Compiling HPC Kernels for the REDEFINE CGRA

Kavitha T Madhu*, Saptarsi Das*, Nalesh S.*, S. K. Nandy* and Ranjani Narayan†
*CAD Laboratory, Indian Institute of Science, Bangalore
Email: {kavitha, sdas, nalesh}@cadl.iisc.ernet.in, nandy@serc.iisc.in
†Morphing Machines Pvt. Ltd., Bangalore
Email: ranjani.narayan@morphingmachines.com

***Abstract*—In this paper, we present a compilation flow for HPC kernels on the REDEFINE coarse-grain reconfigurable architecture (CGRA). REDEFINE is a scalable macro-dataflow machine in which the compute elements (CEs) communicate through messages. REDEFINE offers the ability to exploit high degree of coarse-grain and pipeline parallelism. The CEs in REDEFINE are enhanced with reconfigurable macro data-paths called HyperCells that enable exploitation of fine-grain and pipeline parallelism at the level of basic instructions in static dataflow order. Application kernels that exhibit regularity in computations and memory accesses such as affine loop nests benefit from the architecture of HyperCell [1], [2]. The proposed compilation flow aims at exposing high degree of parallelism in loop nests in HPC application kernels using polyhedral analysis and generates meta-data to effectively utilize the computational resources in HyperCells. Memory is explicitly managed through compiler's assistance. We address the compilation challenges such as partitioning with load balancing, mapping and scheduling computations and management of operand data while targeting multiple HyperCells in the REDEFINE architecture. The proposed solution scales well meeting the performance objectives of HPC computing.**

I. Introduction

Coarse-grain reconfigurable architectures offer the ability to exploit massive parallelism close to a many-core/multicore system with a very low power budget. There exist CGRAs in which ASIC-like hardware data-paths can be created on demand, at runtime. Examples of such CGRAs include Molen Polymorphic Processor [3], Convey Hybrid-Core Computer [4], DRRA [5], REDEFINE [6]. Unlike GPUs, they are not strictly restricted to SIMD paradigm of execution. In the recent past a plethora of such CGRAs have emerged as potential platforms for accelerating kernels from HPC applications. Such architectures aid in exploiting different types of parallelism resident within application kernels, thus paving way for a compiler to expose four main types of (latent) parallelism, viz., ILP, DLP and TLP and pipeline parallelism. REDEFINE is one such CGRA designed to ease exploitation of all kinds of parallelism. A Network-on-Chip (NoC) provides a packet switched network that interconnects compute and storage resources. This inherent flexibility in the NoC aids in pooling together the compute and storage resources suitably for an application kernel. A plurality of Compute Elements (CEs) in REDEFINE aid in exploiting DLP, TLP and pipeline parallelism. The ability of REDEFINE to employ CEs with reconfigurable macro data-paths called HyperCells [1], [2] enables exploitation of fine grain parallelism resident within a coarse grain entity. Also, REDEFINE inherently allows low latency configuration and synchronization. These features coupled with the ability to to partially reconfigure a subset of HyperCells makes it suitable for executing HPC kernels.

While architectures (such as REDEFINE) are available as experimental platforms to exploit different types and granularities of parallelism, a big challenge needs to be addressed: A tool to expose parallelism resident in application kernels. In this paper we focus on compilation strategies for REDEFINE based on polyhedral loop optimizations since most HPC applications spend a large fraction of execution time on loops. Regular loop nests with affine array accesses can be analysed with compilation frameworks based on the polyhedral model [7]. The approach employed in [7] performs partitioning for general purpose control-flow cores with data locality ensured at register or cache levels. Similar approaches have been employed to target specific hardware units such as FPGA [8].

This paper presents a comprehensive compilation solution covering multiple aspects such as partitioning, mapping, data layout for the architecture to optimize the performance of loops. The compilation strategy exploits the communication hierarchy provided by the architecture to generate performance optimal executables. It employs a bottom-up approach where computations are identified for CEs first. The computations are then aggregated into coarse-grain schedulable entities for REDEFINE keeping in mind that the hardware resources are shared among different forms of communication. Maximizing resource utilization is the objective in each step of the compilation flow. Primary goals of the proposed compilation process presented in this paper are partitioning and mapping kernels for maximally utilizing resources (with balanced loads across computing resources), creating schedules for the coarse-grain computations such that there is maximal overlap in inputs and outputs and hence exploit locality, creating meta-data for orchestrating data delivery from an external memory infrastructure onto REDEFINE memory such that memory transaction overheads are minimized and reducing the reconfiguration overheads by employing fine grain reconfiguration.

II. Description of the REDEFINE Architecture

REDEFINE [6] is a CGRA composed of multiple CEs (refer figure 1). The CEs communicate with each other over an NoC. REDEFINE architecture framework facilitates exploitation of coarse-grain as well as pipeline parallelism through execution of kernels on multiple CEs in dataflow order. An application kernel is expressed in terms of convex basic schedulable entities for execution in REDEFINE called HyperOps. A HyperOp further comprises multiple co-operating pHyperOps that communicate with each other. Each pHyperOp is a MIMO operation mapped onto a CE. Multiple pHyperOps forming a HyperOp are launched simultaneously onto REDEFINE

fabric. Memory is structured as an explicitly managed sets of multiple banks as shown in figure 1. It offers a global store for REDEFINE's compute fabric and the address space is logically partitioned to be used for prefetching. Memory barriers and contention resolution mechanism for CEs across HyperOps are realized by the orchestrator. In order to exploit higher degree of ILP resident within each pHyperOp, we enrich CEs with HyperCells[1], [2]. HyperCell is a micro-dataflow machine with a controller designed for executing Dataflow Graphs (DFGs) in a pipelined manner. HyperCell provides close to ASIC realization of a DFG and enables exploitation of high degree of ILP at the level of the DFG it executes and pipeline parallelism across instances of the DFG. HyperCell comprises a reconfigurable data-path, a

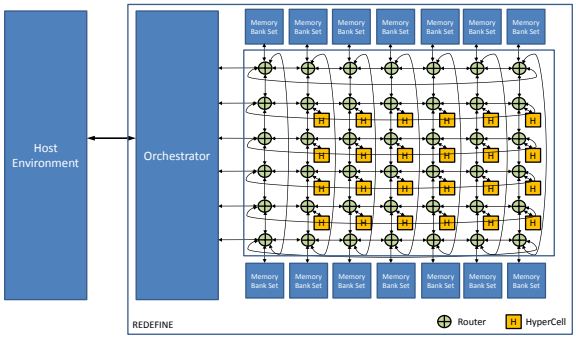


Fig. 1: REDEFINE CGRA with HyperCells as CEs

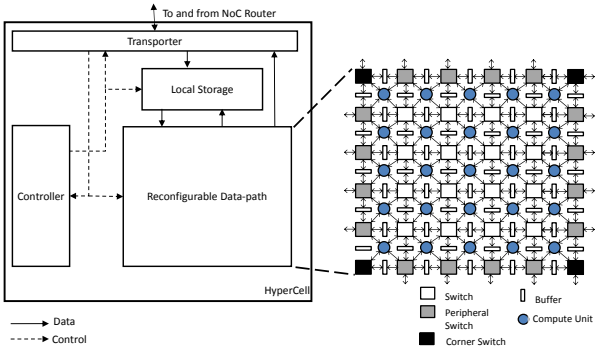
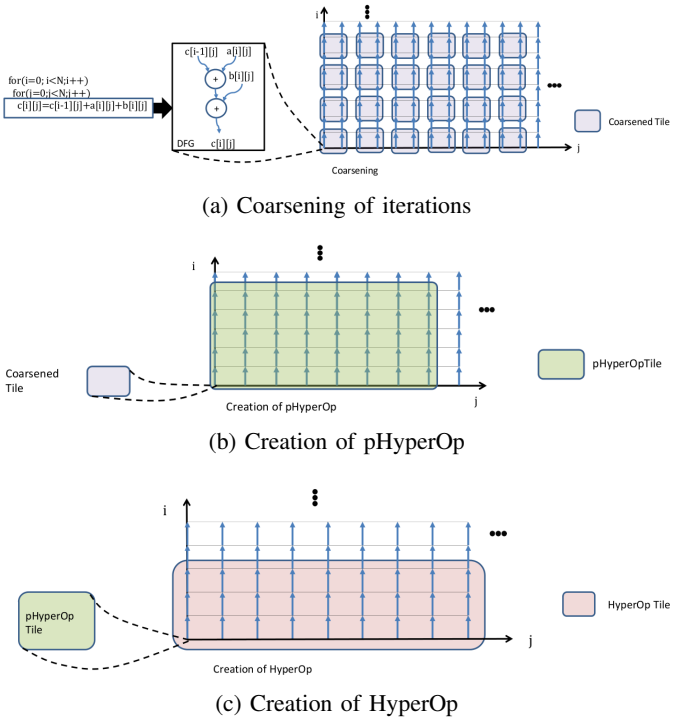


Fig. 2: Micro-architecture of HyperCell

local storage and a controller. The reconfigurable data-path comprises a number of ALUs/FUs connected over a circuit-switched programmable interconnect (refer to figure 2, [1], [2]). The local storage acts as a dedicated operand/output data storage for the reconfigurable data-path and offers high bandwidth for data transfer. HyperCell controller provides the necessary micro-architectural support for exploiting temporal parallelism. Data transfer between memory and local storage and local storage and HyperCell fabric is based on a set of control words. Controller executes instances of the kernels in a modulo schedule based on control sequences that typically have a prologue, a steady state (executed repeatedly depending on the bounds) and an epilogue. The NoC [9] facilitates inexpensive communication between HyperCells with a hand-shake mechanism for flow-control between communicating



(a) Coarsening of iterations

(b) Creation of pHyperOp

(c) Creation of HyperOp

Fig. 3: Coarsening of iterations and creation of pHyperOp and HyperOp in a simple 2-d loop nest

HyperCells. In addition, NoC is used by HyperCells to access memory. Although NoC allows communication between any HyperCell and any set of memory banks, the compiler tries to place data and computations such that memory transactions are local to columns of HyperCells. Compiler also determines the interaction among HyperOps of a kernel. Orchestrator is configured with this information every time a new kernel is to be executed on REDEFINE. Orchestrator also synchronizes execution of HyperOps. Further, when a HyperOp is ready to be launched for execution, orchestrator configures the Hyper-Cells executing a HyperOp. It also manages data movement between the host environment and REDEFINE's memory.

III. Compilation Flow

In this section we discuss our proposed compilation flow. The section is structured as follows. We discuss various factors that affect performance of a kernel in section III-A. In section III-B we present the steps in the proposed compilation flow with the objectives of maximizing utilization of hardware resources, thereby maximizing efficiency of execution.

A. Factors Affecting Performance

Consider a simple two dimensional loop-nest shown in figure 3. The loop when expressed as a dependence graph in the iteration space, is a lattice of points. Each point in the lattice embeds the DFG corresponding to the computation `c[i][j] = c[i-1][j] + a[i][j] + b[i][j]`. The DFG corresponding to each iteration is *mapped* onto a HyperCell's reconfigurable data-path. It may have to be coarsened into tiles (refer figure 3a) to maximize utilization or partitioned

to meet the structural constraints. Each HyperCell executes many instances of the coarsened/partitioned loop body in a pipeline. In the context of multi-dimensional loop nests, a tile (referred to as a pHyperOp) in the coarsened iteration space is assigned one HyperCell for execution (refer figure 3b). Multiple HyperCells are capable of concurrently executing a number of pHyperOps, using the NoC to communicate data between pHyperOps. Compiler combines such pHyperOps into bundles aka HyperOps (refer figure 3c) that are launched atomically on REDEFINE's compute fabric.

Figure 4 depicts the execution of a loop nest divided into multiple HyperOps on REDEFINE. The total execution time of the loop on REDEFINE includes the time spent in actual computation, time spent in configuring the HyperCells and time spent in synchronization. During configuration or synchronization operations, compute elements stay idle resulting in low utilization. Low latency synchronization and reconfiguration at the granularity of HyperOps and inexpensive point-to-point synchronization between HyperCells are inherent in REDEFINE architecture. The architecture offers partial reconfiguration feature reducing the runtime overheads further. Contribution of configuration and synchronization time to total execution time is estimated to be much smaller than actual computation time. Hence, achieving better utilization while performing computations is of utmost importance. Three major factors impact utilization during computations and are listed here in decreasing order of their impact on overall computation time.

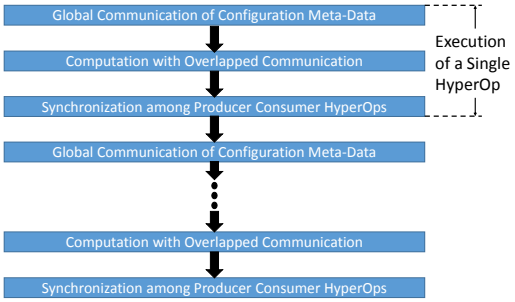


Fig. 4: Execution flow of HyperOps in the REDEFINE architecture

1) Initiation interval of Computations in each HyperCell: Initiation interval corresponds to the rate at which computations can be launched within each HyperCell. Ideally, a new DFG instance should be initiated every cycle in each HyperCell. Computation time (T_{comp}) increases linearly with increase in initiation interval. Architectural artifacts affect the initiation interval of computations as explained subsequently.

Limited Local Storage and network bandwidth induce an initiation interval when the inputs and outputs delivered between HyperCell's data-path and local storage or outputs to another HyperCell require higher bandwidth. The network resources may further be shared among multiple paths of communication (Let p be the number of paths share the set of network resources).

$$\Delta_{in_{loc}} = \frac{\#\ local\ storage\ inputs}{local\ storage\ bandwidth}$$

$$\Delta_{out_{loc}} = \frac{\#\ local\ storage\ outputs}{local\ storage\ bandwidth}$$

$$\Delta_{out,nw} = \frac{\#\ outputs\ written\ to\ other\ HyperCell(s)}{\frac{network\ bandwidth}{p}}$$

$$\Delta_{in,out_{loc}} = max(\Delta_{in_{loc}}, \Delta_{out_{loc}}, \Delta_{out,nw})$$

Memory Bandwidth induces an initiation interval due to inputs and outputs being fetched from memory or written to memory for each instance of DFG.

$$\Delta_{in,out_{mem}} = \frac{\#\ memory\ inputs + \#\ memory\ outputs}{min(memory\ bandwidth, networkbandwidth/p)}$$

Control Storage and Local Storage size of HyperCells limit the longest reuse distance(i.e., the longest interval for which input or output data of a computation can be kept alive for use in a successive instance of computation $max_{control}$) supported by a single HyperCell, increasing the critical path of computation τ and the initiation interval $\Delta_{cp} = \frac{\tau}{max_{control}}$.

Overall initiation interval is given by the following expression and is minimized across various compilation steps.

$$\Delta = max(\Delta_{in,out_{loc}}, \Delta_{in,out_{mem}}, \Delta_{cp})$$

2) Under-utilization of FUs in HyperCell: HyperCell's FUs may be under-utilized due to factors such as imperfection in loops, structural constraints of the data-path such as number of FUs, IO ports and mappability. This has a sub-linear effect on execution time.

3) Under-utilization of HyperCells in REDEFINE fabric: Static mapping of HyperOps onto REDEFINE fabric proves useful in generating configuration statically but may result in under-utilization due to pipeline start and drain and at the boundaries of iteration space. Pipeline start and drain may prevent concurrent start of execution of pHyperOps belonging to a HyperOp.

B. Compilation Steps

The compilation steps are designed for maximizing utilization primarily by minimizing initiation interval due to various hardware artifacts. While other factors may affect utilization, initiation interval impacts performance the most and hence, has higher priority over other optimization criteria among different steps.

1) Identifying DFGs for HyperCell: HyperCell's reconfigurable data-path offers the flexibility to house the computations in a dataflow graph (DFG) that honors its structural constraints and has the fastest communication infrastructure. If the DFG corresponding to a point in the iteration space does not meet the structural constraints of a HyperCell's data-path, it is partitioned into multiple subgraphs that are mapped onto different HyperCells instead of performing loop fission to place the producer and consumer computations close. Each HyperCell has its own configuration storage allowing partitioned graphs to reside in separate HyperCells and communicate seamlessly. On the other hand, multiple points in the iteration space need to be grouped together if the DFG under-utilizes compute resources in the data-path. Tiling is an important loop transformation applied to achieve this. The objective of this step is to identify the correct tile size so as to maximize resource utilization of HyperCells and increase temporal locality of data within a tile. Compute resources of HyperCell maybe under-utilized when executing tiles at the iteration space boundaries. Tiles whose boundaries align with the iteration space boundaries are best suited for achieving high utilization with affordable trade-off in data locality. Imperfect loop nests add to the utilization

problem since they are not executed throughout the iteration space. If loop fission cannot be performed, they need to be handled explicitly when coarsening. The tile size then needs to be computed such that it can accommodate statements causing imperfections. Let the n dimensional iteration space of the candidate loop be represented by the set of iteration vectors $I = \{i_1, i_2, i_3 \cdots i_n\}$. Let $H = \{h_1, h_2, h_3 \cdots h_n\}$ be the tiling hyperplanes such that all h_js are linearly independent and are computed as mentioned in [7] with higher priority associated with WAW dependence vectors than RAW/RAR vectors. WAW vectors are associated with higher priority since excluding a WAW vector results in more memory operations than the other dependence types. Let the size of a tile along each hyperplane vector be defined by a set of real values $\{s_1, s_2, s_3 \cdots s_n\}$[1]. A tile is identified and associated with sizes such that the following conditions are met: DFG corresponding to the tile must be mappable on HyperCell fabric. Initiation interval $\Delta_{tile} = max(\Delta_{in_{loc}}, \Delta_{out_{mem,partial}}, \Delta_{cp,partial})$ is minimum, where $\Delta_{out_{mem,partial}}$ is the initiation interval due to stores to memory caused by RAW dependences only and $\Delta_{cp,partial}$ is the initiation interval due to critical path. Utilization of HyperCell FUs is maximum. Δ_{tile} corresponds to the initiation interval due to coarsening.

2) Creation of pHyperOp: Recall that a pHyperOp is a part of the iteration space that gets mapped onto a single HyperCell. A pHyperOp is created in a similar manner as outlined in step III-B1 where hyperplane vectors and sizes along the vectors are computed. A pHyperOp is created for locality of data using HyperCell's local storage and to effectively pipeline them. Hence, parallel pipelineable instances need to be identified within a pHyperOp. Let $H = \{h_1, h_2, h_3 \cdots h_n\}$ be the hyperplane vectors computed in the coarsened iteration space sorted in the decreasing order in data locality. Parallel instances are identified by finding another vector h_p such that $h_p \cdot d > 0$ for all non RAR dependence vectors d. The vector h_p is the normal to a wavefront along which parallel pipelineable instances of computation are found. Sizes are computed along all hyperplane vectors except one hyperplane vector which indicates the direction along which wavefront instances grow in number. This vector h_{load} is identified as the first vector from left in the set H such that h_{load} is linearly independent of h_p. Each dependence vector in a pHyperOp tile has an associated minimum tile size for least initiation interval, a steady state size of control sequences and a maximum size (in terms of the number of steady state instances) along hyperplane vectors H. The size of pHyperOp tile is chosen such that the following conditions are met: Steady state of control sequence of the pHyperOp can be accommodated in control storage. Initiation interval $\Delta_{pHyperOp} = max(\Delta_{in_{mem}}, \Delta_{out_{loc,partial}}, \Delta_{cp,partial})$ is minimum, where $\Delta_{out_{loc,partial}}$ is the initiation interval due to stores to local storage due to RAW dependences only and $\Delta_{cp,partial}$ is the initiation interval due to critical path. Ratio of maximum size of the pHyperOp to steady state size is maximum. For better load balancing in irregularly shaped iteration spaces, larger size maybe associated with hyperplane vectors that align with the boundaries of the iteration space. Effective initiation interval at the end of this step is $min(\Delta_{pHyperOp}, \Delta_{tile})$.

[1]A tile comprises lattice points enclosed by n pair of parallel hyperplanes. The dimension of the tile along a pair of parallel hyperplanes h_i is a real value s_i.

3) Creation and Mapping of HyperOp: Multiple pHyperOps are aggregated to create a HyperOp. While aggregating, we map pHyperOps to HyperCells keeping in mind the architectural limits such as memory and network bandwidth. A HyperOp uses the NoC for two purposes namely inter-pHyperOp communication and memory transactions. Recall that initiation interval is affected by paths (to memory or between pHyperOps) sharing network resources. Memory banks are available along each column of REDEFINE fabric. To minimize network resource sharing, pHyperOps are mapped onto HyperCells such that a set of HyperCells in a column access the bank of the column. Algorithm 1 describes the creation of a HyperOp and its mapping onto HyperCells. The algorithm either sets up a long pipeline of computations along a vector with most overlap in output data or multiple shorter pipelines based on available memory bandwidth. The shorter pipelines may further be shortened to reduce start and drain delays. Under-utilization at boundaries may be reduced by creating HyperOps along vectors that align with the boundaries of the iteration across multiple steps of compilation.

Algorithm 1 Algorithm to create HyperOp and map onto REDEFINE fabric

1: Map the first pHyperOp to the HyperCell in the top-left corner on REDEFINE fabric.
2: Identify a hyperplane vector v_i with non-RAR dependences in forward direction along v_i and along which most dependence edges from the first pHyperOp align.
3: Compute the memory bandwidth requirement of pHyperOps $m_{required}$ along v_i as $m_{required}$=⌈memory transactions required per instance of the pHyperOp / ($2\times\Delta_{min}$⌉)
4: Let $pipeline\ length = 2 \times m_{required}$. The pHyperOps along v_i are placed in a vertical-reverse S[10] manner such that the instances are $pipeline\ length$ HyperCells apart.
5: **if** $pipeline\ length > 0$ **then**
6: Sort other hyperplane vectors with non-RAR dependences in forward direction in decreasing order of no. of WAW/RAW dependence edges along each iteration vector.
7: From the vectors which have most dependence vectors aligned along them, identify vector v_j along which overlap of inter-HyperCell communication paths is minimum.
8: Place pHyperOps along v_j in between pairs of pHyperOps selected along v_i.
9: **end if**

4) Data Layout: Data layout corresponds to identifying a map of the addresses of inputs and outputs into address space of REDEFINE fabric to ensure minimum memory access conflicts, while distributing the data uniformly. For each pHyperOp placed along a column in REDEFINE fabric, each input is treated as a single block of data. The blocks corresponding to a pHyperOp can be placed in different banks of a set avoiding access conflicts. There maybe inputs shared across pHyperOps along a column. Storing a single copy of such data in a single bank leads to imbalance in storage among banks. On the other hand, multiple copies of such inputs can be created, one copy for each pHyperOp using the data following the dynamic single assignment model. This leads to balanced distribution but poor memory utilization. Instead, such input blocks are partitioned, replicated partially and distributed among banks of memory. Once a layout of inputs and outputs is identified, the sizes of

inputs and outputs are computed based on the size of each bank as explained in the algorithm 2. This indicates the size of a HyperOp in terms of tile dimensions and its corresponding pHyperOp. The maximum size of pHyperOps in terms of iteration space is computed after identifying the layout.

Algorithm 2 Algorithm to layout data for a HyperOp

1: **for all** input and output arrays of the DFG mapped onto HyperCells in each column **do**
2: **if** input array range is unique to each pHyperOp in a column or is partially shared between pHyperOps or is an output array range **then**
3: Place a copy of the array range for each pHyperOp in a round-robin manner to ensure least conflict with operand data of the HyperCell placed previously.
4: **end if**
5: **end for**
6: **for all** input array range being reused among a subset of pHyperOps in a column **do**
7: **if** Steady state of control sequences cannot be increased any further **then**
8: Replicate the entire input array-range being shared such that each pHyperOp gets an independent copy of inputs and place them similarly as the previous step.
9: **else if** the no. of rows of steady state control sequence can be increased n times **then**
10: Compute appropriate partition and replication factors for the data array-range such that the steady state size does not increase by a factor larger than n, conflicts are minimized and data distribution across banks is uniform.
11: **end if**
12: **end for**
13: Compute the size of arrays in each bank such that they add to the memory bank size.

C. HyperOp Scheduling

HyperOp instances need to be scheduled to be executed on REDEFINE fabric honoring dependence and resource constraints. HyperOp instances communicate with each other through memory. An order of execution among HyperOps is identified using hyperplane vectors such that data locality among successive HyperOps is maximized. Orchestrator is configured to honor the order of HyperOp execution. It computes runtime parameters such as loop bounds, input and output addresses. Launching a new HyperOp instance requires configuring this subset of parameters. Orchestrator is configured to prefetch operands for a successive HyperOp instances based on the schedule. Prefetch is expected to overlap with HyperOp execution. A subset of memory banks are used for prefetching while the rest are used by HyperCells. For data that needs to be fetched for a successive iteration, if volume of data required is order of magnitude less than computation, inputs required for a subsequent iteration can be fetched, reducing data transfer overheads. While computations can be launched at a rapid rate within a HyperCell with initiation interval as computed at the end of HyperOp creation step, configuration and synchronization time add to overall execution time. We hence compute the ratio of overall execution time to the computation time as the effective initiation interval and use it as a metric to evaluate the compiler's effectiveness.

IV. Results

In this section, we present experimental results to demonstrate the effectiveness of the proposed compilation process on REDEFINE accelerator enriched with HyperCells. We choose to implement a number of HPC kernels from the Polybench benchmark suite [11]. The kernels include loops from the domains of linear algebra and stencil computations. For each kernel we create 6 experimental setups with different problem sizes. These 6 setups are denoted as Setup 1 to 6 in figures 5, 6 and 7. Kernels *MATMUL, SYRK* and *SY2RK* are of $O(n^3)$ complexity, *GESUMMV* and *GEMVER* are of $O(n^2)$ complexity, *JACOBI 1-D, JACOBI 2-D* and *SIEDEL 2-D* are of complexity $O(mn)$, $O(mn^2)$ and $O(mn^2)$ respectively. Here n ranges from 256 to 8192 in power of 2 while m's values are 2, 10 and 100.

For experimentation, we use a template of the REDEFINE compute fabric with toroidal mesh topology. The CEs are arranged in 6 rows and 4 columns. Each CE is enriched with a HyperCell with 25 FUs. Each FU comprises an integer ALU and a single precision floating point unit. The local storage of each HyperCell consists of 8 banks of 64 deep register files. Each HyperCell has a configuration memory of 16KB. Memory is arranged as two blocks of memory banks. Each block is divided into 6 sets. Each set acts as the data storage for one row of four HyperCells. NoC is capable of delivering 4 load or store requests per cycle. Hence each set is divided into 4 banks, where each bank is 16KB in size. Orchestrator has two main storages: configuration storage (16KB in size) and context memory storage (20KB in size and holds context for four HyperOps at a time).

In order to demonstrate the effectiveness of our proposed compilation process, we present effective computation time, configuration time and synchronization time as fractions of total execution time for various kernels ($\frac{T_{comp}}{T_{exe}}$, $\frac{T_{config}}{T_{exe}}$ and $\frac{T_{sync}}{T_{exe}}$) in figure 5. Except for *JACOBI-1D*, the configuration times are significantly lower compared to total execution time due to large number of computations. Amount of computation in *JACOBI-1D* is not large enough to effectively amortize configuration overheads.

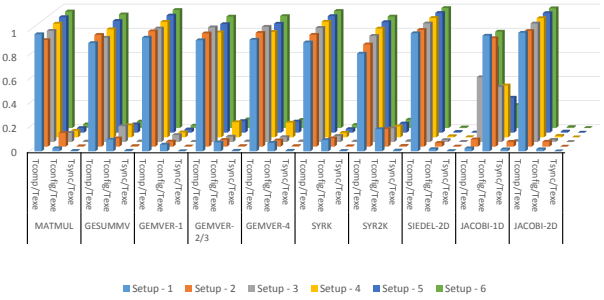


Fig. 5: Effective computation time as a fraction of overall execution time

In figure 6 we present efficiency of execution for the various kernels defined as the ratio of actual execution time of the kernel on REDEFINE and execution time of the same kernel

on an *ideal* machine with the same computational resources. Actual performance is affected by architectural artifacts of REDEFINE and HyperCell such as NoC bandwidth, memory bandwidth. We assume that the *ideal* machine does not suffer from these constraints. Due to effective reuse of operand data by the compiler, we achieve reasonable efficiency for most kernels. As problem size increases, configuration and synchronization overheads get amortized more effectively and the fraction of boundary HyperOps which under-utilize the REDEFINE fabric decreases, resulting in increased efficiency of execution. For larger problem sizes, kernels such as *MATMUL, SYRK, SYR2K* and *JACOBI-1D* achieve efficiency of 0.35 to 0.45. The remaining kernels are constrained by bandwidth limitations of HyperCells and increasing input output bandwidth of each HyperCell is the only way to improve efficiency.

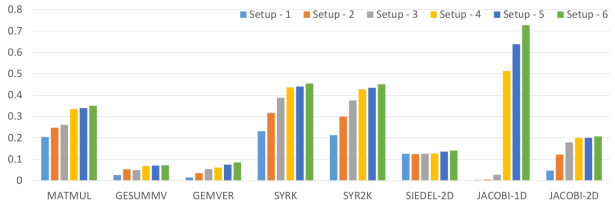


Fig. 6: Efficiency of execution for various kernels

Figure 7 shows percentage difference between the effective initiation interval and the architecture induced lower bound on initiation interval. This is a more realistic comparison to demonstrate the effectiveness of compilation. In this comparison variation in performance among kernels due to bandwidth constraints is nullified. Difference in observed initiation interval and theoretical initiation interval reduces with increased problem sizes. Only larger problem sizes are considered in figure 7. *MATMUL* and *SIEDEL-2D* show more than 40% difference with architecture induced initiation interval. This is attributed to the under-utilization of FUs in HyperCell. *GEMVER-2/3* and *GEMVER-4* also show similar behavior. This is attributed to the under-utilization of HyperCells in REDEFINE fabric. For JACOBI-1D, the high configuration overhead adversely affects the initiation interval. For other kernels the difference is less than 20%.

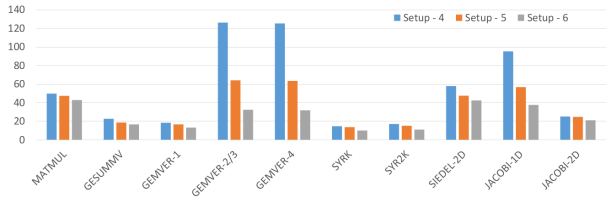


Fig. 7: Percentage difference between effective Initiation Interval & Architecture induced lower bound on Initiation Interval

V. Conclusion

In this paper, we present a compilation strategy for statically analyzable HPC kernels for a massively-parallel CGRA called REDEFINE. REDEFINE has multiple CEs with an NoC interconnect. It facilitates exploiting high degree of coarse grain and pipeline parallelism. The CEs comprise HyperCells for exploiting fine-grain ILP and pipeline parallelism. The compilation flow targets this architecture for accelerating kernels that exhibit regular parallelism such as affine loop nests with large bounds. It is designed using polyhedral compilation principles for partitioning the kernels among multiple HyperCells efficiently for parallelism while balancing utilization of memory and computation resources. While targeting REDEFINE, it is also responsible for generating appropriate configuration for orchestrating execution of coarse-grain computation entities as well as controlling data movement between an external host environment and REDEFINE. Results presented show the effectiveness of the compilation flow in utilizing computation resources.

References

[1] K. T. Madhu, S. Das, M. Krishna, N. Sivanandan, S. K. Nandy, and R. Narayan, "Synthesis of instruction extensions on hypercell, a reconfigurable datapath," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*. IEEE, 2014, pp. 215–224.

[2] S. Das, K. Madhu, M. Krishna, N. Sivanandan, F. Merchant, S. Natarajan, I. Biswas, A. Pulli, S. Nandy, and R. Narayan, "A framework for post-silicon realization of arbitrary instruction extensions on reconfigurable data-paths," *Journal of Systems Architecture*, vol. 60, no. 7, pp. 592–614, 2014.

[3] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The MOLEN polymorphic processor," *IEEE Trans. Computers*, vol. 53, no. 11, pp. 1363–1375, 2004. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TC.2004.104

[4] T. M. Brewer, "Instruction set innovations for the convey HC-1 computer," *IEEE Micro*, vol. 30, no. 2, pp. 70–79, 2010. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/MM.2010.36

[5] M. Shami and A. Hemani, "Partially reconfigurable interconnection network for dynamically reprogrammable resource array," in *ASIC, 2009. ASICON '09. IEEE 8th International Conference on*, 2009, pp. 122–125.

[6] M. Alle, K. Varadarajan, A. Fell, C. R. Reddy, J. Nimmy, S. Das, P. Biswas, J. Chetia, A. Rao, S. K. Nandy, and R. Narayan, "REDEFINE: Runtime reconfigurable polymorphic ASIC," *ACM Trans. Embedded Comput. Syst*, vol. 9, no. 2, 2009. [Online]. Available: http://doi.acm.org/10.1145/1596543.1596545

[7] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: ACM, 2008, pp. 101–113. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375595

[8] C. Alias, B. Pasca, and A. Plesco, "Automatic generation of fpga-specific pipelined accelerators," in *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, 2011, pp. 53–66.

[9] A. Fell, P. Biswas, J. Chetia, S. K. Nandy, and R. Narayan, "Generic routing rules and a scalable access enhancement for the network-on-chip RECONNECT," in *Annual IEEE International SoC Conference, SoCC 2009, September 9-11, 2009, Belfast, Northern Ireland, UK, Proceedings*, 2009, pp. 251–254. [Online]. Available: http://dx.doi.org/10.1109/SOCCON.2009.5398048

[10] N. Bansal, S. Gupta, N. Dutt, and A. Nicolau, "Analysis of the performance of coarse-grain reconfigurable architectures with different processing element configurations," in *Workshop on Application Specific Processors, held in conjunction with the International Symposium on Microarchitecture (MICRO), 2003*. Citeseer, 2003.

[11] "Polybench: Polyhedral benchmark suite," http://www.cs.ucla.edu/ pouchet/software/polybench/.