# Accelerating HPC Kernels with RHyMe - REDEFINE HyperCell Multicore

Saptarsi Das[1], Nalesh S.[1], Kavitha T. Madhu[1], S. K. Nandy[1] and Ranjani Narayan[2]

[1] CAD Laboratory, Indian Institute of Science, Bangalore
Email: {sdas, nalesh, kavitha}@cadl.iisc.ernet.in, nandy@serc.iisc.in
[2] Morphing Machines Pvt. Ltd., Bangalore
Email: ranjani@morphing.in

**Abstract.** In this paper, we present a coarse grained reconfigurable array (CGRA) designed to accelerate high performance computing (HPC) application kernels. The proposed CGRA named RHyMe, REDEFINE HyperCell Multicore, is based on the REDEFINE CGRA. It consists of a set of reconfigurable data-paths called HyperCells interconnected through a network-on-chip (NoC). The network of HyperCells serves as the hardware data-path for realization of HyperOps which are the basic schedulable entities in REDEFINE. RHyMe is specialized to accelerate regular computations like loops and relies on the compiler to generate the meta-data which are used at runtime for orchestrating the kernel execution. As a result, the compute hardware is simple and memory structures can be managed explicitly rendering a simple as well as efficient architecture.

## 1 Introduction

Modern high performance computing (HPC) applications demand heterogeneous computing platforms which consists of a variety of specialized hardware accelerators alongside general purpose processing (GPP) cores to accelerate compute intensive functions. When compared to GPPs, although accelerators give dramatically higher efficiency for their target applications they are not as flexible and performs poorly on other applications. Graphic processing units (GPU) can be used for accelerating a wide range of parallel applications. However GPUs are more suited for single instruction multiple data (SIMD) applications. Field programmable gate arrays (FPGA) may be used to generate accelerators on demand. Although this mitigates the flexibility issue involved with specialized hardware accelerators, the finer granularity of the lookup tables (LUT) in FPGAs leads to significantly high configuration time and low operating frequency. Coarse-grain reconfigurable architectures (CGRA) consisting of a pool of compute elements (CE) interconnected using some communication infrastructure overcomes the reconfiguration overheads of FPGAs while providing performance close to specialized hardware accelerators.

Examples of CGRAs include Molen Polymorphic Processor [13], Convey Hybrid-Core Computer [3], DRRA [12], REDEFINE [2], CRFU [10], Dyser [7], TRIPS [4]. REDEFINE as reported in [2] is a runtime reconfigurable polymorphic applications specific integrated circuit (ASIC). Polymorphism in ASICs is synonymous with attributing different functionalities to fixed hardware in space and time. REDEFINE is a massively parallel distributed system, comprising a set of Compute Elements (CEs) communicating over a Network-on-Chip (NoC) [6] using messages. REDEFINE follows a macro data-flow execution model at the level of macro operations (also called HyperOps). HyperOps are convex partitions of the application kernels data-flow graph, and are composition of one or more multiple-input-multiple-output (MIMO) operations. The ability of REDEFINE to provision CEs to serve as composed data-paths for MIMO operations over the NoC is a key differentiator that sets aside REDEFINE from other CGRAs.

REDEFINE exploits temporal parallelism inside the CEs, while spatial parallelism is exploited across CEs. The CE can be an instruction-set processor or a specialized custom function unit (CFU) or a reconfigurable data-path. In this paper, we present the REDEFINE CGRA with HyperCells [8], [5] as CEs so as to support parallelism of all granularities. HyperCell is a reconfigurable data-path that can be configured on demand to accelerate frequently occurring code segments. Custom data-paths, dynamically set up within HyperCells enable exploitation of fine-grain parallelism. Coarse grained parallelism is exploited across various HyperCells. We refer to this architecture, ie, HyperCells as CEs in REDEFINE as REDEFINE HyperCell Multicore (RHyMe). In this paper we present the RHyMe hardware comprising both the resources for computation and runtime orchestration. The paper is structured as follows. The execution model and a brief overview of compilation flow employed are described in 2. Section 3 presents the hardware architecture of RHyMe. Sections 4 and 5 present some results and conclusions of the paper.

## 2 Execution Model & Compilation Framework

In this section we present a brief overview of the execution model of the RHyMe architecture followed by a high level description of the compilation flow. RHyMe is a macro data-flow engine comprising three major hardware components namely compute fabric, orchestrator and memory (see figure 1). As mentioned previously, the compute fabric is composed of HyperCells. An application kernel to be executed on RHyMe comprises convex schedulable entities called HyperOps that are executed atomically. Each HyperOp is composed of pHyperOps, each of which is mapped onto a HyperCell of RHyMe. In the scope of this exposition, we consider loops from HPC applications as the kernels for acceleration on RHyMe. Computation corresponding to a loop in the kernel is treated as a HyperOp and its iteration space is divided into a number of HyperOp instances. Execution of a HyperOp on RHyMe involves three major phases namely, configuration of the hardware resources (HyperCells and orchestrator), execution of

the instances of a HyperOp by binding runtime parameters and synchronization among HyperOp instances. HyperOp instances are scheduled for execution when the following conditions are met.

- HyperCells executing the HyperOp instance and the orchestrator are configured.
- Operands of the HyperOp instance is available in REDEFINE's memory.
- HyperCells to which the HyperOp is mapped are free to execute the HyperOp.
- Runtime parameters of HyperCells and orchestrator are bound.

A HyperOp requires the HyperCells and orchestrator to be configured when launching the first instance for execution. Subsequent instances require only the runtime parameters to be sent to HyperCells and orchestrator as explained in detail in section 3.

In [9], the authors have presented a detailed description of the execution model and the compilation flow for RHyMe. In the following section we discuss the hardware architecture of RHyMe in greater detail.

## 3 REDEFINE HyperCell Multicore (RHyMe) Architecture

In this section we present the architectural details of RHyMe. As mentioned in section 2, RHyMe has three major components namely *Compute Fabric, Memory, Orchestrator*.

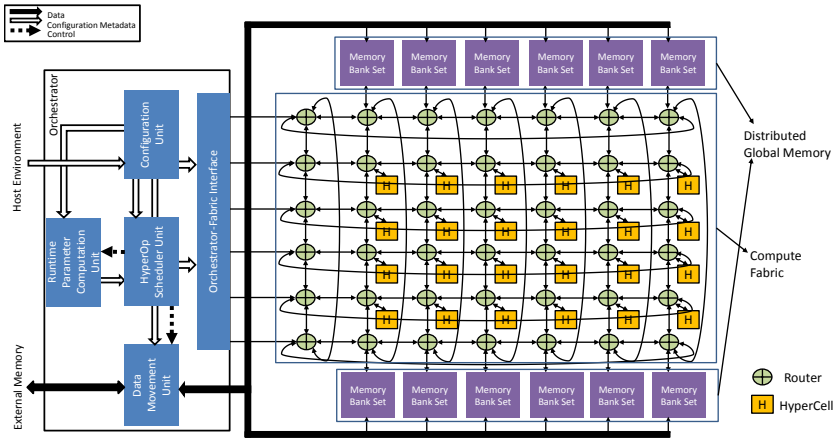


**Fig. 1.** REDEFINE HyperCell Multicore (RHyMe)

### 3.1 Compute Fabric

The Compute fabric of RHyMe consists of a set of HyperCells interconnected via an NoC.

**HyperCell:** Micro-architectural details of HyperCell is presented in [5] and [8]. The authors had presented HyperCell as a hardware platform for realization of multiple input multiple output (MIMO) macro instructions. In this exposition, we adopted the micro-architecture of HyperCell as the CEs in RHyMe. A HyperCell has a controller and a local storage along side a reconfigurable data-path (refer to figure 2). The reconfigurable data-path of HyperCell comprises a set of compute units (CU) connected by a circuit-switched interconnect (refer figure 2). The CUs and switches can be configured to realize various data-flow graphs (DFG). The reconfigurable data-path of HyperCell is designed to support pipelined execution of instances of such DFGs. Flow control of data in this circuit switched network is ensured by a light weight ready-valid synchronizing mechanism ensuring that the data is not overwritten until it is consumed. This mechanism makes HyperCell tolerant to the non-deterministic latencies in data-delivery at CUs' inputs. Local storage of a HyperCell consists of a set of register files, each with one read port and one write port. Each operand data in a register file is associated with a valid bit. An operand can only be read if its corresponding valid bit is set. Likewise, an operand can be written to a register location only if the corresponding valid bit is reset.

Controller is responsible for delivering both configuration and data inputs to the HyperCell's data-path and transferring results to RHyMe's memory. It orchestrates four kinds of data transfers:

- Load data from RHyMe's memory to HyperCell's local data storage.
- Load input from HyperCell's local storage to HyperCell's reconfigurable data-path.
- Store outputs from HyperCell's reconfigurable data-path to RHyMe's memory.
- Store outputs from HyperCell's reconfigurable data-path to local storage for reuse in subsequent instances of the realized DFG.

These data transfers are specified in terms of a set of four control-sequences. The control-sequences are stored in dedicated storages inside the HyperCells. The HyperCell controller comprises four FSMs that process these control sequences. The aforementioned control sequences together realize a modulo-schedule[11] of the DFG instantiated on the data-path. Each control sequence contains a prologue and epilogue, both of which are executed once and a steady state executed multiple times. The sequences are generated in a parametric manner. The runtime parameters are *start* and *end* pointers for *prologue*, *steady state* and *epilogue*, *base addresses of the inputs and outputs*, *number of times steady state is executed* and *epilogue spill*. At runtime, the reconfigurable data-path of HyperCell and its controller are configured for the first HyperOp instance. To facilitate execution of different instances of a HyperOp, the runtime parameters are bound to HyperCell, once per instance.

A control sequence is realized as a set of control words interpreted at runtime. HyperCells' control sequences are also responsible for communicating outputs between HyperCells. In order to facilitate inter-HyperCell communication, the local storage of a HyperCell is write-addressable from other HyperCells. An acknowledgement based synchronization scheme is realized to maintain flow control during communication among HyperCells. Further, control words are grouped together to send multiple operand data together to a remote HyperCell's local storage increasing the granularity of synchronization messages.

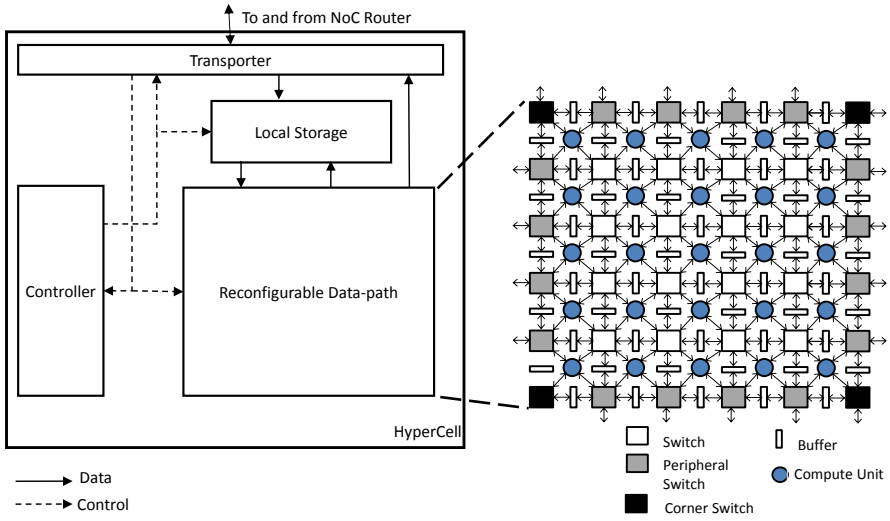


**Fig. 2.** Reconfigurable data-path of HyperCell

**Network on Chip:** The NoC of RHyMe provides the necessary infrastructure for inter-HyperCell communication (refer to figure 1), communication between Memory and HyperCells and communication between Orchestrator and HyperCells. Authors have presented detail micro-architectural descriptions of the NoC in [6]. We adopted the same NoC for RHyMe. The NoC consists of routers arranged in a toroidal mesh topology. Each router is connected to four neighbouring routers and a HyperCell. Packets are routed from a source router to a destination router based on a deterministic routing algorithm namely the *west-first algorithm* [6]. There are four types of packets handled by the NoC namely load/store packets, configuration packets, synchronization packets and inter-HyperCell packets. The load/store packets carry data between the HyperCells and memory. Configuration packets contain configuration meta-data or runtime parameters sent by the orchestrator to the HyperCells. Synchronization packets are sent from each HyperCell to the orchestrator to indicate end of computation for the current pHyperOp being mapped to that HyperCell. Inter-

HyperCell packets consist of data transmission packets between the HyperCells and the acknowledgement packets for maintaining flow control in inter-HyperCell communications. A transporter module acts as the interface between HyperCell and router and is responsible for packetizing data communicated across HyperCells as well as load/store requests (refer to figure 2).

### 3.2 Memory

A distributed global memory storage is provisioned in RHyMe as shown in figure 1. This memory serves as input and output storage to be used by successive computations. This memory can be viewed as overlay memory explicitly managed by the orchestrator. All inputs required by a particular computation is loaded into memory before execution starts. It is implemented as a set of memory banks. Multiple logical partitions are created at compile time. One of the partitions is used as operand data storage and the others act as prefetch buffers for subsequent computations.

### 3.3 Orchestrator

Orchestrator is responsible for the following activities.

- Data movement between RHyMe's memory and external memory interface
- Computation of runtime parameters
- Scheduling HyperOps and HyperOp instances and synchronization between successive instances or HyperOps
- Configuration of HyperCell and the other orchestrator modules

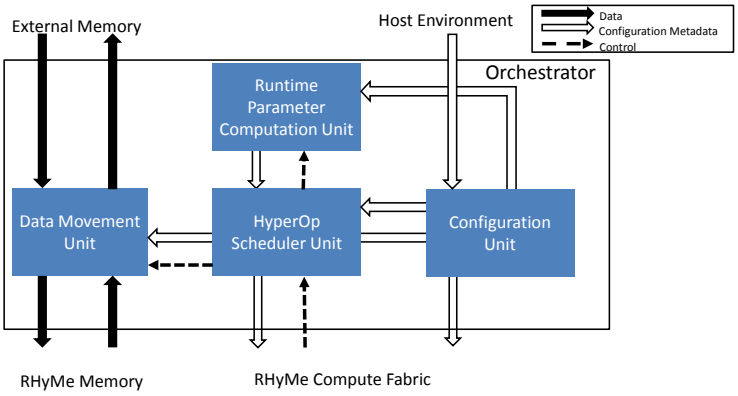


**Fig. 3.** Interactions between modules of the Orchestrator

The aforementioned tasks are carried out by three modules of the orchestrator. Brief descriptions of the modules are given below. Figure 3 depicts the interactions between these various modules.

**Configuration Unit:** Configuration Unit is responsible for the initial configuration of the HyperCells as well as the other units of the orchestrator listed below. The configuration metadata for the HyperCells is delivered through the NoC. Configuration metadata for the other modules of orchestrator is delivered directly to the recipient module. These metadata transactions are presented in figure 3.

**Data Movement Unit:** As seen in figure 3, the Data Movement Unit (DMU) is responsible for managing data transactions between the external memory and RHyMe's memory. It is configured by the configuration unit such that fetching data for computations and write-back to the external memory overlaps with computation. DMU Configuration corresponds to a set of load and store instructions. The overlap in fetching operand data with computation is accomplished by dividing the address space of RHyMe's memory into partitions. As mentioned in section 3.2, during execution of one HyperOp on the compute fabric, one partition of the address space acts as operand storage for the active HyperOp and the rest act as a prefetch buffer for subsequent HyperOp instances. Each partition of RHyMe's memory is free to be written into when its previous contents have been consumed. This is achieved through partition-by-partition synchronization at the HyperOp Scheduler unit. The compilation flow is responsible for creating appropriate configuration meta-data to perform the aforementioned activities.

**Runtime Parameter Computation Unit:** Runtime Parameter Computation Unit (RPCU) is responsible for computation of HyperCell's runtime parameters listed in section 3.1. The RPCU computes the runtime parameters for a successive instance of a HyperOp while HyperCells are busy computing previous instances, thus amortizing the overheads of parameter computation. The runtime parameter computation is expressed as a sequence of ALU and branch operations. The RPCU comprises a data-path that processes these instructions. Similar to the DMU, the RPCU works in synchrony with the HyperOp scheduler unit. The runtime parameters computed are forwarded to the HyperOp scheduler unit which in turn binds them to the compute fabric (see figure 3).

**HyperOp Scheduler Unit:** HyperOp Scheduler Unit (HSU) is responsible for scheduling instances of a HyperOp onto the compute fabric for execution. HSU waits for conditions listed previously to be met to trigger the execution of a new HyperOp or its instance on HyperCells. When all the HyperCells are free to execute a new HyperOp instance, the scheduler unit binds a new set of runtime parameters to HyperCells to enable execution of the instance.

## 4 Results

In this section, we present experimental results to demonstrate the effectiveness of the RHyMe architecture. HPC kernels from the Polybench benchmark suite [1] were employed in this evaluation. The kernels are from the domains of linear

**Table 1.** Computational complexity and problem sizes of the kernels

| Setup | matmul $O(n^3)$ | gesummv $O(n^2)$ | gemver $O(n^2)$ | syrk $O(n^3)$ | syr2k $O(n^3)$ | jacobi1d $O(mn)$ | jacobi2d $O(mn^2)$ | siedel2d $O(mn^2)$ |
|---|---|---|---|---|---|---|---|---|
| Setup1 | $n = 256$ | $n = 256$ | $n = 256$ | $n = 256$ | $n = 256$ | $m = 2$, $n = 256$ | $m = 10$, $n = 256$ | $m = 10$, $n = 256$ |
| Setup2 | $n = 512$ | $n = 512$ | $n = 512$ | $n = 512$ | $n = 512$ | $m = 2$ $n = 512$ | $m = 10$, $n = 512$ | $m = 10$, $n = 512$ |
| Setup3 | $n = 1024$ | $n = 1024$ | $n = 1024$ | $n = 1024$ | $n = 1024$ | $m = 10$, $n = 1024$ | $m = 20$, $n = 1024$ | $m = 20$, $n = 1024$ |
| Setup4 | $n = 2048$ | $n = 2048$ | $n = 2048$ | $n = 2048$ | $n = 2048$ | $m = 100$, $n = 2048$ | $m = 20$, $n = 2048$ | $m = 20$, $n = 2048$ |
| Setup5 | $n = 4096$ | $n = 4096$ | $n = 4096$ | $n = 4096$ | $n = 4096$ | $m = 100$, $n = 4096$ | $m = 100$, $n = 4096$ | $m = 100$, $n = 4096$ |
| Setup6 | $n = 8192$ | $n = 8192$ | $n = 8192$ | $n = 8192$ | $n = 8192$ | $m = 100$, $n = 8192$ | $m = 100$, $n = 8192$ | $m = 100$, $n = 8192$ |

algebra and stencil computations. For each kernel we create 6 experimental setups with different problem sizes listed in table 1. For these experiments, we have selected a template of the RHyMe compute fabric with HyperCells arranged in 4 rows and 6 columns. Each HyperCell comprises 25 compute units (CU), each consisting of an integer ALU and a single precision floating point unit (FPU). The local storage of each HyperCell consists of 8 banks of 64 deep register files. A HyperCell has a configuration memory of 16 KB. As mentioned in section 3.2, RHyMe's distributed global memory is divided in 12 sets. 2 sets on either side of the fabric act as data storage for a column of four HyperCells. A set consists of 4 banks of 16 KB each with one router giving access to 4 banks. The overall storage capacity is hence 768 KB. Since each router is connected to 4 banks on either side, 4 loads/stores can be serviced per request. Thus, each load/store request from a HyperCell can address four words from the memory. RHyMe's orchestrator has a *configuration storage* for different components of the orchestrator and *HyperOp configuration storage* corresponding to HyperCell's configuration metadata. The former is of size 16 KB and latter is 20 KB in size and can hold HyperCell configuration for four HyperOps at a time.

In this exposition, RHyMe is assumed embedded in a heterogeneous multicore machine with a shared L2 cache. The L2 cache size is 512 KB. The data movement unit (DMU) of RHyMe's orchestrator interfaces directly with the shared L2 cache. Figure 4 shows the steps involved in executing a HyperOp in RHyMe. We refer to the data transfer latency as $T_{mem}$, the computation latency as $T_{comp}$, the runtime parameter binding latency as $T_{param}$ and the synchronization latency as $T_{sync}$. For maximizing performance, $(max(T_{mem}, (T_{param} + T_{comp})) + T_{sync})$ should be minimized. Given a kernel and a fixed number of HyperCells, $T_{comp}$, $T_{sync}$ and $T_{param}$ are fixed. Hence, maximizing performance requires $T_{mem}$ to be less than or equal to $(T_{param} + T_{comp})$ such that the computation and parameter binding steps completely overlap the data transfer step. $T_{mem}$ can be reduced by increasing the bandwidth between L2 cache and RHyMe memory. We have hence

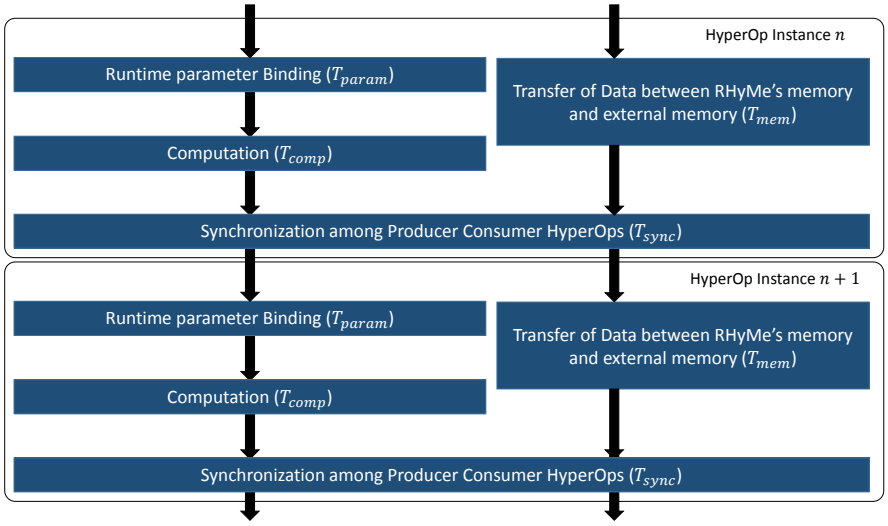


**Fig. 4.** Execution flow of HyperOps

conducted experiments for two different configurations with the results given in table 2. In the first configuration (referred to as MemSetup1), L2 has cache line size of 64B while DMU to RHyMe memory interface is capable of handling one word per cycle. In the second configuration referred to as MemSetup2, the L2 cache line size is doubled to 128B and the DMU to RHyMe memory interface is capable of handling two words per cycle. In table 2 we present $(T_{param} + T_{comp})$ and $T_{mem}$ for various kernels. We define a metric $\eta = \frac{T_{param}+T_{comp}-T_{mem}}{max((T_{param}+T_{comp}),T_{mem})}$ that measures the effectiveness of overlap of the data transfer step with the compute and configuration step. Figure 5 presents $\eta$ for various kernels for the two different configurations. A positive value in figure 5 indicates the fact that data transfer is completely hidden. It can be observed that, increasing the bandwidth between L2 cache and RHyMe memory (MemSetup2) helps in increasing $\eta$ for most of the kernels. However, in case of *gesummv*, *gemver* and *jacobi2d*, $\eta$ is negative with MemSetup2 as well. In case of *jacobi2d*, this is attributed to relatively large amount of data consumed and produced per HyperOp. In case of *gesummv*, *gemver*, both the volume of data required is comparable with the volume of computation in each HyperOp whereas other kernels require an order less volume of data the volume of computation. In case of these two kernels, $T_{mem}$ dominates the overall execution time and no amount of reasonable increase in memory bandwidth can hide it effectively (refer table 2).

In table 3 we present the computation times for various kernels as fractions of their respective overall execution times. We observe that as problem size increases, the fraction grows and becomes close to one. This indicates the effective amortization of configuration and synchronization latencies at larger problem

**Table 2.** Comparison of computation vs memory transaction latencies (per HyperOp) for various kernels

| **Kernels** | $T_{comp}$ | $T_{mem}$ | | **Effectiveness of overlap $\eta$** | |
|---|---|---|---|---|---|
| | | **MemSetup1** | **MemSetup2** | **MemSetup1** | **MemSetup2** |
| matmul | 5234 | 6447 | 3223 | -0.188 | 0.384 |
| gesummv | 3990 | 92776 | 46388 | -0.957 | -0.914 |
| gemver | 5760 | 139137 | 69570 | -0.959 | -0.917 |
| syrk | 5234 | 5843 | 2935 | -0.104 | 0.439 |
| syr2k | 2728 | 3710 | 1874 | -0.265 | 0.313 |
| jacobi1d | 15402 | 164 | 84 | 0.989 | 0.994 |
| jacobi2d | 1596 | 4915 | 2497 | -0.675 | -0.361 |
| siedel2d | 15456 | 14745 | 7385 | 0.046 | 0.522 |

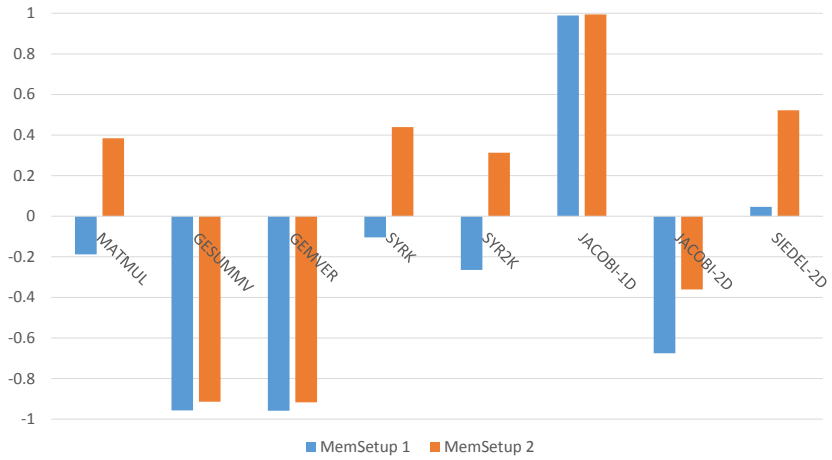


**Fig. 5.** $\eta$ for various kernels

**Table 3.** Computation time as fraction of overall execution time for different kernels

| | Problem Size | matmul | gesummv | gemver | syrk | syr2k | siedel 2d | jacobi 1d | jacobi 2d |
|---|---|---|---|---|---|---|---|---|---|
| MemSetup1 | Setup1 | 0.86 | 0.798 | 0.798 | 0.853 | 0.919 | 0.799 | 0.489 | 0.667 |
| | Setup2 | 0.971 | 0.888 | 0.856 | 0.969 | 0.981 | 0.895 | 0.735 | 0.75 |
| | Setup3 | 0.993 | 0.969 | 0.96 | 0.992 | 0.993 | 0.984 | 0.575 | 0.909 |
| | Setup4 | 0.996 | 0.989 | 0.985 | 0.996 | 0.994 | 0.995 | 0.436 | 0.973 |
| | Setup5 | 0.997 | 0.997 | 0.996 | 0.997 | 0.995 | 0.999 | 0.436 | 0.998 |
| | Setup6 | 0.997 | 0.999 | 0.999 | 0.997 | 0.995 | 0.999 | 0.436 | 0.999 |
| MemSetup2 | Setup1 | 0.907 | 0.799 | 0.798 | 0.916 | 0.948 | 0.806 | 0.324 | 0.667 |
| | Setup2 | 0.982 | 0.887 | 0.856 | 0.982 | 0.985 | 0.944 | 0.581 | 0.75 |
| | Setup3 | 0.994 | 0.969 | 0.959 | 0.994 | 0.992 | 0.991 | 0.575 | 0.927 |
| | Setup4 | 0.996 | 0.989 | 0.985 | 0.996 | 0.993 | 0.997 | 0.599 | 0.979 |
| | Setup5 | 0.996 | 0.997 | 0.996 | 0.996 | 0.993 | 0.999 | 0.605 | 0.999 |
| | Setup6 | 0.996 | 0.999 | 0.999 | 0.996 | 0.993 | 0.999 | 0.605 | 0.999 |

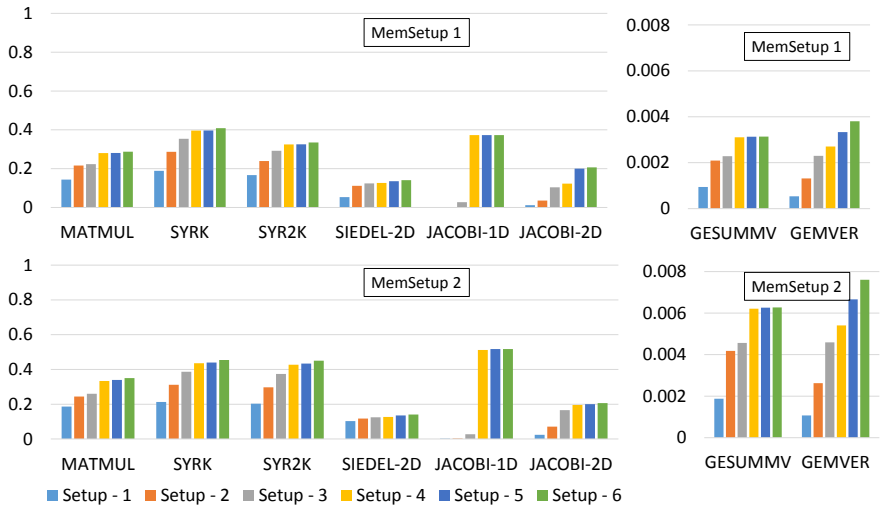


**Fig. 6.** Efficiency of execution for various kernels on RHyMe. *gesummv* and *gemver* plotted separately due to order of magnitude difference in efficiency

sizes. This can be attributed to improvement in temporal utilization of the resources in compute fabric of RHyMe with increase in problem size. An exception to this trend is *jacobi1d*. In case of *jacobi1d*, even for the larger problem sizes (setup 4, 5 and 6), the amount of computation involved is not large enough to effectively amortize configuration overheads. Hence we observe a significant configuration overhead for *jacobi1d*. For any given kernel, efficiency is measured as the ratio of actual performance of the kernel on RHyMe and theoretical peak performance. Actual performance is affected by various architectural artifacts of RHyMe such as NoC bandwidth, RHyMe memory bandwidth, HyperCell's local storage bandwidth. While measuring peak performance we simply consider the parallelism available in each kernel and the number of basic operations that can be executed in parallel.

The efficiency for various kernels with the experimental setups in table 1 can be seen in figure 6. With increasing problem sizes, efficiency increases since configuration and synchronization overheads are more effectively amortized (refer to table 3). As mentioned previously, in case of *gesummv* and *gemver*, the overwhelming dominance of data transfer latency leads to less than 1% efficiency. For the other kernels, we achieve efficiencies ranging from 14% to 40% with large problem sizes.

Table 4 lists the performance for different kernels for largest problem sizes (Setup6) in terms of Giga Floating Point Operations per Second (GFLOPS) at 500MHz operating frequency. The table also presents the improvement in performance achieved by increasing the bandwidth between external L2 and RHyMe's memory. Against a theoretical peak performance of 300 GFLOPs, for

most kernels we achieve performance ranging from 42 to 136 GFLOPS. Due to the reasons mentioned previously, *gesummv* and *gemver* show upto 2 GFLOPS performance and are unsuitable for execution on RHyMe platform.

**Table 4.** Performance of various kernels on RHyMe measured at two different configurations: MemSetup1 & MemSetup2

| **Kernels** | **Performance in GFLOPS** | | **% Increase** |
|---|---|---|---|
| | **MemSetup1** | **MemSetup2** | |
| matmul | 86.104 | 105.035 | 21.986 |
| gesummv | 0.941 | 1.882 | 99.957 |
| gemver | 1.141 | 2.281 | 99.968 |
| syrk | 122.522 | 136.259 | 11.212 |
| syr2k | 100.363 | 135.081 | 34.592 |
| jacobi1d | 42.125 | 42.126 | 0.0026 |
| jacobi2d | 111.790 | 155.152 | 38.788 |
| siedel2d | 61.901 | 61.918 | 0.028 |

## 5 Conclusion

In this paper we presented the architectural details of REDEFINE HyperCell Multicore (RHyMe). RHyMe is a data-driven coarse-grain reconfigurable architecture designed for fast execution of loops in HPC applications. RHyMe facilitates exploitation of spatial and temporal parallelism. The CEs of RHyMe aka HyperCells offer reconfigurable data-path for realizing MIMO operations and alleviate the fetch-decode overheads of a fine-grain instruction processing machine. HyperCell's reconfigurable data-path offers the ability to exploit high degree of fine-grain parallelism while the controller of HyperCell enables exploiting pipeline parallelism. Multitude of HyperCells that can communicate with each other directly enable creation of large computation pipelines. RHyMe employs a lightweight configuration, scheduling and synchronization mechanism with minimal runtime overheads as is evident from the results presented.

## References

1. Polybench: Polyhedral benchmark. www.cs.ucla.edu/p̃ouchet/software/polybench/
2. Alle, M., Varadarajan, K., Fell, A., Reddy, C.R., Nimmy, J., Das, S., Biswas, P., Chetia, J., Rao, A., Nandy, S.K., Narayan, R.: REDEFINE: Runtime reconfigurable polymorphic ASIC. ACM Trans. Embedded Comput. Syst 9(2) (2009), http://doi.acm.org/10.1145/1596543.1596545
3. Brewer, T.M.: Instruction set innovations for the convey HC-1 computer. IEEE Micro 30(2), 70–79 (2010), http://doi.ieeecomputersociety.org/10.1109/MM.2010.36

4. Burger, D., Keckler, S., McKinley, K., Dahlin, M., John, L., Lin, C., Moore, C., Burrill, J., McDonald, R., Yoder, W.: Scaling to the end of silicon with edge architectures. Computer 37(7), 44–55 (July 2004)
5. Das, S., Madhu, K., Krishna, M., Sivanandan, N., Merchant, F., Natarajan, S., Biswas, I., Pulli, A., Nandy, S., Narayan, R.: A framework for post-silicon realization of arbitrary instruction extensions on reconfigurable data-paths. Journal of Systems Architecture 60(7), 592–614 (2014)
6. Fell, A., Biswas, P., Chetia, J., Nandy, S.K., Narayan, R.: 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. pp. 251–254 (2009), http://dx.doi.org/10.1109/SOCCON.2009.5398048
7. Govindaraju, V., Ho, C.H., Sankaralingam, K.: Dynamically specialized datapaths for energy efficient computing. In: HPCA. pp. 503–514. IEEE Computer Society (2011), http://dx.doi.org/10.1109/HPCA.2011.5749755
8. Madhu, K.T., Das, S., Krishna, M., Sivanandan, N., Nandy, S.K., Narayan, R.: Synthesis of instruction extensions on hypercell, a reconfigurable datapath. In: Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on. pp. 215–224. IEEE (2014)
9. Madhu, K.T., Das, S., Nalesh, S., Nandy, S.K., Narayan, R.: Compiling HPC kernels for the REDEFINE CGRA. In: 17th IEEE International Conference on High Performance Computing and Communications, HPCC 2015, 7th IEEE International Symposium on Cyberspace Safety and Security, CSS 2015, and 12th IEEE International Conference on Embedded Software and Systems, ICESS 2015, New York, NY, USA, August 24-26, 2015. pp. 405–410 (2015), http://dx.doi.org/10.1109/HPCC-CSS-ICESS.2015.139
10. Noori, H., Mehdipour, F., Inoue, K., Murakami, K.: Improving performance and energy efficiency of embedded processors via post-fabrication instruction set customization. The Journal of Supercomputing 60(2), 196–222 (2012), http://dx.doi.org/10.1007/s11227-010-0505-0
11. Rau, B.R.: Compiling hpc kernels for the redefine cgra. In: Proceedings of the 27th annual international symposium on Microarchitecture. pp. 63–74. MICRO 27, ACM, New York, NY, USA (1994), http://doi.acm.org/10.1145/192724.192731
12. Shami, M., Hemani, A.: Partially reconfigurable interconnection network for dynamically reprogrammable resource array. In: ASIC, 2009. ASICON '09. IEEE 8th International Conference on. pp. 122–125 (2009)
13. Vassiliadis, S., Wong, S., Gaydadjiev, G., Bertels, K., Kuzmanov, G., Panainte, E.M.: The MOLEN polymorphic processor. IEEE Trans. Computers 53(11), 1363–1375 (2004), http://doi.ieeecomputersociety.org/10.1109/TC.2004.104