Memory-Aware Application Mapping on Coarse-Grained Reconfigurable Arrays*

Yongjoo Kim[1], Jongeun Lee[2,**], Aviral Shrivastava[3], Jonghee Yoon[1], and Yunheung Paek[1]

[1] School of EECS, Seoul National University, Seoul, Korea
[2] School of ECE, Ulsan National Institute of Science and Technology, Ulsan, Korea
Tel.: +82-52-217-2116
jlee@unist.ac.kr
[3] Compiler Microarchitecture Lab, Arizona State University, USA

Abstract. Coarse-Grained Reconfigurable Arrays (CGRAs) are a very promising platform, providing both, up to 10-100 MOps/mW of power efficiency and are software programmable. However, this cardinal promise of CGRAs critically hinges on the effectiveness of application mapping onto CGRA platforms. While previous solutions have greatly improved the computation speed, they have largely ignored the impact of the local memory architecture on the achievable power and performance. This paper motivates the need for memory-aware application mapping for CGRAs, and proposes an effective solution for application mapping that considers the effects of various memory architecture parameters including the number of banks, local memory size, and the communication bandwidth between the local memory and the external main memory. Our proposed solution achieves 62% reduction in the energy-delay product, which factors into about 47% and 28% reduction in the energy consumption and runtime, respectively, as compared to memory-unaware mapping for realistic local memory architectures. We also show that our scheme scales across a range of applications, and memory parameters.

1 Introduction

Coarse-Grained Reconfigurable Arrays, or CGRAs, are a very promising platform, providing up to 10 to 100 MOps/mW of power efficiency [1] while still retaining software programmability. CGRAs are essentially an array of processing elements (PEs), like ALUs and multipliers, interconnected with a mesh-like network. PEs can operate on

* This work was supported by the Korea Science and Engineering Foundation(KOSEF) NRL Program grant funded by the Korea government(MEST) (No. 2009-0083190), the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST)/ Korea Science and Engineering Foundation(KOSEF) (R11-2008-007-01001-0), Seoul R&BD Program(10560), the Korea Research Foundation Grant funded by the Korean Government(MOEHRD) (KRF-2007-357-D00225), 2009 Research Fund of the UNIST (Ulsan National Institute of Science and Technology), and grants from Raytheon, Stardust foundation, and NSF (grant no. 0916652).

** Corresponding author.

Y.N. Patt et al. (Eds.): HiPEAC 2010, LNCS 5952, pp. 171–185, 2010.

the result of their neighboring PEs connected through the interconnection network. In addition, each PE has a small number of local registers to store constants and temporary values. Array variables are typically stored in the local memory, also called frame buffer, which is an on-chip SRAM memory with a very high bandwidth toward the PE array. The word-wide datapaths, area-efficient routing, and word-level programmability make them especially suited for multimedia and compute-intensive applications, whereas FPGAs can be more appropriate for complex logic and bit manipulation. Several CGRAs such as MorphoSys [2], RSPA [3], and ADRES [4], have been proposed and implemented, and a comprehensive summary of many of them can be found in [5].

One of the biggest challenges for CGRAs is application mapping, or compilation. Compilation for CGRAs has traditionally focused on two issues: i) placing *operations* (such as arithmetic/logic, multiplication, and load/store) of a loop kernel onto the PE array, and ii) guaranteeing the data flow, or *communication*, between operations using the existing interconnection resources. The third dimension, which has been typically ignored in previous CGRA compilation techniques [6, 7, 8, 9, 10, 11, 12, 13, 14, 15], is where to place *data*, typically array variables, in the local memory. We refer to operation placement and array placement as *computation mapping* and *data mapping*, respectively. Data mapping is not an issue if the local memory has uniform memory access (UMA) architecture—for instance, if the local memory consists of a single large bank. Then any PE can access any local memory address with equal timing, and thus it does not matter where to place an array. However, if the local memory has nonuniform memory access (NUMA) architecture—for instance, if it consists of multiple banks and each bank is connected to only one row of PEs—where to place array variables among multiple banks can affect computation mapping and greatly impact the quality of the overall mapping. Since the local memory of a CGRA is accessed by many PEs each cycle, typically more than a dozen ports exist between the local memory and PEs. Implementing UMA architecture with such a large number of ports is very expensive, either by single-bank multi-port memory [9, 13] or through hardware arbitration [16]. Thus a compiler technique that can effectively manage the complexity of NUMA architecture for CGRA mapping is highly desirable.

Computation mapping and data mapping are two very closely related problems, so solving them sequentially does not give the optimal solution. If array placement is fixed first, and operations are placed later, the computation mapping problem will involve far more constraints than without array placement, that it may be unsolvable or generate poorer solutions than without array placement. Besides, it is not clear how to fix array placement first without doing at least part of computation mapping. On the other hand, if computation mapping is done first, it automatically determines data mapping, which can lead to other problems. First, the same array may be placed in multiple banks (*duplicate array*) if the array is reused in multiple references in the loop (i.e., by multiple loads with different indexes). This can lower the effective size of the local memory and can significantly degrade the performance especially if the local memory is not very large. Second, bank utilization can be unbalanced to a large degree, which can lower the performance if the unbalance in the bank utilization causes extra buffering in the local memory. Third, in recurrent loops, dependent memory operations must be able to

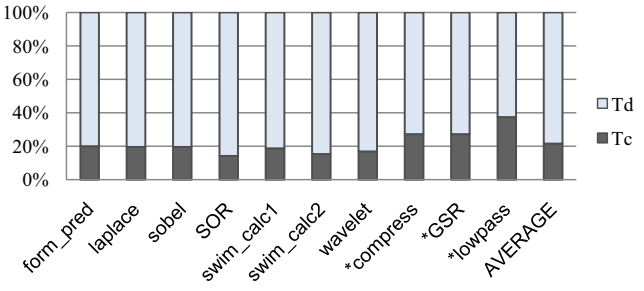

Fig. 1. Many multimedia kernels are memory-bound on RSPA. Td= $t_d/(t_c + t_d)$ and Tc= $t_c/(t_c + t_d)$. Asterisk (*) indicates recurrent loop, where II and t_c can be increased due to data dependence.

access the same array from the same bank. This could be taken care of by constraining dependent memory operations to be mapped to the same rows, for instance, but other constraints (e.g., port contention, memory size restriction) may also be necessary. In general, to guarantee the correctness and optimality of mapping for memory-bound loops on NUMA CGRAs, we must consider not only computation mapping but also data mapping. In this paper we propose a compilation technique, which is aware of the local memory architecture and can find near-optimal mapping considering both array variables and computation operations in memory-bound loops.

After motivating in Section 2 the need for considering the memory architecture and data placement during mapping, we explain our target architecture and application mapping in general in Section 3, and discuss the related work in Section 4. In Section 5 we present our memory-aware heuristic that can be applied to any modular scheduling algorithm such as [9, 13]. Our proposal introduces new costs such as data reuse opportunity cost and bank balancing cost to steer the mapping process to be more aware of the architectural peculiarities. Our experimental results indicate that not only is our proposed heuristic able to achieve near-optimal results as compared to single-bank memory mapping, it can also achieve 62% reduction in the energy-delay product as compared to memory-unaware mapping for multi-bank memory, which factors to 47% and 28% reductions in the energy consumption and runtime, respectively. We also demonstrate that our scheme scales across a range of applications, and memory parameters.

2 Why Consider Data Placement?

If the local memory is large enough, duplicate arrays and unbalanced bank utilization may not be a problem—simply duplicate arrays as many times as needed if each memory bank is unlimited. However, we find that it is not the case in many CGRAs such as MorphoSys and RSPA, and in fact, for larger arrays and loops the entire arrays cannot fit in the local memory and multiple buffer switchings are necessary even during a single loop execution. To minimize slowdown due to buffer switching, those architectures often provide hardware double buffering [2], such that computation (on the PE array) and

data transfer (between the local memory and the system memory) can work on different hardware buffers, overlapping computation and data transfer, and buffer switching can be done instantly, typically in a few cycles. Even with such hardware support, for memory-bound applications it is hard to avoid data transfer becoming the performance bottleneck, since memory bandwidth is not as scalable as increasing the number of PEs on a CGRA.

Figure 1 plots the ratio between t_c and t_d of important loop kernels from MiBench and SPEC benchmark, using the EMS algorithm [13] on the RSPA architecture [3]. The terms t_c and t_d are the computation time and the data transfer time for a tile of a loop, where tile is defined by buffer switching. Then the total execution time of a tile is determined by $\max(t_c, t_d)$. The graph shows that all these loop kernels are indeed memory-bound, i.e., $t_d > t_c$, with the average $t_c/(t_c + t_d)$ being just 22%. Even if we double the memory bandwidth (of between the local memory and the system memory) from 2 bytes per cycle to 4 bytes per cycle, most of the loops still remain memory-bound, with the average $t_c/(t_c + t_d)$ increasing to just 37%. Thus it is important to optimize the data part of the mapping, not just the computation part of it, and even sacrificing computation mapping to some degree in order to gain in data mapping, or in other words balancing t_c and t_d, could lead to enhancement in the overall performance.

3 Background: Architecture and Application Mapping

3.1 CGRA Architecture

CGRA is essentially an array of processing elements (PEs), connected through a mesh-like network (see Figure 2(b)). Each PE can execute an arithmetic or logic operation, multiplication, or load/store. PEs can load or store data from the on-chip local memory, but they can also operate on the output of a neighboring PE connected through the interconnect network. Many resource-constrained CGRA designs have some PEs dedicated for some specific functionality. For example, in each row, typically a few PEs are reserved for multiplication in addition to ALU operations, and a few can perform loading and storing from/to the local memory. The functionality of a PE, i.e., the choice

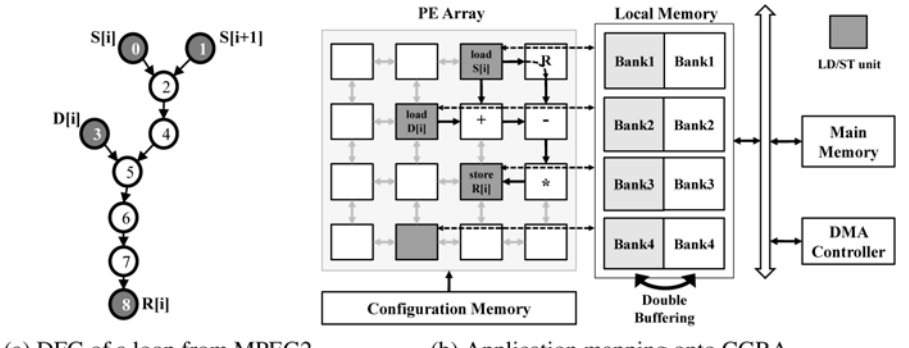

Fig. 2. CGRA architecture and application mapping

of source operands, destination of the result, and the operation it performs is specified in the *configuration*, which is generated as a result of compiling the application on to the CGRA.

A CGRA processor is used as a coprocessor to a main processor. The main processor manages CGRA execution, such as loading of CGRA configurations and initiating CGRA execution, through memory-mapped I/O. Once the CGRA starts execution, the main processor can perform other tasks. Interrupts can be used to notify the completion of CGRA execution. The local memory of a CGRA is managed by the CGRA through DMA. Hardware double buffering allows for full overlap between computation and data transfer on the CGRA, as well as quick switches between buffers; this becomes very critical for large loops, which may require multiple buffer switches during the execution of a single loop.

3.2 Application Mapping

CGRAs are typically used to accelerate the innermost loops of applications, thereby saving runtime and energy. The innermost loop of a perfectly nested loop can be represented as a data flow graph, in which the nodes represent micro-operations (arithmetic and logic operations, multiplication, and load/store), and the edges represent the data dependency between the operations. A loop kernel from MPEG2 is illustrated in Figure 2(a), where dark nodes represent memory operations. While not for this loop, the data dependency can be in general loop-carried. The task of mapping an application onto a CGRA traditionally comprises of mapping the nodes of the data flow graph onto the PE array of the CGRA, and to map the edges onto the connections between the PEs. Since the mesh-like interconnection can be restrictive for application mapping, most CGRAs allow PEs to be used for routing of data (*routing PE*). In the routing mode, the PE does not perform any operation, but just transfer one of the inputs to its output. This flexibility can be exploited by allowing the edges in the data flow graph to be mapped onto paths (composed alternatively of interconnection and a free PE, starting and ending in an interconnection) in the CGRA. Pipelining is explicit in the CGRA, in the sense that the result of computation inside one PE can be used by the neighboring PEs in the next cycle. For effective application mapping, the compiler must software-pipeline the loop before mapping it onto the the PEs for effective mapping. Thus in addition to the problem of expressing the application in terms of the functionality of PEs, a CGRA compiler must explicitly perform resource allocation, pipelining, and routing of data dependencies on the CGRA. It is for these reasons that the problem of application mapping on CGRA is challenging.

4 Related Work

Earlier research on CGRAs was mostly about architecture design [5], but with the recognition that application mapping is the bottleneck, recent work increasingly focuses on application mapping techniques.

4.1 Architecture

Data transfer architectures between local memory and PEs can be classified into implicit load-store and explicit load-store architecture. Implicit load-store CGRA architectures, e.g., MorphoSys [2], do not have explicit load and store instructions. Data has to be pre-arranged in the local memory, organized like a queue, and the topmost element is broadcast to the CGRA every cycle. On the other hand, in explicit load-store CGRAs, e.g., ADRES [4], PEs can explicitly compute the address of the memory location that they intend to access, and read/write to that location. While the implicit load-store architectures are potentially much more power efficient, they are more challenging to program, and also incur penalties relating to the efforts required to arrange the data in a very specific order in the local store.

Local memory can be designed as single-bank or multi-bank. Single-bank memory makes programming much easier; however, it is very difficult to provide all the necessary ports for the PE array with just one bank. One solution is to use multi-port SRAM cells, which are however extremely expensive in terms of area, power, and speed [17]. With multi-bank memory, it is the responsibility of the programmer/compiler to make sure the data that a PE accesses is present in a bank that it has access to. Alternatively, one can use hardware arbitration to make every bank accessible to any PE [16], which makes the local memory design more complicated with higher power, area, and possibly cycle time compared to multi-bank memory without hardware arbitration. Our work provides a software solution rather than a hardware solution to the problem of managing multi-bank memory.

Hardware double buffering, e.g., MorphoSys [2] and RSPA [3], can speed up the data transfer between the system memory and the CGRA local memory, while some architectures, e.g., ADRES [16], opt for a single large buffer. Double buffering becomes more useful if the local memory size is smaller, or the loops and arrays of the applications are larger. We assume explicit load-store, multi-bank, and double-buffered local memory in this work.

4.2 Compilation

Most previous work on application mapping for CGRA [6, 7, 8, 9, 10, 11, 12, 13, 14, 15] does not explicitly consider the local memory architecture or data placement. They assume that all the required data is already present in the local memory, and every load-store PE can access that data whenever they need to. Even with such a simplification, the application mapping problem for CGRA is shown to be very hard [11], having to deal with operation placement on a 2D array considering the communication between them (*spatial mapping*) [12], as well as possibly changing configurations every cycle (*temporal mapping*) [8, 13].

One exception to this is [18], which assumes a hierarchical memory architecture, where the PEs are connected to a L0 local memory, which connects to the external main memory through an L1 local memory. Since both these local memories are scratchpads, and therefore statically scheduled, their main interest is in improving the reuse between the L0 and L1 local memories. An early work [19] on CGRA presents a methodology to evaluate memory architectures for CGRA mapping; however, it lacks a detailed mapping algorithm. [20] also considers memory architecture for mapping, and is therefore

most closely related to our work. However, their mapping assumes multi-bank memory with arbitration logic and single buffering, and therefore is not applicable to our target architecture while we explore the impact of partitioned, or multi-banked, memory architecture and also explore the impact of limited memory bandwidth on the mapping.

5 Our Approach

The real challenge of considering data placement during CGRA mapping is in how to minimize *both* t_c and t_d together within a single framework or algorithm. Simply minimizing t_d is trivial; for instance, fixing the placement of all the arrays beforehand will do, but it may increase t_c excessively. Considering t_d only is what has been typically done in previous approaches, which may fail to minimize the overall t, or the maximum of t_c and t_d. Moreover, data mapping should be emphasized only if the application is memory-bound, which adds to the complexity of our problem. Thus, our CGRA mapping problem considering both computation and data mapping is more complicated than the traditional CGRA mapping problem considering computation only, which is already NP-hard [11]. Hence we propose a heuristic in this paper. We will also demonstrate through our experiments that our heuristic can achieve near-optimal results for many loops.

Our heuristic considers: i) minimizing duplicate arrays (or maximizing data reuse), ii) balancing bank utilization, and iii) balancing t_c and t_d. A unique feature of our heuristic is that it merely defines some cost functions for those memory-related considerations, rather than prescribing a whole new algorithm, so that our heuristic can be easily integrated with other existing memory-unaware mapping algorithms. While our technique is generally applicable to any modular scheduling algorithm considering one operation at a time such as [8, 13], for the sake of the discussion we use the EMS algorithm in this paper as it is one of the best known.

5.1 Balancing Computation and Data Transfer

To balance optimization effort for computation and data parts we first perform *performance bottleneck analysis*. *Performance bottleneck analysis* determines whether it is computation or data transfer that limits the overall performance. We define the data-transfer-to-computation time ratio (DCR) as $DCR = t_d/t_c$. For this we generate an initial, memory-unaware mapping and compute t_c and t_d. t_c is equal to the II multiplied by the tile size, and t_d includes both, the time to bring the data needed for the iteration, and also the time to writeback the data that needs to be committed back to the memory, after each tile. A loop is memory-bound if $DCR > 1$, and roughly represents the optimization opportunity for our memory-aware mapping.

5.2 Maximizing Data Reuse

Temporal reuse of data, or the use of the same data or array elements in different iterations of a loop, is frequently found in many loop kernels. Temporal as well as spatial reuse is automatically exploited by data caches for general purpose processors; however, for CGRAs everything must be explicitly controlled by compilers. Traditional compilation flows for CGRA, which are memory unaware, do not treat specially arrays

with reuse. As a result, two load operations, even if they read from the same array, will typically be mapped to different rows. Note that this is not an issue of functional correctness, but of performance in NUMA CGRAs, since duplicating the arrays in multiple banks solves the correctness problem. An alternative approach is to realize reuse by mapping to the same row all the load operations accessing the same array, which we call *reuse through the local memory*.[2]

Reuse through the local memory has the benefit of lowering the local memory pressure, but at the cost of constraining the computation mapping. Therefore whether and how much reuse to realize should be decided carefully for optimal results. To guide the decision we introduce *data reuse opportunity cost* (DROC). DROC is defined for an operation and a PE, and measures the goodness of a reuse opportunity which will be forfeited if the operation is mapped to the PE. Intuitively, if two load operations have a reuse relation (i.e., they load from the same array), placing them on the same row has merit, which is forfeited if they are placed to PEs on different rows. This reuse opportunity is what DROC tries to quantify.

Data Reuse Analysis: Data reuse analysis finds the amount of potential data reuse between every pair of memory operations. Our data reuse analysis first creates a Data Reuse Graph (DRG) from the data flow graph of a loop. DRG is an undirected graph, where nodes correspond to memory operations and edge weights approximate the amount of reuse between two memory operations. Edges with zero weight are omitted. If two memory operations access different arrays, then the edge weight is zero. Otherwise, the edge weight is estimated to be $TS - rd$, where TS is the tile size, and rd the reuse distance in iterations. Although the eventual tile size for the mapping can only be determined after the loop has been mapped, and it depends on the amount of reuse realized, even an approximate value will do. This is because, all we want by these weights is that the memory operations that share more data should have greater chances to be mapped to the same row. We approximate $TS \approx MS/D_i$, where MS is the size of the local memory, and D_i is the average amount of data that needs to be transferred between the local memory and the external main memory for one iteration of our initial memory-unaware mapping ($D_T = T \cdot D_i$). MS/D_i would have been the tile size for the initial memory-unaware mapping on a single-bank memory architecture. The bigger challenge is to estimate rd, and in general, it can be extremely hard to analyze, especially in the presence of pointers and aliases. Fortunately in many cases reuse takes a very obvious form which can be found even by a very simple analysis. When the access functions (or index expressions) of two references to the same array have an affine form with the same coefficient, rd can be approximated to the difference in the constants divided by the coefficient.[3]

[2] When there is data reuse between two memory operations, the reuse can be realized by routing the data through either the register file (assuming it is rotating) , through routing PEs, or through the memory. Routing data either through the register file or through routing PEs can be wasteful since the involved PEs cannot perform any other operation. In addition, the number of wasted PEs to route data using these schemes is proportional to II, and therefore can be rather large. Therefore, we realize all the data reuse through the local memory.

[3] Only if the coefficient divides the difference; otherwise, there is no reuse between the two references.

Once the DRG is constructed, computing DROC is easy. Given scheduling context information such as what operations have been already scheduled and which operation is about to be scheduled, we first find the set of edges, called *frontier edge set*. For operation u, which is about to be scheduled, the frontier edge set of u includes every edge that connects u and another memory operation, v, in the DRG, and can be found very easily. Then for each edge e in the frontier edge set, we compute its reuse opportunity as $ro_e = w_e \cdot DCR/s_u$, where w_e is the nonzero weight of edge e in the DRG, DCR is the data-transfer-to-computation time ratio of the loop, and s_u is the size of the frontier edge set, or the number of edges in it. (Dividing by the number of edges is necessary to prevent DROC from increasing disproportionately compared to other costs that may exist.) Finally, the reuse opportunity of each edge e induces DROC of the same amount, for all the load-store PEs other than the PE to which v is mapped. DROC induced by all reuse opportunities are added up if the frontier edge set is larger than one. DROC is zero if the frontier edge set is empty.

Example: Consider mapping the DFG shown in Figure 3(b) (dark nodes are memory load operations) onto the 2x2 CGRA shown in Figure 3(a) (dark PEs are load-store PEs). The DRG for the DFG is shown in Figure 3(c). Figure 3(d)–(g) illustrate the mapping results in a tabular format, where the vertical direction represents time in cycles. Suppose that we are about to schedule the edge connecting operations 7 and 8 after having scheduled operations 0 through 6 as shown in Figure 3(d). Operation 7 is a load operation $B[i+1]$, and operation 8 is an arithmetic operation.

The EMS algorithm works as follows: first the routing costs for each open PE slot where the memory operation can be scheduled are updated as in Figure 3(d). Routing cost is calculated by multiplying the unit routing cost (which is assumed to be 10) by the number of routing PEs needed to map the edge. In this example, if we schedule operation 7 in time slot 1 of PE3, at least two routing operations are needed to map operation 8. Thus, routing cost in the time slot 1 of PE3 is 20. Considering these costs, operation 7 will be mapped onto the time slot 3 of PE1, which has the minimum cost. The final solution generated by EMS is shown in Figure 3(e). However, this mapping requires array B to be duplicated in two banks.

DROC helps avoid duplicating reused arrays. In the same example the DROC cost induced by the reuse relation between operations 1 and 7 is 30, assuming that the DCR parameter is 3. This DROC cost is added to all the load-store PEs except for PE3, which forces operation 7 to be scheduled onto the time slot 2 of PE3, as shown in Figure 3(g). Though this new mapping results in the use of an extra PE as a routing PE, it increases the utilization of array B, which may reduce the overall execution time.

5.3 Balancing Bank Utilization

The next important issue in application mapping onto a NUMA CGRA is that, if the scheduler is not careful, it can skew the distribution of the data in the memory banks. For example, the solution can result in mapping all the data to just one bank, and not utilizing the other banks. This can happen, if the application mapping is unaware of the banked memory architecture, but also if we apply our data reuse optimization too aggressively and map all the arrays to the same bank. Such a mapping can reduce the performance, since it decreases the effective local memory size, results in smaller tiling

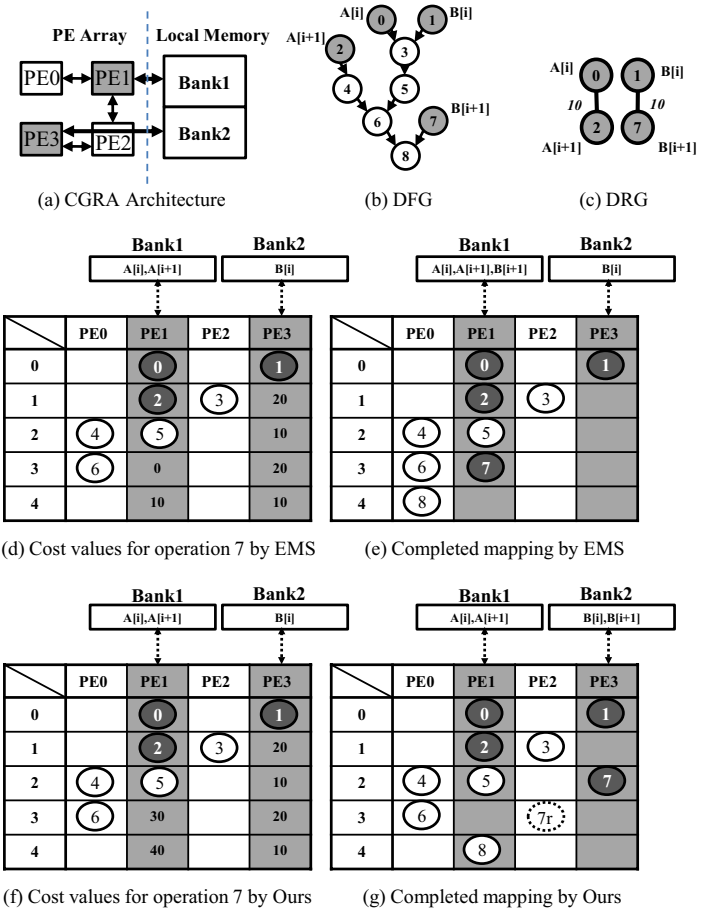

Fig. 3. Data reuse example. Mapping operation 7 to PE3 allows the reuse of array B between operations 1 and 7. Assuming: Base Routing Cost = 10, DCR = 3.

factor for the loop, and may cause very frequent buffer switching for hardware buffering. One desirable shape of the data placement is uniform distribution of the data among the banks. This can be rather easily solved by adding an additional cost to the PEs to which load/store operations have been mapped, called *bank balancing cost*. We define the bank balancing cost for a PE p, as $BBC(p) = b \cdot m(p)$, where b is a design parameter called the base balancing cost, and $m(p)$ is the number of memory operations already mapped onto PE p.

Figure 4 illustrates our compilation flow. The two analyses, performance bottleneck analysis and data reuse analysis, are performed before time-consuming modulo scheduling. Memory-aware modulo scheduling refers to the EMS algorithm extended by adding DROC and BBC to the existing cost function, which does not significantly increase the complexity of the mapping algorithm. The partial shutdown exploration is explained in Section 6.3.

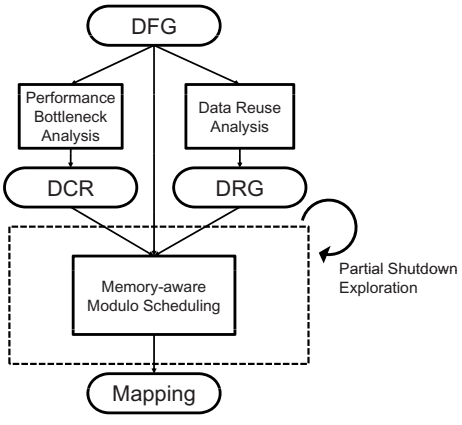

Fig. 4. Application mapping flow. Note: DFG (Data Flow Graph), DCR (Data-transfer-to-Computation time ratio), and DRG (Data Reuse Graph).

6 Experiments

6.1 Setup

We demonstrate the effectiveness of our memory-aware compilation heuristic on a set of important kernels from the MiBench benchmark suite [21], multimedia benchmarks, and SPEC 2000. Our target architecture is a 4x4 architecture, as illustrated in Figure 2(b), with load-store units alternating in the two middle columns. The 4x4 configuration is the basic unit in many CGRA architectures including ADRES (4x4 tiles), MorphoSys (4x4 quadrants), and also frequently used to evaluate various mapping algorithms (e.g., [9, 12, 13]). For the PE array, we assume that a PE is connected to its four neighbors and four diagonal ones. The local memory architecture has 4 banks, each connected to each row (i.e., to the load-store unit of the corresponding row). The detail of the local memory architecture is modeled after the RSPA architecture [3]. The local memory is double buffered in hardware and the buffers can be switched in one cycle. The size of each buffer is 768 bytes, or 384 16-bit words, and is connected to the system memory through a high-performance 16-bit pipelined bus. The system memory operates at half the frequency of the processor, thus the memory bandwidth is 16 bits per 2 cycles.

In the literature mapping algorithms are often compared in terms of II, which is valid, since CGRA processors are under a complete compile-time control; it is like a VLIW processor without pipeline stall. However II captures the quality of the computation mapping only, and cannot capture the possible delay due to the memory bottleneck. We therefore use the CGRA runtime, which is computed by adding up tile execution times, where tile execution time is the maximum of computation II multiplied by the tile size and the memory access time for the tile. We assume that an array shared by two references such as $A[i]$ and $A[i+5]$ requires $T+5$ elements per tile instead of just T, where T is the tile size. If an array is duplicated in multiple banks with different offsets,

we assume that the array is loaded twice from the system memory, which is the most straightforward way to load them; otherwise, the DMA would have to be smart enough to copy a part of the array from one bank to another, and manage the remaining part.

For the energy model of the CGRA, we consider both the dynamic power and the leakage power of PEs and memory banks. The dynamic power model of a PE is derived from RSPA, considering three operating states: ALU (including load/store), multiplication, and routing. The dynamic power model of a memory bank is given by CACTI 5.1 [17]. The leakage power is assumed to be 20% of the dynamic power of an ALU operation for a PE, and of a read operation for a memory bank.

6.2 Efficiency of Our Memory-Aware Mapping

Though our memory-aware mapping may reduce the total execution time of a loop (i.e., $\max(t_c, t_d)$), the computation time (t_c) will be minimized in the case of traditional memory-unaware mappings such as EMS. The minimum computation time could be realized if single-bank memory were used, although it seems likely to have other negative effects such as increased cycle time, power, and area, and may cancel out the benefit. Thus we compare three cases: *Ideal* (single-bank + EMS), *EMS* (multi-bank + EMS), and *MA* (multi-bank + our memory-aware extension of EMS). For a realistic multi-bank local memory, the *Ideal* single-bank performance only serves as the upper limit that a realistic multi-bank mapping could achieve. We compare the three cases in terms of cycle count. In the case of *Ideal*, the possible cycle time increase is not taken into account, nor is the memory bandwidth restriction (hence the name). In the case of *EMS*, the array placement is determined in a straightforward manner after computation mapping is done.

Figure 5 compares the runtime of the three cases (in cycle count), normalized to that of *EMS*. Comparing *Ideal* and *EMS* indicates that for memory-bound loops, the cost of not considering array placement early in the compilation flow is quite high. By sequentially mapping computations and arrays, the runtime can increase by more than 40% on average compared to the Ideal case for memory-bound loops. On the other

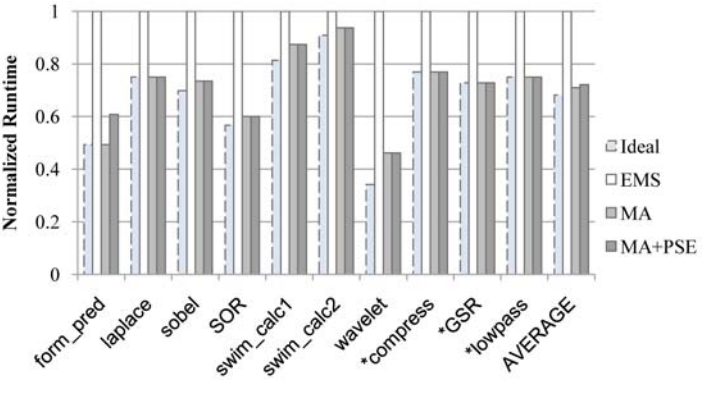

Fig. 5. Runtime comparison

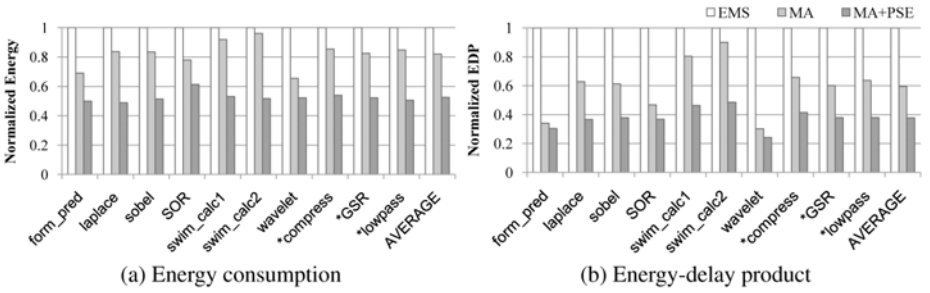

Fig. 6. Energy efficiency comparison

hand, if data mapping is considered proactively along with computation mapping as in our heuristic, the runtime increase can be very effectively suppressed. Compared to the *EMS*, our heuristic can reduce the runtime by as much as 30% on average for memory-bound loops. This strongly motivates the use of less expensive multi-bank memories for CGRAs rather than the more expensive and more power-dissipating single-bank memories.

Reduced runtime by our heuristic also translates into reduced energy consumption on the CGRA. Figure 6(a) compares the energy consumption by the base EMS vs. our heuristic. While our heuristic can sometimes generate less efficient computation mappings compared to the base EMS, for instance, by using more routing PEs, our heuristic can effectively reduce the leakage energy by reducing the runtime, which leads to significant energy reduction by our heuristic. Accordingly, the EDP, or the energy-delay product, is also reduced significantly by our heuristic, as indicated by Figure 6(b).

6.3 Partial Shutdown Exploration

For a memory-bound loop, the performance is often limited by the memory bandwidth rather than by computation, which will be increasingly the case as the number of PEs increases. For such a case we can dramatically reduce the energy consumption of CGRA by shutting down some of the rows of PEs and the memory banks, effectively balancing computation and memory access. While this kind of optimization could be applied with any mapping algorithm, it becomes more interesting with our memory-aware mapping heuristic, as both our heuristic and partial shutdown try to exploit the same opportunity existing in memory-bound loops; one by reducing the memory access load, the other by reducing the computation rate.

We explore all the partial shutdown combinations on the PE rows and the memory banks, to find the best configuration that gives the minimum EDP. The design space is not large, with only 16 configurations to explore as there are 4 rows and 4 banks. The results are summarized in Figures 5 and 6 (the last bars). The results suggest that the partial shutdown optimization can considerably reduce the energy consumption and the EDP, by more than 35% on average, even after our memory-aware heuristic is applied. Compared to previous memory-unaware technique without partial shutdown optimization, our technique can achieve 62% reduction in the energy-delay product, which factors into

Table 1. Best configurations by partial shutdown exploration (r=#rows, m=#banks)

Mem BW	form_pred	laplace	sobel	SOR	swim_calc1	swim_calc2	wavelet	*compress	*GSR	*lowpass
1w/2cyc	2r1m	2r1m	3r2m	2r1m	3r1m	3r1m	2r1m	1r1m	1r1m	2r2m
1w/1cyc	2r2m	3r2m	4r4m	2r2m	4r2m	3r2m	3r2m	2r2m	2r2m	2r2m

about 47% reduction in the energy consumption and 28% reduction in the runtime. For this exploration we also vary the memory bandwidth. The runtime and energy reduction shows a similar trend (not shown), but interestingly the best configurations (shown in Table 1) tend to be larger as the memory bandwidth is increased.

Our partial shutdown exploration gives further justification for the multi-bank memory architecture, as it is more amenable to partial shutdown than the single-bank memory architecture. And it also reinforces the importance of developing memory-aware mapping techniques for multi-bank or NUMA memory architectures, such as ours.

7 Conclusion

The promise of Coarse-Grained Reconfigurable Arrays (CGRAs) providing very high power efficiency while being software programmable, critically hinges on the effectiveness of application mapping. While previous solutions have focused on improving the computation speed of the PE array, we motivate the need for considering the local memory architecture and data placement to achieve higher performance and energy efficiency for memory-bound loops on CGRAs. We propose an effective heuristic that can be easily integrated with existing modular scheduling based algorithms, and which considers various memory architecture parameters including the number of banks, local memory size, and the communication bandwidth between the local memory and the system memory. Our experimental results on memory-bound loops from MiBench, multimedia, and SPEC benchmarks demonstrate that not only is our proposed heuristic able to achieve near-optimal results as compared to single-bank memory mapping, it can also achieve 62% reduction in the energy-delay product as compared to memory-unaware mapping for multi-bank memory, which factors into 47% and 28% reductions in the energy consumption and runtime, respectively. Further, our extensive experiments show that our scheme scales across a range of applications, and memory parameters.

References

1. Bormans, J.: Reconfigurable array processor satisfies multi-core platforms. Chip Design Magazine (2006)
2. Singh, H., Lee, M.-H., Lu, G., Bagherzadeh, N., Kurdahi, F., Filho, E.: Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications. IEEE Trans. Comput. 49(5), 465–481 (2000)
3. Kim, Y., Kiemb, M., Park, C., Jung, J., Choi, K.: Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization. In: DATE 2005, Washington, DC, USA, pp. 12–17. IEEE Computer Society, Los Alamitos (2005)

4. Mei, B., Vernalde, S., Verkest, D., Lauwereins, R.: Design methodology for a tightly coupled vliw/reconfigurable matrix architecture: A case study. In: DATE 2004, p. 21224 (2004)
5. Hartenstein, R.: A decade of reconfigurable computing: a visionary retrospective. In: DATE 2001, Piscataway, NJ, USA, pp. 642–649. IEEE Press, Los Alamitos (2001)
6. Lee, J., Choi, K., Dutt, N.: Compilation approach for coarse-grained reconfigurable architectures. IEEE D&T 20, 26–33 (2003)
7. Lee, J., Choi, K., Dutt, N.: An algorithm for mapping loops onto coarse-grained reconfigurable architectures. ACM SIGPLAN Notices 38(7), 183–188 (2003)
8. Mei, B., Vernalde, S., Verkest, D., De Man, H., Lauwereins, R.: Dresc: a retargetable compiler for coarse-grained reconfigurable architectures, December 2002, pp. 166–173 (2002)
9. Park, H., Fan, K., Kudlur, M., Mahlke, S.: Modulo graph embedding: mapping applications onto coarse-grained reconfigurable architectures. In: CASES 2006, pp. 136–146. ACM, New York (2006)
10. Hatanaka, A., Bagherzadeh, N.: A modulo scheduling algorithm for a coarse-grain reconfigurable array template. In: IPDPS 2007, March 2007, pp. 1–8 (2007)
11. Ahn, M., Yoon, J., Paek, Y., Kim, Y., Kiemb, M., Choi, K.: A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures. In: DATE 2006, 3001 Leuven, Belgium, pp. 363–368. European Design and Automation Association (2006)
12. Yoon, J., Shrivastava, A., Park, S., Ahn, M., Jeyapaul, R., Paek, Y.: Spkm: a novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures. In: ASP-DAC 2008, pp. 776–782 (2008)
13. Park, H., Fan, K., Mahlke, S., Oh, T., Kim, H., Kim, H.: Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In: PACT 2008, pp. 166–176. ACM, New York (2008)
14. Venkataramani, G., Najjar, W., Kurdahi, F., Bagherzadeh, N., Bohm, W.: A compiler framework for mapping applications to a coarse-grained reconfigurable computer architecture. In: CASES 2001, pp. 116–125. ACM Press, New York (2001)
15. Lee, W., Barua, R., Frank, M., Srikrishna, D., Babb, J., Sarkar, V., Amarasinghe, S.: Space-time scheduling of instruction-level parallelism on a raw machine. In: ASPLOS-VIII, pp. 46–57 (1998)
16. Bougard, B., De Sutter, B., Verkest, D., Van der Perre, L., Lauwereins, R.: A coarse-grained array accelerator for software-defined radio baseband processing. IEEE Micro 28(4), 41–50 (2008)
17. Thoziyoor, S., Muralimanohar, N., Ahn, J., Jouppi, N.: Cacti 5.1. Technical report (2008)
18. Dimitroulakos, G., Galanis, M., Goutis, C.: Alleviating the data memory bandwidth bottleneck in coarse-grained reconfigurable arrays. In: ASAP 2005, Washington, DC, USA, pp. 161–168. IEEE Computer Society, Los Alamitos (2005)
19. Lee, J., Choi, K., Dutt, N.: Evaluating memory architectures for media applications on coarse-grained reconfigurable architectures. In: Proc. ASAP, pp. 172–182. IEEE, Los Alamitos (2003)
20. Dimitroulakos, G., Georgiopoulos, S., Galanis, M., Goutis, C.: Resource aware mapping on coarse grained reconfigurable arrays. Microprocess. Microsyst. 33(2), 91–105 (2009)
21. Guthaus, M., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: Mibench: A free, commercially representative embedded benchmark suite. In: IWWC, pp. 3–14 (2001)