from devices.

Throughout this section we use part of a Hybrid, Monte-Carlo Partial Differential Equation (Hybrid PDE) solver [12] (MC). MC is explained in detail in Section 4.

3.1 Offline phase

To develop the offline phase we use the Polyhedral Extraction Tool (PET) [14], ISL [13] and PolyLib [15] frameworks. PET produces a polyhedral model from C source code. ISL

```
kernel void DoRandomWalks2D(
  global float *D,  global float *x, global float *result,
  unsigned int num_walks, float btol,  unsigned int nodes)
{
  private long  me = get_local_id(0), us = get_local_size(0);
  private long __group_id_x = get_group_id(0);
  /* Some variable declarations and statements omitted */
  if ( __group_id_x < nodes )
  {
    _x[0] = x[__group_id_x*2];
    _x[1] = x[__group_id_x*2+1];
    for ( i=0; i<num_walks; ++i )
    {
      _x[0] = x[__group_id_x*2];
      _x[1] = x[__group_id_x*2+1];
      perform_random_walks(num_walks, _x, _D);
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    if ( me == 0 )
    {
      d = compose_d();
      result[__group_id_x] = d;
    }
  }
}
```

Listing 2: Original MC kernel.

```
/* ocl_offsets: Holds the offset values (Dynamic mode) */
kernel void DoRandomWalks2D( constant const int *ocl_offsets,
  global float *D,  global float *x, global float *result,
  unsigned int num_walks, float btol,  unsigned int nodes)
{
  private long  me = get_local_id(0), us = get_local_size(0);
  private long __group_id_x = get_global_id(0)/us;
  /* Some variable declarations and statements omitted */
  if ( __group_id_x < nodes )
  {
    _x[0] = x[(__group_id_x-x0y)*x0w -x0o];
    _x[1] = x[(__group_id_x-x1y)*x1w +1 -x1o];
    for ( i=0; i<num_walks; ++i )
    {
      _x[0] = x[(__group_id_x-x2y)*x2w -x2o];
      _x[1] = x[(__group_id_x-x3y)*x3w +1 -x3o];
      perform_random_walks(num_walks, _x, _D);
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    if ( me == 0 )
    {
      d = compose_d();
      result[__group_id_x -result2o] = d;
    }
  }
}
```

Listing 3: Modified MC kernel.

is a library for manipulating quasi-affine sets and relations, providing tools for powerful and compact representation of polyhedra. Finally, PolyLib is a library for manipulating parametric polyhedra.

The output of the offline algorithm is a set of parametrized ranges for all array accesses in a sub-kernel. Each range is also tagged with information to describe the type of each access: *read* or *write*. The rest of this subsection will focus on the offline phase of our algorithm which is illustrated in Figure 1.a.

3.1.1 Array indexes de-linearization

We begin the analysis by examining each statement which involves accesses to array elements. Our framework handles 1D and 2D accesses as long as they follow the linearized $array[row*width+column]$ notation. However, as specified

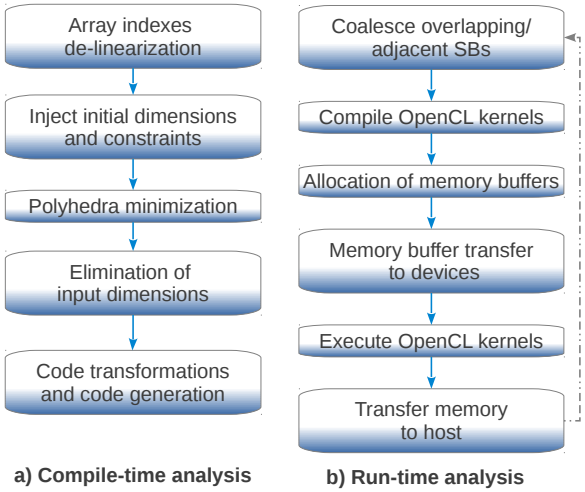

Figure 1: Proposed Hybrid methodology.

in section 2.2 the linearized notation is "illegal" for polyhedral analysis in the general case because it may not be affine. Consequently such accesses cannot be directly represented in the polyhedral model. We overcome this by extending PET to de-linearize array accesses, considering *row* and *column* as independent indexes and *width* as an extra parameter.

3.1.2 Inject initial dimensions and constraints

At a higher level, an OpenCL kernel can be viewed as the innermost body of a six-level nested loop which represents the kernel execution geometry. The three innermost loops correspond to the execution of work-items within a work-group, whereas the three outer loops correspond to the execution of work-groups within a grid. The geometry information however, is not explicitly defined within the kernel source code. In this step, we introduce the kernel execution geometry constraints to the polyhedra produced by PET to represent each memory access.

3.1.3 Polyhedra minimization

This step projects out all superfluous constraints, namely constraints not necessary for defining the specific memory access represented by each polyhedron. Removing those constraints, corresponding to unnecessary dimensions of the polyhedron, involves recursively identifying them and finally projecting them out using the Fourier-Motzkin elimination process [16] offered by ISL. This step is not necessary for the correctness of our analysis, however it significantly reduces its execution time.

3.1.4 Elimination of input dimensions

The next step of the algorithm is to project out all input dimensions using the Fourier-Motzkin elimination process. All constraints related to those dimensions are converted to constraints related to parameters. Parameters are defined at run-time and have constant value throughout the execution of the kernel (which is not the case for input dimensions). Therefore memory accesses defined using parametric expressions are in-ambiguous at execution time. All resulting polyhedra consist solely of output and parameter dimensions.

3.1.5 Code transformations and generation

As mentioned earlier, the goal of our methodology is to partition the original computation to smaller chunks, and associate each chunk with the exact subset of the original input or output data required for or produced by its execution. Data access indexes have, therefore, to be rewritten, in order to correctly map to the specific, smaller buffers accompanying each chunk of computation.

More specifically, we modify the source code so that accesses are now of the form: $array[(row - offset\_row) * effective\_width + column - offset\_col]$. The offsets $offset\_row$ and $offset\_col$, as well as the $effective\_width$ take constant values after the online phase of our analysis, namely at execution time.

The modifications required for the MC kernel are shown in listing 3. Changes to the original code appear in red.

Finally, a new function is generated, which calculates the buffer ranges touched by each memory access in the produced chunks of computation, based on the original kernel parameters and kernel execution geometry. This function will be invoked by the runtime once per kernel execution before the data allocation and management mechanisms.

3.2 Online phase

The online phase, depicted in Figure 1.b, takes place at execution time, right before the compilation of the OpenCL kernels[1]. The function discussed in Section 3.1.5 is executed to generate the access ranges for each OpenCL device.

At this point we introduce two terms:

Memory Access Region: A *MAR* is extracted for each polyhedron. It is the memory region which contains the positions touched by a specific memory access instruction, as represented by the respective polyhedron.

Sub-Buffer: A *SB* is a memory region which will be allocated on a compute device and contains at least one MAR. It can be visualized as the bounding box of a set of MARs.

3.2.1 Coalescing of overlapping/adjacent SBs

During this step overlapping or adjacent SBs are coalesced to allocate the required memory on the devices.

When memory coalescing takes place, two SBs will be joined together. The $effective\_width$ of the MARs contained within the SB will be computed. For example consider the Figure 2: two overlapping SBs each one consisting of a single MAR are illustrated. The first one involves an access to a 2D matrix which accesses rows 20 to 120 and columns 30 to 130. The second 2D SB comprises a MAR which involves rows 10 to 110 and columns 50 to 150. These two $100x100$ SBs overlap and are therefore candidates to our memory coalescence scheme. If the bounding box of the SBs contains fewer elements than the sum of elements in each access then the memory coalescence is performed. The idea is to merge SBs which are mapped to neighbouring or overlapping memory regions. In this particular case, a coalescence will provide a compression ratio of $\frac{2*(100*100)}{110*120} \approx 1.52X$ compared to allocating two separate memory regions, one for each SB. The $effective\_width$ of the MARs in the resulting SB is 120.

3.2.2 Allocation of memory buffers

After all memory coalescing opportunities are explored, the final memory buffer size for each array used in the OpenCL sub-kernel is allocated. This involves placing the output SBs of the previous step in consecutive memory regions.

[1]In OpenCL, kernels are compiled just-in-time, i.e. device binaries are produced at execution time.

```
1  long foo(int array[], int width)
2  {
3    int row, column;
4    long dummy = 0;
5
6    for (row=10; row<120; ++row )
7      for (column=30; column<130; ++column)
8        dummy += array[row*width+column]
9                 * array[(row+10)*width+(column+20)];
10
11   return dummy;
12 }
```

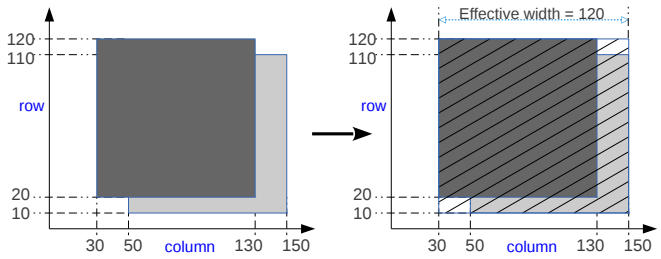

Figure 2: Memory coalescence example.

3.2.3 Memory buffer transfer to devices

Memory is transferred from the host to the devices if a sub-kernel executes for the first time or the indexes accessed by it have changed since its last invocation. This way, data which should only be copied once on the device and used multiple times, across consecutive executions of the sub-kernel, are retained.

3.3 OpenCL Code transformation issues

Our approach supports two modes: *Static* and *Dynamic*. *Static* should be used for kernels whose parameters remain constant between subsequent invocations. Offsets are injected into the kernel at the moment of just-in-time compilation using **#define** directives to minimize the performance overhead of index computations. *Dynamic* targets cases where the thread topology or kernel arguments change between consecutive kernel invocations. This mode, only compiles each kernel once and uses a vector of offsets instead of static **#define** directives. However, each read/write access on a vector is penalized with three extra reads which acquire the offset information, as well as two extra subtractions to apply the offsets. Common sub-expression elimination reduces this overhead to the lowest possible.

4. EVALUATION

For our experimental evaluation we used an Intel(R) Core (TM) i7-4820K CPU clocked at 3.70GHz and four NVIDIA GeForce GTX 680 GPUs. The system RAM is 32 GB and each GPU has 2.0 GB on-device memory. We experimented with 4 benchmarks. MC is an alternative method of approaching multi-domain, multi-physics problems. It enables the solution of a PDE in a subdomain of the original problem. The core kernel of this application performs random walks from points of the subdomain boundary to the boundary of the original domain, in order to estimate the boundary conditions for the subdomain. Sobel is a filter used for

Benchmark	Single CPU	Single GPU	Multi-Device OpenCL		
			Manual	Framework Static	Framework Dynamic
Sobel	7.661	X	1.384	1.401	1.438
MC	1714.369	50.544	14.181	14.202	14.458
Matrix Mult	342.839	X	18.001	18.153	18.216
Gesummv	1.998	X	0.505	0.502	0.5

Table 1: Execution times (in seconds) for different execution and data management scenarios. Entries marked as X represent configurations which failed due to lack of memory on the device.

edge detection in images. Gesummv is part of the Polybench benchmark suite[5]; it performs two matrix vector multiplications and then adds the results to produce a new vector. Finally, we use a matrix multiplication kernel. Before each experiment we profile the applications to discover the optimal workload partition between the CPU and the GPUs. The hybrid compile- and run-time analysis automatically manages data allocations and transfers.

In Table 1 we summarize the execution times of the OpenCL implementation of all applications on the multicore CPU (using all cores), a single GPU and a multi-GPU setup. In the latter case we report the performance of both a manual data management implementation and that of our framework using either the static or the dynamic strategy. Our results indicate the necessity of utilizing all available resources on a heterogeneous system to achieve optimal results. For applications such as Sobel, Matrix Mult and Gesummv the memory footprint – 7.32 GB, 2.19 GB and 4.29 GB – can easily exceed the capacity of a single GPU. MC, on the other hand, is compute bound. In this case the execution time rather than the memory footprint is the bottleneck; GPU acceleration decreases execution time by 33.91 times compared with the multithreaded CPU implementation, however combining all 4 GPUs results to a 118.58x speedup.

Figure 3 depicts a comparison of the overhead of our methodology and a runtime-only memory management technique, the Unified Memory (UM) feature of NVIDIA CUDA, against manual data management implementations in OpenCL and CUDA respectively. Our framework comes at an average 1.24% execution time overhead compared with the manual data management in OpenCL. This overhead is due to the fact that index calculations on memory access statements have to be modified to account for the automated data layout. An example case of such changes is illustrated in listing 3. In UM, managed memory is shared by both CPU and GPU using a single pointer to data allocated on pinned memory on the Host. The data are managed between devices at the granularity of pages, without programmer intervention. We found that the overhead of using UM in conjunction with multiple GPUs is on average 614% compared with a CUDA implementation with manual data management. The overhead becomes higher in cases of true or false sharing of the same memory page by multiple devices. This happens because CUDA UM uses a single writer scheme to ensure memory coherence. We should point out that CUDA UM operates on memory pages pinned in physical memory, the number of which is typically limited by the operating system. In our experiments, we had to scale-down problem sizes up to 4 times compared with the OpenCL counterparts, in order for CUDA to succeed in allocating the required pinned memory.

Finally, we evaluated the overhead of our framework against manual data management implementations in terms of the

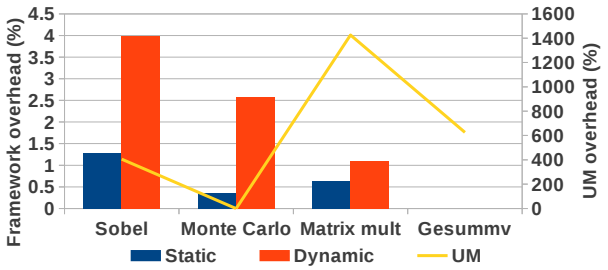

Figure 3: Comparison of our framework overhead and NVidia CUDA UM against multi-device implementations of the benchmarks using manual data management.

volume of data transferred. The static version of our framework allocates and transfers the exact same amount of data as the hand-mapped. versions of the benchmarks. The dynamic version allocates and transfers an additional small buffer (ocl_offsets[]) to each device which is however just few hundreds of bytes. This is orders of magnitude smaller than the data footprint of problems one would typically consider solving on a multi-GPU system.

5. RELATED WORK

We are aware of a few other proposals regarding communication code generation targeting hybrid platforms with distributed memory architectures.

The authors of [3, 4] use the polyhedral model to split a single task to computational tiles. They, too, discover the dependencies between different tiles and produce code that manages the data transfer between host and devices. Their approach focuses on reducing data transfers by only sending data which are essential for each computation. However, they make the assumption that each device has allocated enough space to hold all data used by the original task, even though a single computational tile may use just a subset of these data. Therefore, this approach is not applicable if the full working set of the original kernel does not fit in the memory of each device.

In [17] the authors apply polyhedral analysis to discover the bounds of memory accessed by a source code and transform it in order to minimize the amount of memory allocated. The proposed solution manages to reduce memory allocation, however at the expense of increased use of temporary variables. This excessively increases register pressure on GPUs, thus significantly penalizing performance. Our methodology is a best effort approach towards fine-tuning memory allocation close to the minimum required size, without however exercising pressure to other resources.

Additional works [7, 8, 10] target executing code on a

single GPU. All three frameworks allocate memory for the entire data range on all devices, however they intelligently manage data transfers between the CPU and GPU.

Finally, [9] is a framework designed to partition a single OpenCL kernel over multiple GPUs while maintaining the image of a single compute device for the software developer. The authors split the kernel onto multiple devices at run-time, by introducing a sampling phase before the actual kernel invocation to estimate memory access bounds.

6. CONCLUSIONS

We have introduced and implemented a methodology using polyhedral analysis for automating data management on heterogeneous systems.

From the experimental evaluation it is evident that runtime-only approaches – as is the case with CUDA UM – are not capable of efficiently handling the task of data management on heterogeneous systems. Similarly, compile-time approaches lack the necessary information and flexibility. The proposed hybrid compile- and run-time method offers the necessary flexibility. At the same time it introduces negligible execution-time overhead, as the main part of the analysis is carried out by the compiler.

We plan to further expand our analysis to extract inter-kernel data flow. Beyond GPUs, such a framework would be invaluable for FPGA accelerators. Memory on such devices is organized as small memory islands distributed throughout the silicon and it is critical to both intelligently distribute data close to the logic that consumes them, and to identify data movement between different logic blocks in order to properly size and instantiate memory buffers.

Acknowledgments

This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: THALES grant number MIS 379416.

References

[1] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2004.

[2] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *Compiler Construction*, pages 283–303. Springer, 2010.

[3] U. Bondhugula. Automatic distributed-memory parallelization and code generation using the polyhedral framework. Technical report, Tech. Rep. 2011-3, Department of Computer Science and Automation, Indian Institute of Science, 2011.

[4] R. Dathathri, C. Reddy, T. Ramashekar, and U. Bondhugula. Generating efficient data movement code for heterogeneous architectures with distributed-memory. In *Proceedings of Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 375–386. IEEE, 2013.

[5] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10, May 2012.

[6] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Grösslinger, and L.-N. Pouchet. Polly-Polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, 2011.

[7] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August. Dynamically managed data for CPU-GPU architectures. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO)*, pages 165–174. ACM, 2012.

[8] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August. Automatic CPU-GPU communication management and optimization. *ACM SIGPLAN Notices*, 46(6):142–151, 2011.

[9] J. Kim, H. Kim, J. H. Lee, and J. Lee. Achieving a single compute device image in OpenCL for multiple GPUs. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming (PPoPP)*, pages 277–288. ACM, 2011.

[10] S. Pai, R. Govindarajan, and M. J. Thazhuthaveetil. Fast and efficient automatic memory management for GPUs using compiler-assisted runtime coherence scheme. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques (PACT)*, pages 33–42. ACM, 2012.

[11] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G. A. Silber, and N. Vasilache. Graphite: loop optimizations based on the polyhedral model for gcc. In *proceedings of the 4th gcc developper's summit*, pages 179–198, June 2006.

[12] M. Vavalis and G. Sarailidis. Hybrid-numerical-PDE-solvers: Hybrid Elliptic PDE Solvers. `http://dx.doi.org/10.5281/zenodo.11691`, Sep 2014.

[13] S. Verdoolaege. isl: An integer set library for the polyhedral model. In *Mathematical Software–ICMS 2010*, pages 299–302. Springer, 2010.

[14] S. Verdoolaege and T. Grosser. Polyhedral extraction tool. In *Proceedings of Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*, 2012.

[15] D. K. Wilde. A library for doing polyhedral operations. *Parallel Algorithms and Application*, 15(3-4):137–166, 2000.

[16] H. P. Williams. Fourier-Motzkin elimination extension to integer programming problems. *Journal of combinatorial theory, series A*, 21(1):118–123, 1976.

[17] T. Yuki and S. Rajopadhye. Canonic Multi-Projection: Memory Allocation for Distributed Memory Parallelization. 2011.