

Coarse Grain Reconfigurable Architecture Proposal

Tian Zhao, Nipun Agarwala, David Koeplinger, Yaqi Zhang, Artem Vasilyev, and Raghu Prabhakar

1. Motivation and Background

Machine learning algorithms are prevalent today in various systems and services like battery life optimization and recommendation systems. The growing availability and usage of data has found CPUs wanting in performance. Since machine learning applications often exhibit high degrees data level parallelism with only small amounts of control flow, there has been an increasing trend to move away from traditional control-based, general purpose CPUs.

GPUs offer significant speedups over CPUs for highly parallel applications, but high power consumption by GPUs has made the industry pursue other accelerator options simultaneously. Custom hardware accelerator designs can achieve better performance per energy than GPUs and can more flexibly capture large amounts of data parallelism. In the past, custom hardware accelerators have typically been implemented as either ASICs or as designs on field programmable gate arrays (FPGAs). ASICs offer high performance and energy efficiency, but are limited to accelerating only a single application or predefined set of applications. ASICs also have extremely large non-recurring costs associated with design, verification, and tapeout. FPGAs designs are much cheaper and somewhat faster to implement and verify, but FPGAs suffer large area, power, and performance overheads relative to ASICs [13]. These overheads are largely due to the routing required between single bit lookup tables and registers that make up much of the FPGA compute fabric.

Course-grain reconfigurable architectures (CGRAs) have previously been proposed to bridge the performance per energy gap between ASICs and FPGAs. In addition to LUTs for arbitrary boolean logic, CGRAs include a large number of configurable, dedicated ALUs, multipliers, and other dedicated compute units. These units use word widths much larger than the single bit LUTs available in FPGAs, and thus require significantly less routing delay and energy and better performance and energy efficiency than FPGAs.

The primary problem with past proposed CGRAs has been in their programming model. GPUs have seen success in industry largely because of associated languages like OpenCL and CUDA which are relatively easy to learn for CPU software programmers compared to hardware description languages (HDLs) traditionally used for ASIC and FPGA designs. Previous work on CGRAs typically have not addressed the programmability of the proposed architecture, thus stifling their adoption. To address this issue, in this paper we propose a new coarse-grained reconfigurable architecture for the acceleration of machine learning applications, designed in conjunction with a high level programming model. Our architecture is designed to support programs represented in the Delite Hardware Definition Language (DHDL), a template-based intermediate language for hardware designs which can be automatically generated from high level languages such as OptiML, a high level, functional DSL for machine learning. We also show evaluation and refinement of our proposed architecture using a set of benchmark applications from the machine learning and financial analytics domains.

2. Related Work

A variety of CGRA architectures have previously been proposed. ADRES [6] is a 2D array of functional units which can be configured both as a dataflow or VLIW processor. ADRES was designed alongside the DRES compiler to be programmed with ANSI C. NeuroCGRA [15] is a dynamic architecture that allows for morphing between conventional CGRAs and neural network applications. It addresses the issue that most of existing CGRAs do not provide strong support for neural networks. PipeRench [14] is an interconnected network of configurable logic and storage elements. The advantage of PipeRench is that it uses a technique called pipeline reconfiguration that helps improve compilation time, reconfiguration time, and forward compatibility. The Chameleon project [16] is a coarse-grained reconfigurable core for DSP algorithms in wireless devices. REMARC [17] is a reconfigurable coprocessor coupled with a main RISC

processor. It consists of a global control unit and 64 programmable logic blocks called nano-processors. RaPiD [18] is a coarse-grained configurable architecture for constructing deep computational pipelines. It provides a large number of ALUs, multipliers, registers and memory modules, which can be taken use of to configure appropriate pipelined datapath. DySER [19] is an execution model that dynamically specializes hardware to match application phases on a heterogeneous array of computation units interconnected with a circuit-switched network. Garp [20] is a hybrid architecture that combines a MIPS processor with a reconfigurable coprocessor on the same die to benefit from the features of both. It is designed to fit into an ordinary processing environment. Although this collection of CGRAs covers a good variety of different applications, the common problem across all proposed architectures is that the proposed programming models are too low level and inflexible for widespread adoption. The ADRES architecture supports ANSI-C, but the architecture itself can only implement a small dataflow graph of function units. The PipeRench compiler takes a single assignment, C-like programming language as input. The remaining mentioned architectures are programmed with assembly or directly at the architecture configuration level. Our CGRA is codesigned with an intermediate hardware language which can be generated from high level, domain-specific languages.

A great deal of previous work has also looked at accelerating machine learning applications with FPGAs. Zhang et al. [10] give a procedure for optimizing deep neural network designs on FPGAs using roofline model driven design space exploration. TABLA [11] uses an overlay architecture, much like a CGRA, to simplify the generation of FPGA-based accelerators for machine learning tasks using stochastic gradient descent. Previous work has also been done on manual implementations of accelerators for machine learning algorithms like *k*-Means clustering [12] and convolutional neural networks [9]. However, routing energy and delay costs in FPGAs fundamentally limit their performance and energy efficiency compared to coarse-grained architectures.

Farabet, et al [9] examined accelerating convolutional neural networks for live image processing using both FPGA and ASIC designs. While the FPGA was capable of live processing for filters up to 21x21 pixels in size, the ASIC design was projected to use roughly 15x less power and to be capable of live processing for much more complex tasks. Other work has also looked at the design of partially configurable ASIC designs for accelerating a select set of machine learning applications. For instance, the DaDianNao multichip architecture [7] is able to achieve speedups of up to 450x and energy improvements of up to 150x over GPU on convolutional and deep neural networks. The PuDianNao architecture [8] was designed as a partially reconfigurable accelerator focused on computation patterns from seven core machine learning techniques, and is able to achieve energy improvements up to 128x over an NVIDIA GPU. While these ASIC architectures can achieve significant speedups and energy improvements relative to GPUs, they support only the applications they were designed to accelerate. Furthermore, these ASIC architectures do not have a true programming model. Users can only specify application parameters and, in some cases, very minor variations on the original algorithm. Our proposed CGRA architecture offers significantly more configurability than ASIC designs, and thus is capable of implementing a much larger set of applications. Our programming model for the CGRA reflects this versatility while also entirely abstracting architecture details away from the user.

3. Delite Hardware Definition Language

In programming the CGRA, we use the Delite Hardware Definition Language (DHDL) as an intermediate language for describing hardware datapaths. A DHDL program describes a dataflow graph consisting of various kinds of nodes connected to each other by data dependencies. Each node in a DHDL program corresponds to one of the supported architectural templates listed in Table 1. DHDL is represented in-memory as a parameterized, hierarchical dataflow graph. DHDL is implemented as an embedded domain-specific language in Scala, thereby leveraging Scala's language features and type system.

3.1 Generating DHDL from Parallel Patterns

A fundamental design goal of DHDL is that it should be automatically generated from high-level languages which express the computation using parallel patterns such as *map*, *reduce*, *zip*, *groupBy* and *filter*. Previous work has shown that parallel patterns can be used to improve both productivity and performance

by using patterns as a basis for high level languages [1, 2] and sophisticated compiler frameworks [3]. The templates in DHDL are inspired from these well-known parallel patterns. This makes it possible to define explicit rules to generate DHDL for each parallel pattern mentioned above.

3.2 Language Constructs

A hardware datapath is described in DHDL using various nodes connected to each other by their data dependencies. DHDL also supports variable bit-width fixed-point and variable precision floating point types and associated type checking. Every node that either produces or stores data has an associated type.

	Template	Description	Parameters
Primitive Nodes	+, -, *, /, <, >, mux	Basic arithmetic, logic, and control operations	Vector Width, Type
	Ld, St	Load and store to and from on-chip memory	Vector Width, Bank Stride
Memories	OffChipArray	N-dimensional off-chip memory array	Size, Dimensions, Type
	BRAM	On-chip scratchpad memory	Size, Word Width, Double Buffering, Vector Width, # Banks, Interleaving Scheme, Type
	Priority Queue	Hardware sorting queue	Double Buffering, Size, Type
	Reg	Non-pipeline register	Double Buffering, Vector Width
Controllers	Counter	Counter chain used to provide loop iterator variables	Vector Width
	Pipe	Hardware pipeline of primitive operations. Typically used to represent bodies of innermost or perfectly nested loops	Parallelization Factor, Pattern
	Sequential	Sequential control flow that represents non-pipelined execution of multiple stages	Parallelization Factor, Pattern
	Parallel	Fork-join style parallel container with a synchronizing barrier at the end	
	CoarsePipe	Coarse-grained pipeline where multiple stages are executed as a pipeline using asynchronous handshaking signals	Parallelization Factor, Pattern
Memory Command Generators	TileLd	Load a tile of data from an off-chip array	Tile Size, Word Width
	TileSt	Store a tile of data to an off-chip array	Tile Size, Word Width

Table 1: Description of templates in DHDL and parameters for each template

Table 1 describes the hardware templates and associated parameters supported in DHDL. There are four types of nodes:

3.2.1 Primitive Nodes

Primitive nodes correspond to basic operations, such as arithmetic, boolean logic, and multiplexers. Some common complex multi-cycle operations such as abs, sqrt and log are also supported as primitive nodes.

Every primitive node represents a vector computation; a “vector width” parameter defines the number of parallel instances of each node. Scalar operations are thus special cases where the associated vector width is 1.

3.2.2 Memories

DHDL distinguishes between on-chip buffers and off-chip memory regions by representing them explicitly using separate nodes. This is used to capture on-chip and off-chip accesses which have different access times resource requirements. *OffChipArray* represents an N-dimensional region of memory stored on off-chip DRAM. *BRAM*, *Priority Queue* and *Reg* correspond to different types of on-chip buffers specialized for different kinds of computation. *OffChipArrays* are accessed using nodes called memory command generators, while on-chip buffers are accessed using primitive *Ld* (load) and *St* (store) nodes. The banking factor for a *BRAM* node is automatically calculated using the vector widths of all the *Ld* and *St* nodes accessing it such that the required memory bandwidth can be met.

3.2.3 Controllers

Several controller templates are supported in DHDL to capture imperfectly nested loops and parallelism at multiple nesting levels. Parallel patterns in input designs are represented using one of the *Pipe*, *CoarsePipe*, or *Sequential* controllers with an associated Counter node. Each of these controllers is associated with a parallelization factor and the parallel pattern from which it was generated, which is used in replicating the nodes for parallelization. For example, nodes associated with the *map* pattern are replicated and connected in parallel, whereas nodes associated with the *reduce* pattern are replicated and connected as a balanced tree. *Pipe* is a dataflow pipeline which consists of purely primitive nodes. This typically represents innermost bodies of parallel loops that are traditionally converted to pipelines using software pipelining techniques. *CoarsePipe* represents a coarse-grained pipeline where each of its stages are other controller nodes. *CoarsePipe* orchestrates the execution of its stages in a pipelined fashion using asynchronous handshaking signals, thereby being able to tolerate variations in the execution times of each stage. Communication buffers used in between stages are converted to double buffers. *Sequential* represents unpipelined execution of a chain of controller nodes. *Parallel* is a container to execute multiple controller nodes in parallel with an implicit barrier at the end of execution. *Counter* is a simple chain of counters required to generate loop iterators. *Counter* has an associated vector width so that multiple successive iterators can be produced in parallel. This vector width is typically equal to the parallelization factor of the *Pipe*, *CoarsePipe*, or *Sequential* it is associated with.

3.2.4 Memory Command Generators

OffChipArrays in DHDL are accessed at the granularity of tiles, where tile is an regular N-dimensional region of memory. Previous work [4, 5] has shown the importance of tiling transformations to maximize locality and generate efficient hardware. Accesses to *OffChipArrays* are explicitly captured in DHDL using special *TileLd* (tile load) and *TileSt* (tile store) controllers. Each *TileLd* and *TileSt* node instantiates data and command queues to interface with the memory controller, and contains control logic to generate memory commands.

3.3 Transformations

The DHDL compiler implements basic optimizations such as constant propagation, constant folding and bitwidth optimizations as graph rewrite rules during IR construction. Two transformations are defined to expand a DHDL graph for a given set of parallelism factors: *Replicate* creates multiple copies of a subgraph and connects all replicated copies in a *Parallel* container such that all copies run in parallel. *TreeReduce* replicates a control node and connects the copies as a balanced reduction tree. These transformations are required to capture parallelism in outer loop levels. Since each primitive node is implicitly parallel, parallelism factors for *Pipes* are contained within the vector widths of its primitive nodes. These transformations are run before the graph is mapped onto the CGRA hardware.

3.4 Application Example

Figure 1 shows the GDA kernel written in DHDL complete with off-chip memory transfers while Figure 2 shows the corresponding hardware diagram. Note that the design captures nested parallelism with two levels of *CoarsePipes* with stages separated by double buffers. Some parameters, like number of banks for

BRAM, are omitted as they are automatically inferred based on parallelization factors. The design is parameterized using three kinds of parameters: parallelism factors controlling number of parallel iterations, tile sizes corresponding to on-chip buffer sizes, and *CoarsePipe* toggles which controls whether an outer loop should be implemented as a *Sequential* or a *CoarsePipe*. The *CoarsePipe* toggle parameters also control whether the buffers internal to the *CoarsePipe* should be double-buffered. Note that by supplying different parameters, different design points implementing GDA can be automatically generated from the same DHDL source code.

```
// Declare all off-chip arrays
val x = OffChipArray(type=Float, N, D)
val y = OffChipArray(Bool, N)
val mu0 = OffChipArray(Float, D)
val mu1 = OffChipArray(Float, D)
val sigma = OffChipArray(Float, D, D)
// Declare on-chip BRAMs - boolean param indicates if a BRAM is a double buffer
val xtile = BRAM(type=Float, size=B*D, dblbuf=true)
val ytile = BRAM(Bool, B, true)
val subTile = BRAM(Float, D, true)
val mu0tile = BRAM(Float, D, false)
val multile = BRAM(Float, D, false)
val sigmaTile = BRAM(Float, D*D, false)

// Sequential S1:
// (1) Load mu0 and mu1, (2) Do GDA, (3) Store sigma back
Sequential(1) {
  // S1.1: Load mu0 and mu1 (offchip -> BRAM)
  Parallel (
    TileLd(mu0, mu0tile, D),
    TileLd(mu1, multile, D)
  )
  // S1.2: Outer CoarsePipe C1:
  CoarsePipe(N, B) { i =>
    // C1.1: Load xtile and ytile (offchip -> BRAM)
    Parallel (
      TileLd(x, i, xtile, B, D),
      TileLd(y, i, ytile, B)
    )
    // C1.2; Inner CoarsePipe C2 - Do GDA
    CoarsePipe(B) { j =>
      // C2.1: Subtract loop
      Pipe(D) { k =>
        subTile[k] = Mux(ytile[j])(xtile[j,k] - mu0[k], xtile[j,k] - mu1[k])
      }
      // C2.2: OuterProd loop
      Pipe(D,D) { (k1, k2) =>
        sigma[k1,k2] += subTile[i] * subTile[j]
      }
    }
  }
  // S1.3: Store sigma back (BRAM -> offchip)
  TileSt(sigma, sigmaTile, D, D)
}
```

Figure 1: Full DHDL code example for GDA benchmark

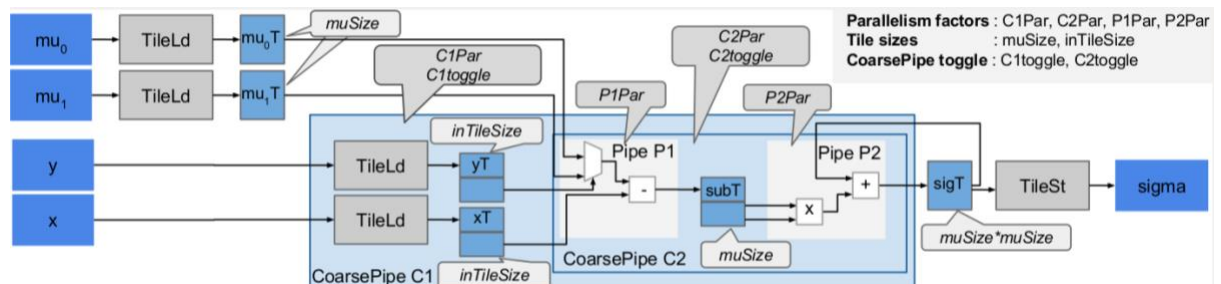


Figure 2: Corresponding parameterized GDA design from DHDL description in Figure 1. Bubbles denote parameters that apply to the template it points to. Note that some parameters apply to more than one template.

4. Applications

We use the following data analytics benchmarks to motivate the design of our architecture:

- 3D convolution convolves two three-dimensional arrays along each of their dimensions.
- Black-Scholes is a financial analytics application for option pricing.
- Gaussian discriminant analysis (GDA) is a classification algorithm which models the distribution of each class as a multivariate gaussian.
- Logistic regression is a binary discriminative classification algorithm that uses the sigmoid function for predictions
- k -Means clustering partitions a set of input points into k groups by iteratively calculating the k centroids which best represent the data
- k -Nearest neighbors classifies a set of test points using a labeled set of output points. Each test point is classified as the most commonly occurring label among the k closest labeled points.
- Sobel edge detection and other two dimensional image convolutions are common operations in image processing. Due to high degrees of data locality and access regularity, they are often accelerated with specialized hardware.
- TPC-H Query 6 is a data querying benchmark which reads a table of purchase records and filters out all records which match a given predicate. It then computes the sum of the product of two columns in the filtered records.

We also consider floating point matrix multiply, dot product, and matrix row summation, as these access patterns often occur in machine learning applications.

5. Proposed Architecture

The proposed architecture is similar to an island style FPGA. The main elements are compute blocks (CBs) that hold functional units and implement operations on the data or contain memory and provide data. Connection between compute blocks are done through wire segments which are connected together by switch boxes and connected to CBs with connection boxes. Both switch and connection boxes are built with statically configurable multiplexers.

The price of programmability is in area and energy consumed by switch and connection boxes, which in turn depend on the number of wire tracks in the architecture. The proposed architecture is organized hierarchically, with decreasing numbers of wires with higher levels of the hierarchy. This reduction expresses expected levels of locality in the applications and would follow Rent's rule.

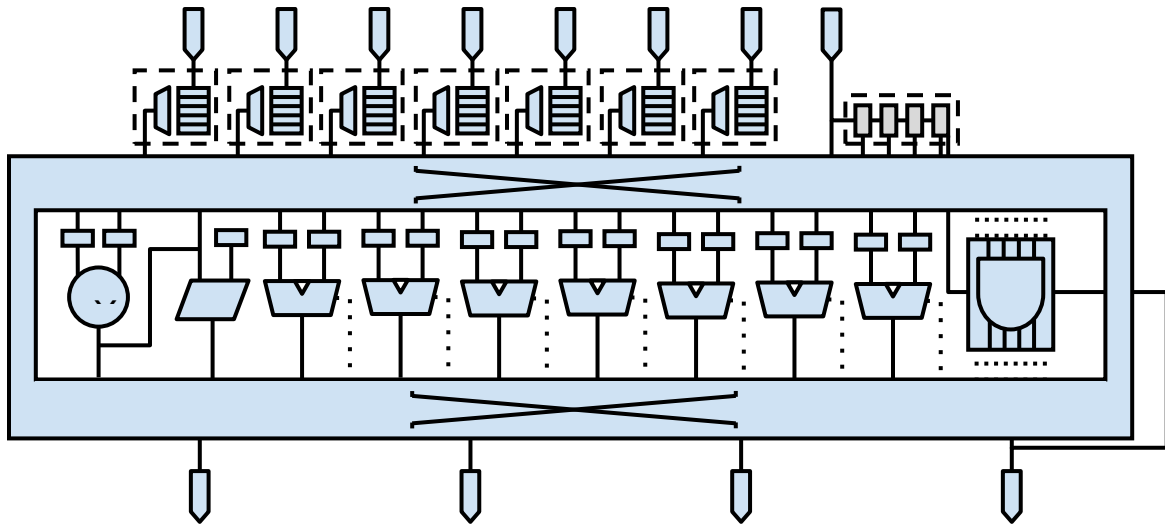


Figure 3: Compute block

A *compute block* (Figure 3) holds a number of *processing elements* (PEs), input/output ports, registers that hold values used in computation and FIFOs composed of pipeline flip flops. These resources are connected with depopulated x-bars internally, but they have to use ports to connect to other blocks. Number of elements, x-bar depopulation and design of each PE are all architecture parameters. Because the number of external connections is smaller than the number of connections required to run each PE independently and in parallel, we expect that PE utilization will not be 100% as we require the ability to perform a number of different primitive operations (add, compare, multiply, shift, logic, etc). VLSI implementation of these functions is quite different in amount of required area, but all of these functions are required in a programmable architecture. One of the options here is to make PEs homogeneous which would ease architectural mapping but each PE would implement all functions and the area would be large. Instead, we can use the fact that we expect some PEs to be unused in a compute block to hide inefficiency of heterogeneous blocks. For example, assume that I is the total number of input ports to the compute blocks and that $O=I/2$ is the number of outputs. If the total area of heterogeneous PEs is the same as the area of $I/2$ homogeneous blocks, the resulting area efficiency will be higher.

At the intermediate hierarchy level we have *processing cores* (Figure 4) which contain memory and compute block resources required to implement an application. Processing cores on the edge of the two dimensional array also include memory controllers for accessing off-chip memory. Compute resource are a large number (architecture parameter) of *compute blocks* connected between each other and with memory blocks in island style fashion. A compiler may map an application to multiple cores if there is not enough resources in a single core, or it can unroll the computation and put several copies into a single core to increase throughput and utilization.

The highest hierarchy level is composed of a two dimensional array of processing cores (Figure 5). These processing cores are connected by load and store units to on-chip scratchpad memories, as well as a small number of data busses connecting nearest neighbors. Adjacent horizontal strips of processing cores are also connected by scalar paths (shown as yellow wires) for implementing scalar reductions across multiple cores.

The interconnect topology between cores is another architecture parameter, but overall we expect two types of connections at this level: core to core and intra-core. Core to core facilitates data movement between different loops in the user program, while intra-core is only needed for better resource utilization - to allow a single function use resources of a neighboring core. The first type needs to connect potentially all cores together, but it can be done with time multiplexed bus because this is essentially the same as copying from one scratchpad memory to another within a single core. We enforce this memory copy pattern when two "loops" are mapped to different cores, because some kind buffering is required. If two functions

are mapped into the same core, a parent function can write directly into child's memory. The second type is similar to the connections within a core - it is statically assigned and goes through the switch boxes, but only connects the neighboring cores and the number of tracks is smaller than within the core.

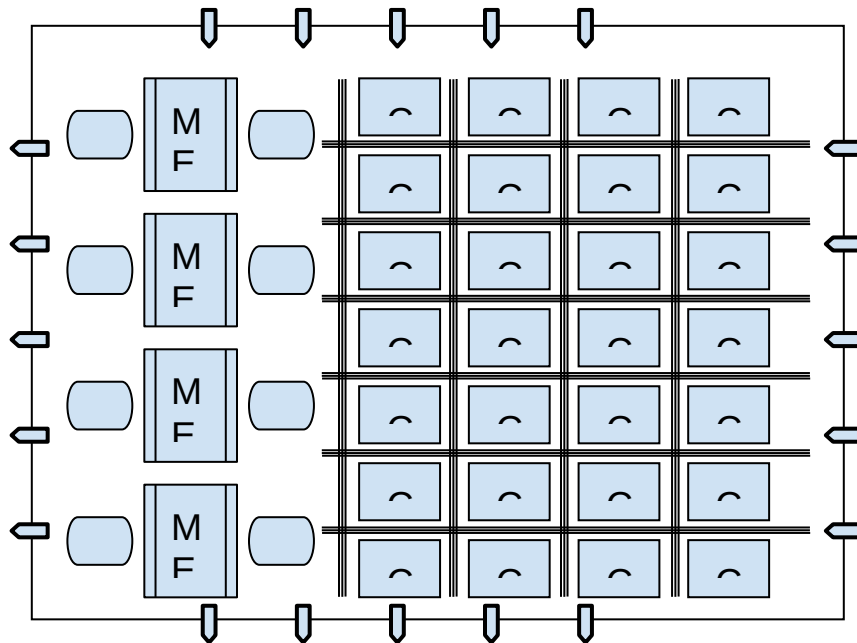


Figure 4: Processing core

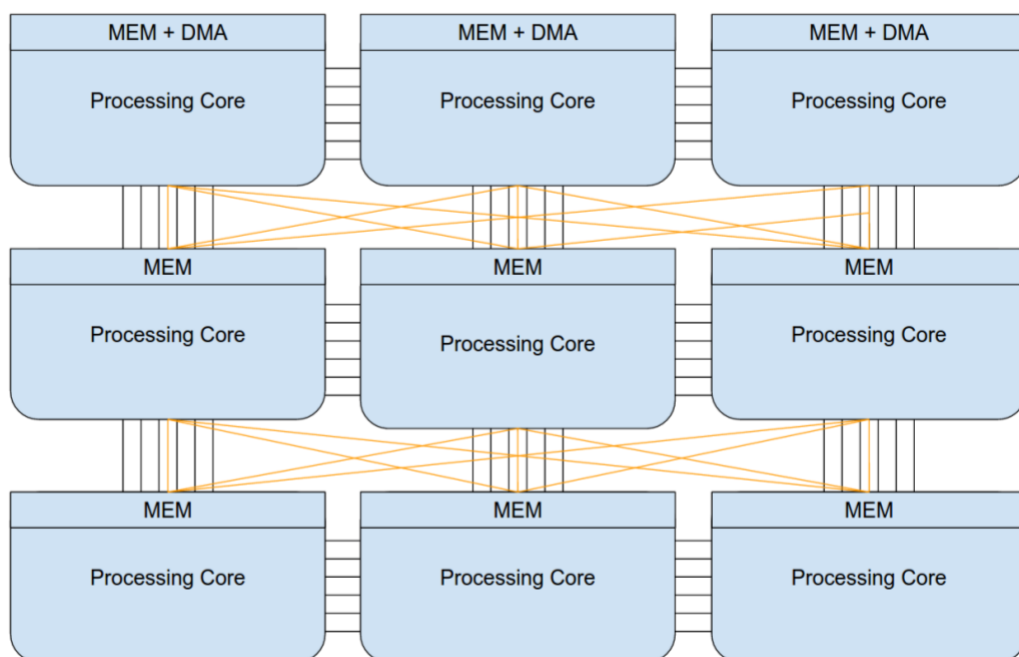


Figure 5: Coarse view of proposed architecture

6. Evaluation

circuit	PE	Counter	MemLd	MemSt	BRAM
DotProd_60	57	4	4	0	8
OutProd_sm	141	25	5	4	33
GDA_sm	13	7	4	1	7
MatMul_sm	13	7	3	1	5
TPCHQ6_sm	51	4	8	0	16
Kmeans_sm	46	9	2	1	5
DotProd_lg	58	10	4	0	32
OutProd_lg	485	137	5	4	257
GDA_lg	219	56	4	1	84
MatMul_lg	311	59	5	1	54
TPCHQ6_lg	629	27	12	0	288
Kmeans_lg	280	41	2	1	11

Table 2: Complexity of applications used for evaluation

We evaluated our proposed architecture on a subset of our motivating benchmarks:

- Vector dot product (dotprod)
- Vector outer product (outerprod)
- Gaussian Discriminant Analysis (GDA)
- Matrix multiplication (gemm)
- TPC-H Query 6 (tpchq6)
- K-means

Each application had two form: small (sm) that represents the minimal number of compute nodes and large (lg) which is unrolled version for the increased throughput. Table 2 shows the complexity of each application and gives statistics for different architectural blocks.

Our main goal is to evaluate the price of programmable wires and gain a sense of architecture performance. The main question we want to answer is: what's the area/power of connection and switch boxes? How much power do the wires take?

VPR can automatically size and evaluate the cost of wires and switch boxes and connection boxes between major blocks, but it assumes that connections inside hierarchical blocks are fully specified. For this reason, we use a simplified architecture for our experiments. The architecture used is flat (no hierarchy) with 5 types of elements: *BRAM*, *TileLd*, *TileSt*, *Count*, and *PE* with switch boxes and connection boxes between them (see Figure 6). Each PE has and ALU, MUX, MULT, 8x 4-to1 LUTs but only one of these modes can be active, in other word, each PE can do only one operation on 32bit data (but more than one 1bit logic operation can be packed into LUTs). Element are organized into repeating pattern of rows according to a formula: *TileLd*, *BRAM*, *TileSt*, $2 \times PE$, *Counter*, $14 \times PE$. We have chosen this initial mix of elements so that most of the area on the chip is dedicated to the compute (each BRAM is about 10x larger than a PE). The interconnect was implemented with heterogeneous wire segments of lengths 1, 4, 8, 16, and 32 with no depopulation in the connection boxes ($F_{in} = 1$ and $F_{out} = 1$) and Wilton pattern in the Switch boxes with $F_s = 3$. We have used fully populated connection boxes because current VPR version doesn't support bus-based routing. Bus-based routing would cause a reduction in the area and power of the

connection boxes (by some integer factor, up to 32x) but the number of tracks and hence switch boxes area would remain the same.

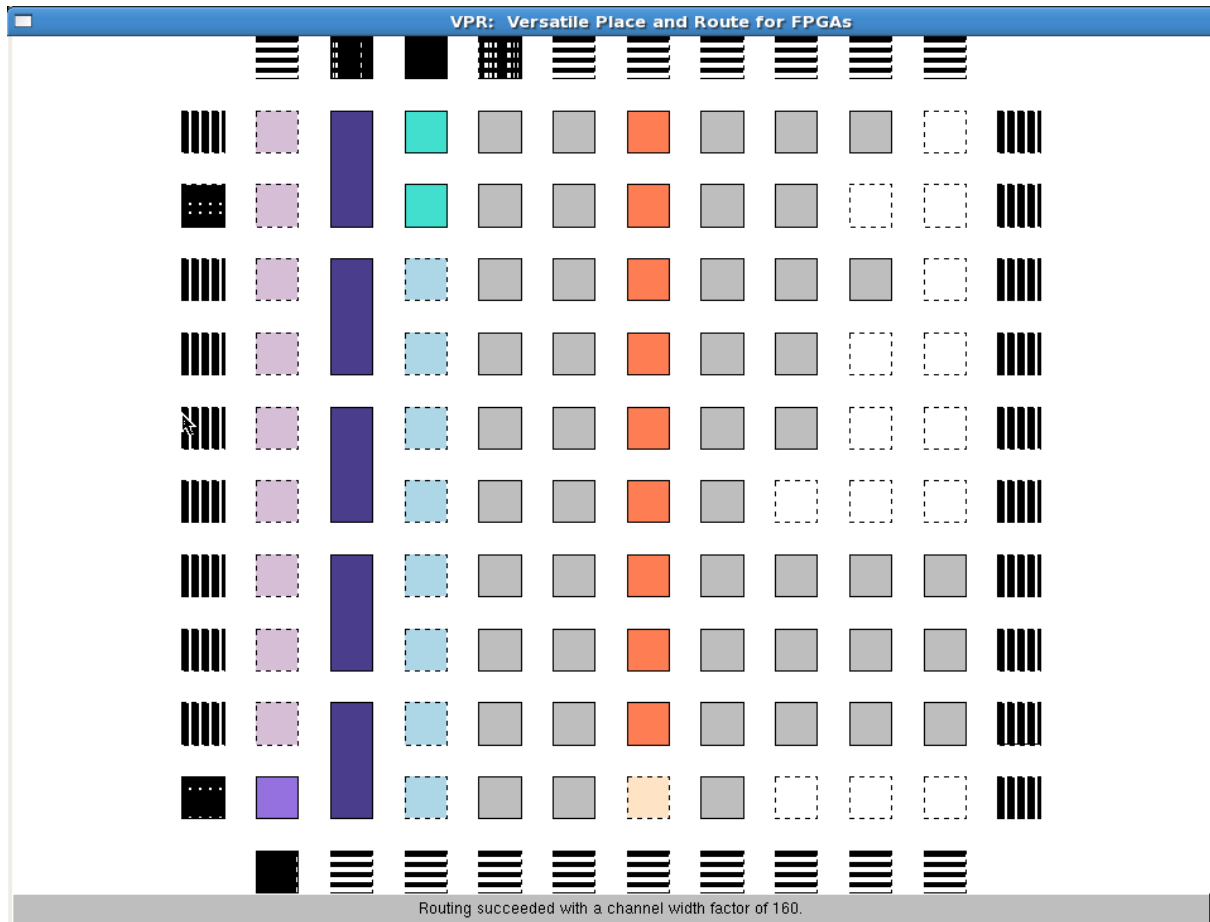


Figure 6: Sample architecture organization and VPR placement

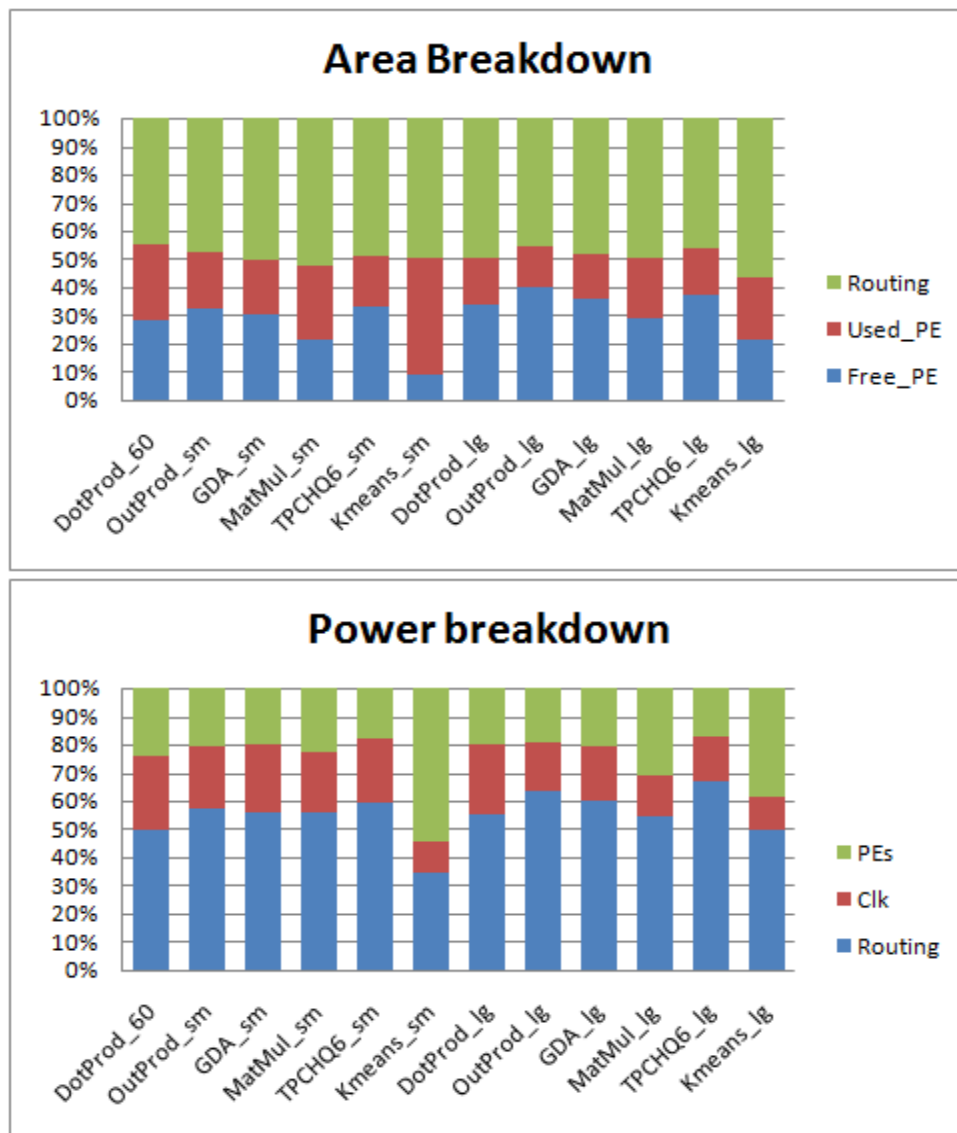
Our results are summarized in the Table 3. Figure 6 shows Area/Power breakdown and resource utilization. A few characteristics emerge from the results

- the minimum number of tracks is about the same across applications (~170)
- BRAM are over utilized (need to have more bandwidth from BRAM)
- Average wire length grows unproportionally: 13->31 for TPCH

circuit	Area, mm ²	Power, W	Fmax, MHz	Minimum tracks	Arg wire length	Area Eff, GOP/mm ²	Energy Eff, GOP/W
DotProd_60	2.07	0.111	310	116	14.2	0.819	15.29
OutProd_sm	9.86	0.400	216	140	17.0	0.228	5.63
GDA_sm	1.80	0.085	177	156	9.4	0.067	1.42
MatMul_sm	1.00	0.085	258	182	11.0	0.282	3.32
TPCHQ6_sm	4.67	0.203	294	162	12.9	0.158	3.64
Kmeans_sm	0.95	0.139	361	160	7.7	3.729	25.35
DotProd_lg	9.11	0.455	327	156	15.6	0.048	0.96
OutProd_lg	85.94	1.677	113	130	22.0	0.034	1.72

GDA_lg	26.61	1.052	182	142	20.8	0.073	1.85
MatMul_lg	16.43	0.843	167	164	15.0	0.438	8.54
TPCHQ6_lg	87.58	1.396	91	136	31.1	0.055	3.47
Kmeans_lg	7.13	0.688	217	222	12.3	2.351	24.35

Table 3: VPR results



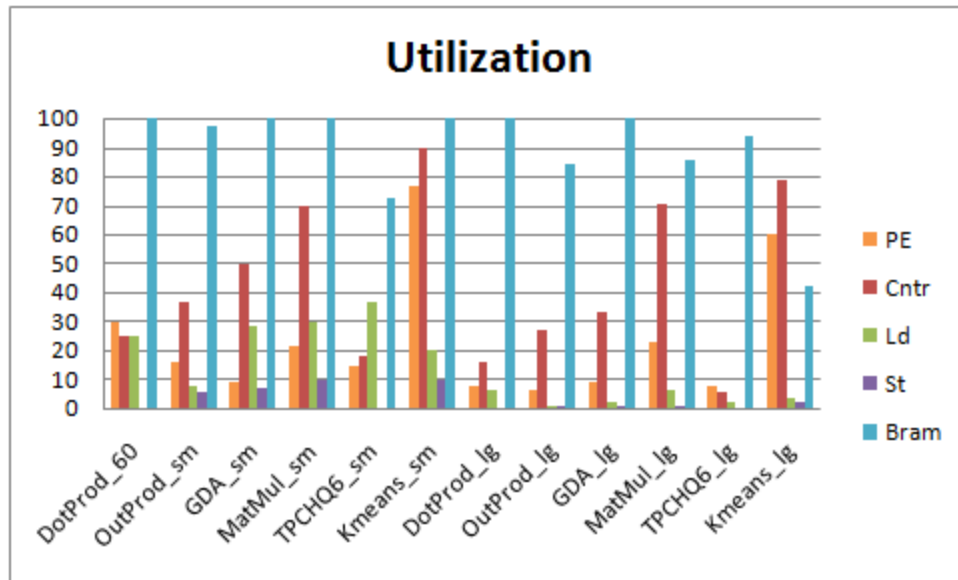


Figure 6: Area and power breakdown and resource utilization for benchmark applications.

As a consequence of the unproportional growth of average wire length we see a maximum frequency degradation as well, which we can interpret as a performance degradation of the routing (see Figure 7). Some of it could be fixed by constraining the design, however we treat it as an evidence in favor of hierarchical organization and a sign that the optimal core size should be around 10mm²

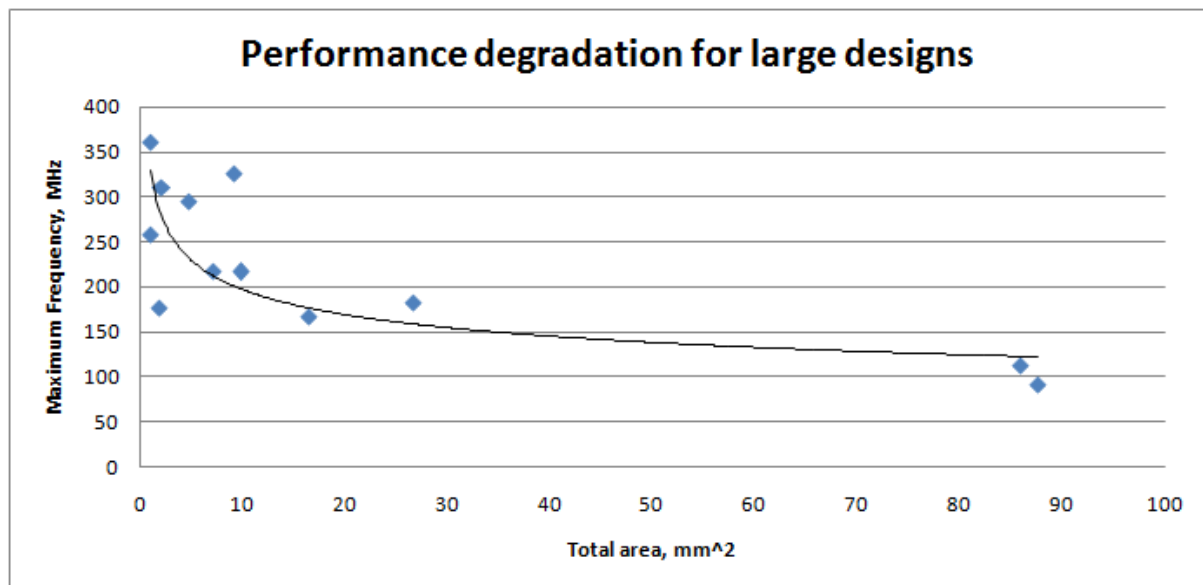


Figure 7: Routing performance degradation.

7. Conclusion and Future Work

Future work for architecture:

1. Implement each module in Verilog to get better area/power estimates
2. Bus-based routing to reduce Connection Box cost
3. Hierarchical design
4. Pipeline wires to increase maximum frequency (like done in latest Altera chips:
https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01220-hyperflex-architecture-fpga-socs.pdf)

5. Increase BRAM bandwidth
6. Find optimal mix of elements
7. Create “fracturable” elements so that 32bit ALU can do two 16bit or four 8bit operations
8. Add correct register retiming
9. Explore RTR (run time reconfiguration) and dynamic content(like in SPR and PipeRench)

References

- [1] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*.
- [2] Arvind K. Sujeeth, Hyoukjoong Lee, Kevin J. Brown, Hassan Chafi, Michael Wu, Anand R. Atreya, Kunle Olukotun, Tiark Rompf, and Martin Odersky. OptiML: An implicitly parallel domain specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning, ser. ICML, 2011*.
- [3] Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for 13 performance-oriented embedded domain-specific languages. In *TECS '14: ACM Transactions on Embedded Computing Systems*, July 2014.
- [4] Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '13*.
- [5] Raghu Prabhakar, David Koeplinger, Kevin Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. Generating configurable hardware from parallel patterns. 2015. URL <http://arxiv.org/abs/1511.06968>.
- [6] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins: ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In *Field Programmable Logic and Application, FPL '03*.
- [7] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. DaDianNao: A machine-learning supercomputer. In *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '14*.
- [8] Daofu Liu, Tianshi Chen, Shaoli Chen, Jinhong Zhou, Shengyuan Zhou, Oliver Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. PuDianNao: A polyvalent machine learning accelerator. *ASPLOS '15*.
- [9] Clément Farabet, Berin Martini, Polina Akselrod, Selçuk Talay, Yann LeCun, and Eugenio Culurciello. Hardware accelerated convolutional neural networks for synthetic vision systems. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 257-260. IEEE, 2010.
- [10] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *23rd International Symposium on Field-Programmable Gate Arrays, FPGA '15*.
- [11] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon

- Kim, and Hadi Esmaeilzadeh. TABLA: A Unified Template-based Framework for Accelerating Statistical Machine Learning. 2015. URL <https://smartech.gatech.edu/handle/1853/54043>
- [12] H.M. Hussain, K. Benkrid, H. Seker, and A.T. Erdogan. FPGA implementation of k-means algorithm for bioinformatics application: An accelerated approach to clustering microarray data. In *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pp. 248–255, June 2011.
 - [13] Ian Kuon and Jonathan Rose. "Measuring the gap between FPGAs and ASICs." In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 203-215. CAD '07.
 - [14] A. T. H. Schmit, D. Whelihan, PipeRench: A virtualized programmable datapath in 0.18 micron technology. In *Proceedings of the Custom Integrated Circuits Conference*, pp. 6366, 2002.
 - [15] S.M.A.H. Jafri, T. N. Gia, S. Dytckov, M. Daneshtalab, A. Hemani, J. Plosila, and H. Tenhunen. NeuroCGRA: A CGRA with support for neural networks. In *International Conference on High Performance Computing & Simulation*, pp. 506-511. HPCS '14.
 - [16] Gerard JM Smit, Paul M. Heysters, and Bert Molenkamp. The Chameleon project in retrospective. In *5th PROGRESS Symposium on Embedded Systems*. pp. 181–184, 2004
 - [17] Takashi Miyamori and Kunle Olukotun. REMARC: Reconfigurable multimedia array coprocessor. In *IEICE Transactions on information and systems* 82, no. 2: 389-397. 1999.
 - [18] Carl Ebeling, Darren C. Cronquist, Paul Franklin, and Chris Fisher. RaPiD—a configurable computing architecture for compute-intensive applications. *University of Washington Department of Computer Science & Engineering Tech Report TR-96-11-03*. 1996.
 - [19] Zijian, Hou, Chen Xin, and He Weifeng. Improved pipeline data flow for DySER-based platform. *16th International Symposium on Quality Electronic Design (ISQED)*, 2015. IEEE, 2015.
 - [20] J.R. Hauser and J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. In *The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 1997. *Proceedings*. pp.12-21, 16-18. April, 1997.