Adaptive Modular Mapping to Reduce Shared Memory Bank Conflicts on GPUs

Innocenzo Mungiello, Francesco De Rosa

Abstract This paper presents the experimental evaluation of a new data mapping technique for the GPU shared memory, called Adaptive Modular Mapping (AMM). The evaluated technique aims to remap data across the shared memory physical banks, so as to increase parallel accesses, resulting in appreciable gains in terms of performance. Unless previous techniques described in literature, AMM does not increase shared memory size as a side effect of the conflict-avoidance technique. The paper also presents the experimental set-up used for the validation of the proposed memory mapping methodology.

1 Introduction

Exascale scientific and high-performance computing (HPC) are considered essential in many areas [25] and are today embracing a wealth of different applications [4, 3, 2]. During the last years, HPC architectures have been increasingly moving to heterogeneous systems. In fact, it is now common to integrate in the same HPC system manycore CPUs, accelerators like Graphics Processing Units (GPUs), and even Field-Programmable Gate Arrays (FPGAs) devices, which were traditionally used only for implementing highly specialized circuit-level solutions [6, 7, 11, 5, 10]. In this context, GPUs are gaining importance and are now being used for a variety of domains and applications [21]. FPGAs also provide a solution for building customized computing platforms, although they pose non-trivial challenges, mostly involving complexity, high-level design as well as suitable programming models, which are somewhat mitigated by GPUs [31, 16]. In current

Innocenzo Mungiello
University of Naples Federico II and Centro Regionale ICT (CeRICT), Naples, Italy, e-mail: innocenzo.mungiello@unina.it

Francesco De Rosa
University of Naples Federico II, Naples, Italy

computing technologies, a major component of the power consumption and performance degradation is due to data movement rather than mere processing, which is indeed a widely studied subject in a number of different contexts [26]. This work thus addresses the impact on performance of memory optimization techniques for heterogeneous architectures, particularly GPUs. The emphasis is on the multi-bank organization of the on-chip GPU shared memory, where the whole shared addressing space is partitioned in a cyclic way and is potentially subject to an access conflict problem [14]. In particular, the paper focuses on the experimental evaluation of a new data mapping approach called Adaptive Modular Mapping (AMM) which differs from other works [19, 28, 23, 20] in that it does not involve an increase of the shared memory size, which in some cases can decrease performance as a side effect. The evaluation of the AMM technique highlights a significant gain in terms of performance, as confirmed by an experimental set-up used for performing measures on a physical GPU-based platform.

The paper is structured as follows. Section 2 discusses the background and the motivation of this work. Section 3 recapitulates the approach adopted for data partitioning. Section 4 shows the set-up used for the experimental evaluation. Section 5 concludes the paper with some final remarks.

2 Background

Memory access has traditionally been an important optimization problem for parallel systems [13, 15], and in many classes of systems it may significantly impact perfomance, along with the interconnection subsystem [17, 18, 12]. In particular, memory access *latency* is often an important factor determining the overall system performance. Many solutions have been proposed to face this problem on parallel architectures, e.g. the design of a hierarchical memory system, the introduction of shared scratchpad memories, and the increase of thread-level parallelism to *hide* memory latency.

Most of these features are present on Graphical Processor Units (GPUs) that represent today a major shift in high-performance computing architectures focusing on increasing the execution throughput of parallel applications. In fact, on modern GPU architectures like the NVIDIA Kepler or the newest Maxwell, the memory system is designed in a hierarchical fashion, as shown in Figure 1.

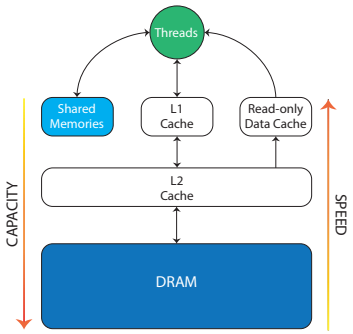

Fig. 1 GPU NVIDIA Kepler Memory Hierarchy.

In this way, parallel processing elements access simultaneously several independent memory banks through complex interconnects. This potentially provides an opportunity for improving the memory bandwidth available to the application. In fact, even if the DRAM latency is too high, it can be hidden by running thousands of threads.

Another way to improve system performance is to use *scratchpad* shared memories. For performance purposes, the on-chip shared memory of a GPU device is normally divided in 32 banks, which can be accessed in a parallel way from all the threads in a *warp*. Data allocated to the shared memory are cyclically distributed over the banks.

Figure 2 shows an example of data mapping on shared memory. In this example the data are 4 byte-word index and, for simplicity, we show only four banks.

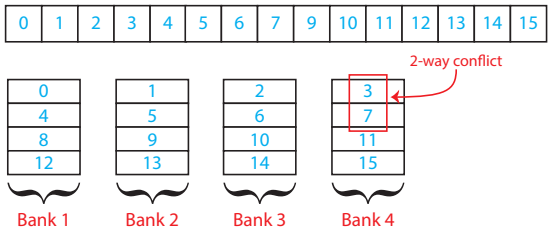

Fig. 2 Shared Memory Data Mapping. The figure shows a possible 2-way bank conflict if a thread would access Bank 4 and fetch value "3" and another thread would access Bank 4 and fetch value "7".

The access mode can affect the performance of a kernel. In fact, allowing all the threads in a warp to fetch data in parallel from this multi-bank memory can lead to great performance improvements, but it requires explicit, bank-aware organization of the data layout. In particular, in the case where two or more threads in the *same* warp try to access *different words* stored in the same bank, the interconnection network is no more able to provide the required data to all the threads in parallel. This situation, called *bank conflict*, is a major problem related to the use of the on-chip shared memory. In particular, if two threads try to access different words stored in the same bank, we have a 2-way bank conflict, as shown in Figure 2. If three threads try to access different words stored in the same bank, a 3-way bank conflict occurs,

and so on. The worst case involves all 32 threads in a warp trying to access different words stored in the same bank, causing a 32-way bank conflict. Whenever a conflict occurs, it is resolved by serializing the accesses in time. As an example, the serialization in a 2-way scenario leads to double latency, decreasing the performance considerably and increasing power consumption, especially if the kernel uses shared memory intensively [27].

2.1 Related Works

Several techniques are presented to solve bank conflicts and reduce memory access latency [8, 29]. The simplest one is *Memory Padding*, presented by NVIDIA in [1, 24, 9]. This technique solves bank conflicts in many cases by simply using an extra empty column of shared memory. While effective and simple, this technique has the disadvantage of wasting shared memory and this can cause problems in certain situations.

A.H. Khan et al. [22] analyze the Matrix Transpose problem and provided a solution very close to AMM techinque for that particular kernel. S. Gao et al. [19] present a framework for automatic bank conflict analysis and optimization. Kim et al. [23] present CuMAPz, a tool to compare the memory performance of a CUDA program and help programmers to optimize them. Sung et al. [28] propose DL, a practical GPU data layout transformation system that solves both problems. Grun et al. [20] present APEX, an approach that extracts, analyzes, and clusters the most active access patterns in the application, and aggressively customizes the memory architecture to match the needs of the application. Z. Wang et al. [30] use a machine-learning approach to transform data layout and improve performance.

3 Data Layout Transformation

Heterogeneous platforms provide some form of customizability that can be exploited to improve performance. GPUs offer a certain degree of freedom enabling the application developer to tailor the bank mapping on the application access pattern. Some applications need access patterns yielding bank conflicts. A naive pattern is shown in Figure 3 where 4 threads access a 4-bank memory in lock-step in such a way that a 25% efficiency is obtained. From the developer point of view a 4x4 matrix accessed with a 4-location strided pattern leads to this behaviour. The simplest way to correct such an access pattern is the Memory Padding technique described in Section 2.1. The resulting bank mapping will be as shown in Figure 3 where, for example, each green cell indicates a concurrent access. In this case the padding technique provides a conflict-free access pattern which improves memory efficiency but also causes a wasted memory problem, as pointed out by the 4 unused red cells, which could under-perform the naive one. In particular, the padding technique can

lead to a performance problem in scenarios exhibiting the scratchpad shared memory as the *limiting factor*, because the number of threads simultaneously eligible to work depends on the memory size. Nvidia CUDA exposes a thread-group based programming model where the whole computation will be completed by several groups of threads simultaneously running. The thread groups are scheduled on the basis of the available resources. Since the padding technique causes more memory to be allocated, a smaller number of threads can be simultaneously eligible to run. These conditions cause decreasing performance although some or all of the bank conflicts are solved. The proposed technique prevents wasting memory by allocating minimal memory, as needed to satisfy the maximum number of threads. The resulting mapping scheme outperforms the padding and the uncorrected ones under the following condition:

cond 1 The scratchpad shared memory is a limiting factor;
cond 2 Memory Padding leads to a smaller number of eligible threads to run simultaneously;

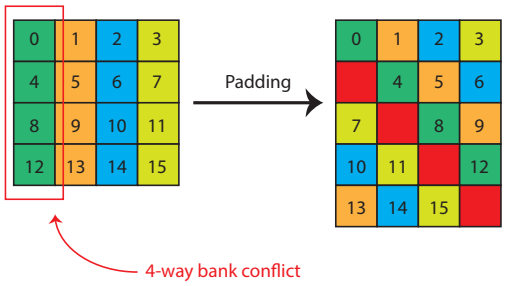

Fig. 3 Padding application. In this case the technique solves all conflicts, but there are empty memory cells highlighted in red.

3.1 Adaptive Modular Mapping

Under the above described conditions, the proposed technique performs a modular remapping of shared memory accesses based on the matrix width, which solves some or all bank conflicts and avoids increased memory size unlike padding techniques. The proposed technique involves the resolution of a linear programming model (Model 4 below) with the aim of extracting new mapping schemes avoiding bank conflicts.

$$\sum_{i=1}^{N\_IT} x_{ijk} = 1 \quad \forall j \in \{1..N\_TH\} \quad \wedge \quad \forall k \in \{1..N\_B\}$$

$$\sum_{j=1}^{N\_TH} x_{ijk} = 1 \quad \forall i \in \{1..N\_IT\} \quad \wedge \quad \forall k \in \{1..N\_B\}$$

$$\sum_{k=1}^{N\_B} x_{ijk} = 1 \quad \forall i \in \{1..N\_IT\} \quad \wedge \quad \forall j \in \{1..N\_TH\}$$

$$\sum_{j=1}^{N\_TH} \sum_{k=1}^{N\_B} x_{ijk} = N\_TH \quad \forall i \in \{1..N\_IT\}$$

$$minimize \left(z = \sum_{j=1}^{N\_TH} \sum_{k=1}^{N\_B} \sum_{k=1}^{N\_IT} x_{ijk} \right)$$

Model 4 N_IT stands for number of iterations, N_TH stands for number of threads and N_B stands for number of banks. x_{ijk} means thread j accesses bank k at i-iteration. This is an NP-Hard problem and the simplest feasible solution is selected.

The results of the Adaptive Modular Mapping technique is shown in Figure 5. Suppose a N-bank memory system and a total of M threads with a M-way conflict.

```
__shared__ int shmem[M][N];

int index = threadIdx.x;

for (int i = 0; i < N; i++)
    shmem[index][i]=some_value;
```

The Adaptive Modular Mapping is highlighted in the following code.

```
__shared__ int shmem[M][N];

int index = threadIdx.x;

for (int i = 0; i < N; i++)
    shmem[index][(index+i)%N]=some_value;
```

where:

- % stands for the modulo reduction operator;
- shmem with the CUDA `__shared__` keyword declares a memory shared among all the threads of a block.

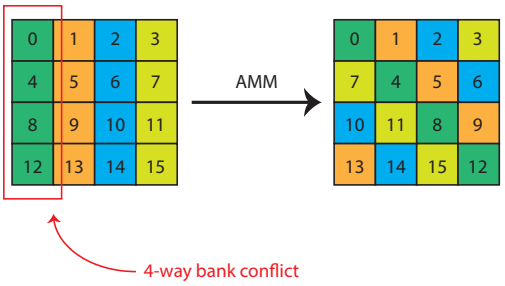

Fig. 5 Adaptive Modular Mapping application. In this case the technique solves all conflicts

As shown above this technique requires a very little effort from the developer point of view, while leading to minor memory requirements and consequently better exploiting parallel computing resources. In particular, the % operator can be realized efficiently by an *AND* operation when dexter is a power-of-two operand.

4 Experimental Evaluation

This section presents the set-up used to carry out the experimental evaluation of the above optimization technique and collect performance data from a physical platform. A suite of kernels has been properly selected to set a scenario where conditions **cond 1** and **cond 2** described in Section 3 are met.

4.1 System Overview

The system includes a host PC running a WMware Virtual Machine with Ubuntu 14.04, with 4 cores, 8 GB of RAM ,and the JetPack 2.1 installed on it. In this environment, NVIDIA Nsight Eclipse Edition is used to write CUDA code and, then, to compile and remotely run it on a Jetson TK1 development board. Memory and Processor frequency of the Jetson TK1 are fixed to 792MHz and 804MHz respectively in order to avoid measurement errors.

4.2 Results

In this section experimental results are discussed. By using the system described in Section 4.1 it was possible to evaluate the impact of the AMM technique on the performance of the code. Five kernels are used to test the AMM technique. They are summarized in Table 1.

Kernel	Suite	Description
Transpose	CUDA SDK	All accesses in global memory are coalesced, there are bank conflicts, and arithmetic operations are low.
Convolution Col	CUDA SDK	All accesses in global memory are coalesced, there are bank conflicts, and arithmetic operations are low.
Lud_Perimeter	RODINIA	All accesses in global memory are coalesced, there are bank conflicts, and arithmetic operations are low.
Foo 1	CUSTOM	Ad-hoc implemented kernel to reproduce coalesced and uncoalesced global memory access pattern on user choice.
Foo 2	CUSTOM	Ad-hoc implemented kernel to reproduce intensive arithmetic operations.

Table 1 Kernels used to test the AMM technique

According to the different characteristics of the kernels, the results can be divided in three classes listed below:

1. All global memory accesses are coalesced, so the memory latency to hide is low. In this scenario the *padding* technique increases the naive kernels performance despite fewer threads are simultaneously running. The AMM technique outperforms both, as shown in Figure 6.

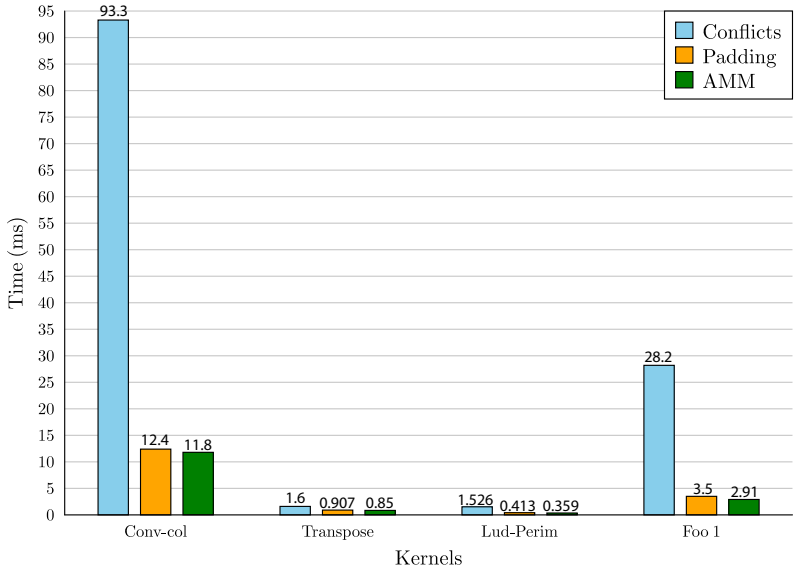

Fig. 6 Case 1 Results. The AMM technique decreases the execution time of all kernels.

2. Global memory accesses are un-coalesced, so the memory latency to hide is high. In this scenario the *padding* technique decreases the naive kernel perfor-

mance because fewer threads are simultaneously running. The AMM technique outperforms both, as shown in Figure 7.

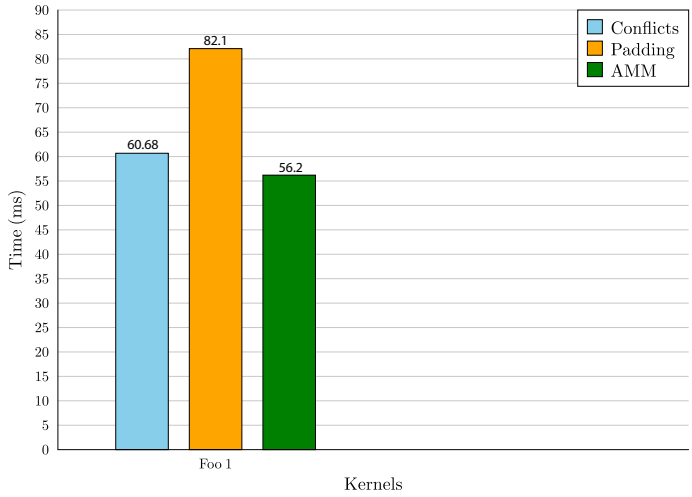

Fig. 7 Case 2 Results. The AMM technique decreases the execution time of the kernel.

3. All global memory accesses are coalesced, so the memory latency to hide is low, but there is a high number of arithmetic instructions. In this scenario, the *padding* technique decreases the naive kernel performance because fewer threads are simultaneously running. The AMM technique in this case equals the naive kernel performance as shown in Figure 8

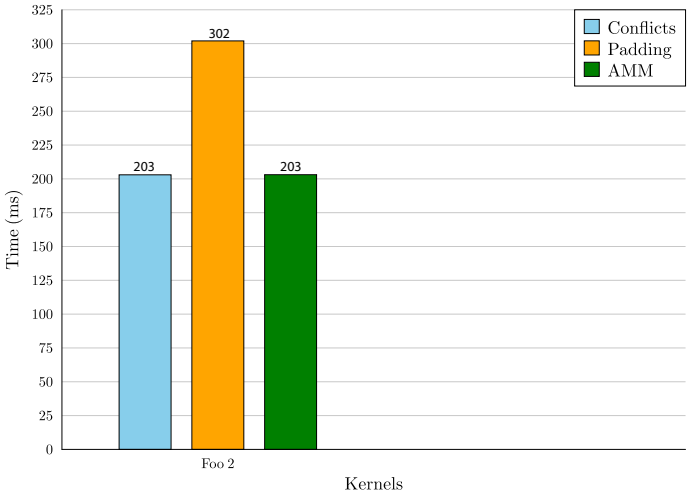

Fig. 8 Case 3 Results. In a compute intensive kernel, AMM technique equals naive performance.

5 Conclusions

GPUs have become extremely important for today HPC as they provide an effective answer for many increasingly demanding applications. This work proposed an optimization technique for GPU on-chip memory. The paper presented both the formal model underpinning the technique and the experimental evaluation on a number of kernels. The results pointed out the significant incidence that such techniques may have on execution time.

Acknowledgments. This work is supported by the European Commission in the framework of the H2020-FETHPC-2014 project n. 671668 - MANGO: exploring Manycore Architectures for Next-GeneratiOn HPC systems.

References

1. CUDA C Programming Guide
2. Amato, F., Fasolino, A., Mazzeo, A., Moscato, V., Picariello, A., Romano, S., Tramontana, P.: Ensuring semantic interoperability for e-health applications. In: Proceedings of the International Conference on Complex, Intelligent and Software Intensive Systems, CISIS 2011, pp. 315–320 (2011)
3. Amato, F., Mazzeo, A., Penta, A., Picariello, A.: Building RDF ontologies from semi-structured legal documents. pp. 997–1002 (2008)
4. Amato, F., Moscato, F.: A model driven approach to data privacy verification in e-health systems. Transactions on Data Privacy **8**(3), 273–296 (2015)

5. Barbareschi, M.: Implementing hardware decision tree prediction: a scalable approach. In: 2016 30th International Conference on Advanced Information Networking and Applications Workshops (WAINA), pp. 87–92. IEEE (2016)
6. Barbareschi, M., Battista, E., Mazzocca, N., Venkatesan, S.: A hardware accelerator for data classification within the sensing infrastructure. In: Information Reuse and Integration (IRI), 2014 IEEE 15th International Conference on, pp. 400–405. IEEE (2014)
7. Barbareschi, M., De Benedictis, A., Mazzeo, A., Vespoli, A.: Providing mobile traffic analysis as-a-service: Design of a service-based infrastructure to offer high-accuracy traffic classifiers based on hardware accelerators. Journal of Digital Information Management **13**(4), 257 (2015)
8. Che, S., Sheaffer, J.W., Skadron, K.: Dymaxion: optimizing memory access patterns for heterogeneous systems. In: Proceedings of 2011 international conference for high performance computing, networking, storage and analysis, p. 13. ACM (2011)
9. Cheng, J., Grossman, M., McKercher, T.: Professional Cuda C Programming. John Wiley & Sons (2014)
10. Cilardo, A.: Efficient bit-parallel $GF(2^m)$ multiplier for a large class of irreducible pentanomials. IEEE Transactions on Computers **58**(7), 1001–1008 (2009)
11. Cilardo, A.: Exploring the potential of threshold logic for cryptography-related operations. IEEE Transactions on Computers **60**(4), 452–462 (2011)
12. Cilardo, A., Fusella, E., Gallo, L., Mazzeo, A.: Exploiting concurrency for the automated synthesis of MPSoC interconnects. ACM Transactions on Embedded Computing Systems **14**(3) (2015)
13. Cilardo, A., Gallo, L.: Improving multibank memory access parallelism with lattice-based partitioning. ACM Transactions on Architecture and Code Optimization **11**(4) (2014)
14. Darte, A., Dion, M., Robert, Y.: A characterization of one-to-one modular mappings. Parallel Processing Letters **6**(01), 145–157 (1996)
15. Darte, A., Schreiber, R., Villard, G.: Lattice-based memory allocation. IEEE Transactions on Computers **54**(10), 1242–1257 (2005)
16. Escobar, F.A., Chang, X., Valderrama, C.: Suitability analysis of fpgas for heterogeneous platforms in hpc. IEEE Transactions on Parallel and Distributed Systems **27**(2), 600–612 (2016)
17. Fusella, E., Cilardo, A.: H^2ONoC: A hybrid optical-electronic NoC based on hybrid topology. IEEE Transactions on Very Large Scale Integration (VLSI) Systems (2016)
18. Fusella, E., Cilardo, A.: Minimizing power loss in optical networks-on-chip through application-specific mapping. Microprocessors and Microsystems (2016)
19. Gao, S., Peterson, G.D.: Optimizing cuda shared memory usage
20. Grun, P., Dutt, N., Nicolau, A.: Apex: access pattern based memory architecture exploration. In: Proceedings of the 14th international symposium on Systems synthesis, pp. 25–32. ACM (2001)
21. Hallmans, D., Åsberg, M., Nolte, T.: Towards using the graphics processing unit (gpu) for embedded systems. In: Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012), pp. 1–4. IEEE (2012)
22. Khan, A., Al-Mouhamed, M., Fatayar, A., Almousa, A., Baqais, A., Assayony, M.: Padding free bank conflict resolution for cuda-based matrix transpose algorithm. In: Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2014 15th IEEE/ACIS International Conference on, pp. 1–6. IEEE (2014)
23. Kim, Y., Shrivastava, A.: Cumapz: a tool to analyze memory access patterns in cuda. In: Proceedings of the 48th Design Automation Conference, pp. 128–133. ACM (2011)
24. Kirk, D.B., Wen-mei, W.H.: Programming massively parallel processors: a hands-on approach. Newnes (2012)
25. Luebke, D.: Cuda: Scalable parallel programming for high-performance scientific computing. In: 2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro, pp. 836–838. IEEE (2008)
26. Lustig, D., Martonosi, M.: Reducing gpu offload latency via fine-grained cpu-gpu synchronization. In: HPCA, vol. 13, pp. 354–365 (2013)

27. Mungiello, I.: Experimental evaluation of memory optimizations on an embedded gpu platform. In: 2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), pp. 169–174. IEEE (2015)
28. Sung, I.J., Liu, G.D., Hwu, W.M.W.: Dl: A data layout transformation system for heterogeneous computing. In: Innovative Parallel Computing (InPar), 2012, pp. 1–11. IEEE (2012)
29. Ueng, S.Z., Lathara, M., Baghsorkhi, S.S., Wen-mei, W.H.: Cuda-lite: Reducing gpu programming complexity. In: International Workshop on Languages and Compilers for Parallel Computing, pp. 1–15. Springer (2008)
30. Wang, Z., Grewe, D., Oboyle, M.F.: Automatic and portable mapping of data parallel programs to opencl for gpu-based heterogeneous systems. ACM Transactions on Architecture and Code Optimization (TACO) **11**(4), 42 (2015)
31. Zhang, Z., Fan, Y., Jiang, W., Han, G., Yang, C., Cong, J.: High-level synthesis: From algorithm to digital circuit (2008)