MULTI-KERNEL FLOORPLANNING FOR ENHANCED CGRAS

Aaron Wood, Adam Knight*, Benjamin Ylvisaker†, Scott Hauck**

*Dept. of Electrical Engineering and †Dept. of Computer Science and Engineering
University of Washington, Seattle, WA 98195
*{arw82, amknight, hauck}@ee.washington.edu †ben8@cs.washington.edu

ABSTRACT

Signal processing applications have been shown to map well to time multiplexed coarse grained reconfigurable array (CGRA) devices, and can often be decomposed into a set of communicating kernels. This decomposition can facilitate application development and reuse but has significant consequences for tools targeting these devices in terms of allocation and arrangement of resources. This paper presents a CGRA floorplanner to optimize the division and placement of resources for multi-kernel applications. The task is divided into two phases aligned with the respective goals. Resource allocation is accomplished through incremental assignment to minimize performance bottlenecks while operating within the bounds of the maximum available resources. The resulting allocation of resources is arranged in the device using simulated annealing and a perimeter-based cost function which serves to minimize resources needed for both inter- and intra-kernel communications. The floorplanner is applied to a set of multi-kernel benchmarks demonstrating resource allocations providing maximum throughput across a range of available resources. The algorithms are very fast, taking only a few seconds while producing high quality results. Inter-kernel wire lengths are almost always minimal, and the resource allocation is proven optimal.

1. INTRODUCTION

CGRA devices represent an emerging family of architectures exploring alternatives to commodity microprocessor computing. There is a large body of research converging on these styles of architectures from a variety of perspectives from FPGAs to VLIWs. These devices have been shown to be well suited to applications oriented toward streaming computation. Arguably the most important driver of architecture development and adoption is tool support for efficiently mapping applications to these new architectures. Supporting this notion, we present an automated floorplanning tool as part of a tool chain to leverage the advantages of CGRAs for multi-kernel applications.

This work was supported by NSF grant #CCF-1116248 and by Department of Energy grant #DE-FG02-08ER64676.

Floorplanning is a challenging problem for traditional ASIC and FPGA netlists. While the size of the problem for a CGRA is mitigated somewhat by the coarse granularity of the functional units, it is further distinguished from traditional floorplanning in two main ways. First of all, floorplanning normally deals with a fixed quantity of resources. Once a netlist is mapped and packed to a target technology, the quantity and type of resources required in the physical device is essentially fixed. In the CGRA case, the number of resources allocated is more flexible by trading physical resources for time through time multiplexing. Previous work on the Mosaic project [1] considered a single kernel of computation targeting the CGRA device. In this paper we examine multiple communicating kernels sharing the same physical device, made possible by enhancements to the Mosaic compiler [2].

The multi-kernel floorplanning problem for CGRAs is complicated by the flexibility of mapping each kernel to the device when multiple kernels are involved. The number of resources allocated to each kernel can be adjusted with a corresponding impact on the performance of the kernel. Thus, we must decide how to divide the available resources among them. The second issue is related to the shape of kernel regions. Rectangular regions, while convenient for ASIC or FPGA floorplans, are not well suited to the CGRA floorplanning problem. For example, a five cluster kernel can either be restricted to only 1x5 layouts, which radically increases wire lengths within a kernel, or allowed to take a 2x3 shape, wasting 17% of the allocated resources. The allocation of resources in general should make use of as many resources as available to maximize performance. However, a given allocation of resources is not guaranteed to fit on the device in a rectangular region for each kernel. We provide an alternative solution that allows irregularly shaped regions in the resulting floorplan.

2. BACKGROUND

The Mosaic project [1] is developing an infrastructure to explore CGRA architectures and CAD tools. These CGRAs are built from clusters of functional units and memories with a configuration plane to enable cycle to cycle static scheduling of operations on these resources [3]. In order to enable executing multiple independent kernels, an enhanced CGRA allows configurable subsets of the

resources to operate as independent CGRA regions within the architecture to allow kernels with different performance characteristics to reside on the same device. This means that within a CGRA region, operations and interconnect are scheduled and have a fixed execution sequence. However, different CGRA regions are able to operate independently in the fabric. This allows individual CGRA regions to be tailored exclusively to a particular kernel of computation instead of trying to shoehorn an entire application into a single monolithic kernel spread across the entire device. Between CGRA regions, the application uses massively parallel processor array (MPPA) style flow controlled buffered interconnect, effectively decoupling the control domains of individual CGRA regions.

Mosaic uses pipelined interconnect in a fixed frequency device. This eliminates adjusting the clock speed as a technique to address communication rate mismatches between kernels. The advantage is that in a practical implementation, only a single clock network and PLL are required. Additional hardware is also not required to synchronize at arbitrary clock boundaries between clusters simplifying the device architecture.

There are two important properties of individual kernels that the floorplanner uses. The first is the size of the kernel, measured by the total number of operations that must be executed. The second is the recurrence initiation interval. Initiation interval (II) is the number of cycles between starting subsequent iterations of a loop. The recurrence II is the length of the shortest loop carried dependence cycle in the dataflow graph. This represents the minimum II achievable for the kernel and therefore the maximum throughput given sufficient resources.

The hybrid architecture has significant benefits when compared to existing approaches. In CGRAs, all computations must operate in lockstep, slowing the entire system to the rate of the slowest element; the hybrid architecture allows individual kernels to operate at their own rate, often achieving significantly higher throughput. In MPPAs, such as Ambric [10], users have to write code for each individual processor and must refactor the design manually to use more resources; the hybrid architecture can automatically spread a given computation across multiple compute units, allowing the user to express a computation in its most natural decomposition while relying on the tools to automatically harness multiple compute resources for individual kernels to provide the best overall throughput.

Supporting the multi-kernel flow in the Mosaic project requires integration of the new floorplanner into the existing toolchain. Figure 1 shows where the floorplanner is inserted into the Mosaic tool chain in order to manage multiple kernels in the overall application. Design entry begins with Macah [4] which already supports defining multiple kernel blocks [2]. The compiler generates a dataflow graph for each kernel. The compiled dataflow graphs are consumed by SPR [5]. This tool is inspired by

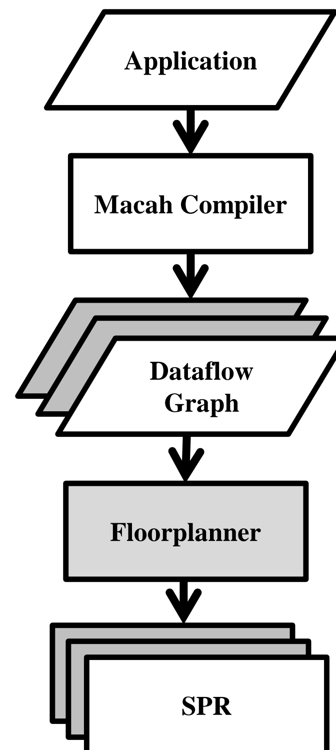

Figure 1. Multi-kernel Mosaic tool chain.

VLIW compilation for scheduling as well as FPGA tools for assigning operations to the physical resources as part of placement and routing. SPR targets an individual kernel for CGRA style execution.

In order to support the new multi-kernel model, it is necessary to find an allocation of resources to maximize throughput of the overall computation while respecting the finite amount of total resources available. Once this division of resources is decided, a global placement works to minimize resources dedicated to the communication links between kernels. Once each kernel is assigned a region of the device, the existing toolset can be applied to map each kernel onto the subset of resources allocated to it. Therefore, the floorplanner is situated between Macah and SPR to provide the resource partitioning and global placement. Resources in Mosaic CGRA architectures are grouped into clusters of multiple ALUs, memories, stream ports and other resource types on a square grid. For the purposes of this floorplanning task, these clusters are the granularity at which the resource allocation and placement are performed.

The Mosaic hardware supports multiple kernels by mapping them to different regions of the chip. Each kernel operates like a CGRA array, with a fixed modulo-scheduled operation, deep pipelines, and time-multiplexed logic and routing resources. This provides very cheap and effective parallelism for streaming computations. Signals between kernels operate with handshaking, moving data independent of the IIs or stalls of the intervening kernels [12]. As such,

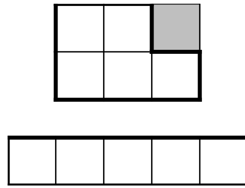

Figure 2. Five cluster kernel floorplans.

the inter-kernel wires are more expensive than the intra-kernel wires, and thus the length of communication wires between kernels must be carefully controlled.

3. RELATED WORK

Floorplanning is an important part of ASIC and FPGA design flows. In the FPGA space, Xilinx PlanAhead [6] allows the designer a high degree of control over where specific modules or components are placed in the architecture, which in many cases can mean the difference between a hopelessly long placement phase and a design that meets the required timing. While floorplan regions are constrained to rectangular regions, they can be composed together to provide irregularly shaped region. However, this is entirely performed manually by the designer.

There is a large body of additional floorplanning work to consider as well. In [7] the authors target heterogeneous FPGA architectures using a slicing technique with compaction. While well suited for FPGAs, this technique does not map well to the coarse granularity of CGRAs. For example, Figure 2 shows a five cluster kernel mentioned previously which suffers from wasted resources or poor wire length in the top and bottom arrangement respectively.

In [8], the authors use a hierarchical clustering approach. This again leaves unused resources in the array based on the communication pattern of the macro based netlists targeted. In a coarse grained device like a CGRA, this leads to poor utilization where it is much more costly than in an FPGA.

The StreamIt language and compiler [9] is a related project that offers a similar model of computation with actors that communicate through FIFO channels. However, the approach is quite different because the underlying hardware is an array of processors which are able to change tasks to a much higher degree than our enhanced CGRA model, which is limited to its static schedule of instructions. With StreamIt, kernel code may be swapped in and out of a particular core, where in our case each computational element is a member of a kernel region and has a small number of operations which must operate in lock step with other members of the same region for the lifetime of the application.

The Ambric [10] flow controlled interconnect channels are similar to the inter-kernel communication resources of the enhanced CGRA. However, they do not support a scheduled execution mode, making it less amenable to operating in a CGRA mode for individual kernels spread across a collection of resources. At its debut, Ambric's programming model required development of individual programs to execute on the processors in the array. This meant that a developer would need to figure out how to break down an application into components each suitable for implementation on a single Ambric processor and then write a small program for each one. Even for applications where one program might be reused on many processors, handling distribution of data to and from each processor would still need to be managed manually. Tool support for leveraging the array without the need to decompose an application by hand is a key feature of the Mosaic project.

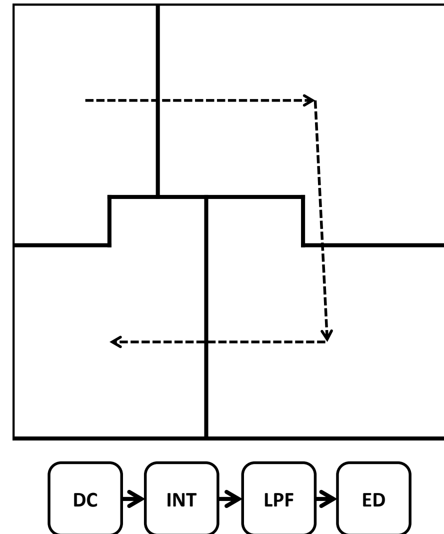

Figure 3. Digital camera pipeline (IPL) kernel communication.

Table 1. Digital camera pipeline kernel properties

Kernel	Recurrence II	Stateless Ops	Stateful Ops
DC	8	160	33
INT	19	325	126
LPF	4	209	134
ED	15	177	90

4. FLOORPLANNING ALGORITHM

Floorplanning must both determine the number of resources assigned to each kernel to achieve the best throughput, and place those resources onto the device to minimize the communication costs in the system. These two questions naturally break the floorplanning problem into two phases:

- Resource Allocation: the goal is to optimally assign a finite quantity of available resources amongst the various kernels to maximize the throughput of the overall application. It is given per-kernel information (Table 1) including the

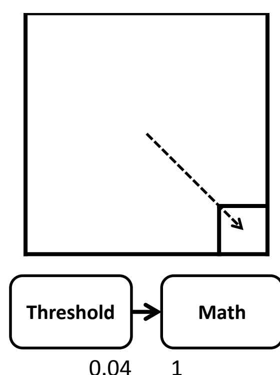

Figure 4. PET kernels and floorplan.

number of operations performed and the recurrence II (a limit on the maximum throughput). Each inter-kernel signal is also annotated by the number of data items per iteration sent and received by the source and destination of the signal respectively. From this data it computes an assignment of resources to each kernel, and passes this on to the Kernel Placement step.

- Kernel Placement: the goal is to place the resources assigned in Resource Allocation, seeking to minimize the resulting routing costs. Resources dedicated to a given kernel should be contiguous and as compact as possible to limit the length of intra-kernel routing. To minimize the more expensive inter-kernel signal lengths, kernels that communicate with each other should be placed close together. An example result is shown in Figure 3.

4.1. Resource Allocation

Intuitively, each kernel wants as many resources as possible to reduce the schedule depth and increase its throughput. However, this must be balanced in the context of the overall application. With finite resources available in the device, the topology of connections between kernels and the performance of neighboring kernels, maximizing performance of individual kernels will not necessarily produce an optimal solution. For example, consider the positron emission tomography (PET) event detector application (Figure 4 bottom). It has two kernels, and a simple allocation would give each kernel half of the available resources. However, the application actually consists of a line-rate threshold kernel that must quickly process data, looking for a relatively rare event, and a math kernel that does complex processing on those events. In the example, the send and receive rates are 0.04 and 1 tokens per initiation for the threshold and math kernels respectively measure in data tokens per II. The best allocation (Figure 4 top) actually dedicates almost all of the chip resources to the threshold kernel, since that boosts overall throughput, while starving the math kernel for resources ends up not affecting overall throughput at all.

```
 1 Assign all kernels minimum legal resources
 2 while (true) {
 3    store current solution
 4    for each kernel
 5       calculate resII from ops and resources
 6    for each port
 7       calculate rate with resII and port rate
 8    for each stream
 9       set stream rate to "slower" rate
10    for (total number of ports) {
11       for each port
12          if port rate > stream rate
13             change port rate to stream rate
14             propagate to kernel's other ports
15    }
16    add resources to all limiting kernels
17    check termination conditions
18 }
```

Figure 5. Resource Allocation Algorithm

While the best allocation of resources for the PET application is relatively obvious, for a more complex network of kernels it is much more difficult. Each kernel with its own resource requirements, recurrence II and stream rates, ultimately interacts with all of the kernels in the context of the total resource limit of the device itself.

The resource allocation portion of the algorithm is outlined in Figure 5. At a high level, the algorithm begins with a minimal resource allocation to each kernel. The main loop performs an analysis of the application in the context of the current resource allocation, adds resources to kernels that limit throughput, and then iterates until the device is filled or no further performance gain is possible due to limits in the kernels themselves.

The algorithm starts with a graph describing the communication between the various kernels from the Macah compiler, as well as information about each kernel from SPR, the tool that performs scheduling, placement and routing for an individual kernel in Mosaic. As an example, we will use the digital camera pipeline (IPL) application in Figure 3, with kernel parameters shown in Table 1. The recurrence II is the lower limit on II if the kernel is not resource limited. The number of operations for a kernel indicates the size of the dataflow graph representing it. All of these operations must ultimately have an issue slot available in the device. For a given number of resources, the resulting II can be calculated, or for a given II, the number of resources can be calculated. In one extreme, provided sufficient instruction memory, all operations could be executed on one functional unit. The other more desirable extreme spreads operations out among a collection of functional units to take advantage of data and pipeline parallelism in the application. Lastly, a production or consumption rate for each output or input port is provided by the developer to indicate how often a value is produced or consumed on a per iteration basis. This information allows the algorithm to assess bottlenecks in the communication between kernels operating at their own

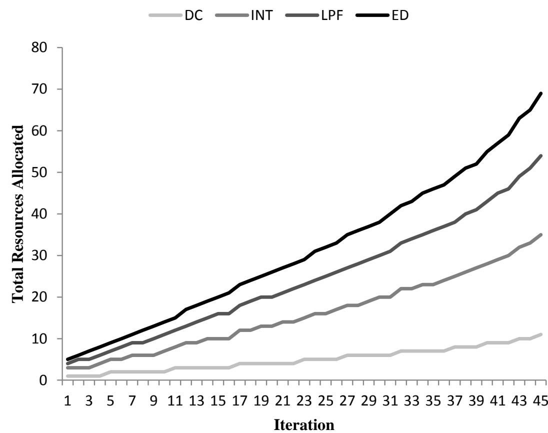

Figure 6. IPL resource allocation progression

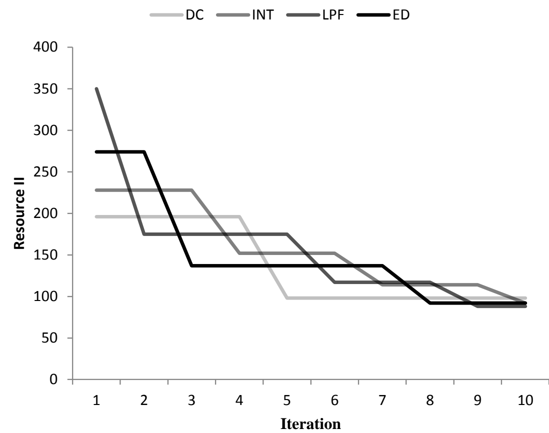

Figure 7. IPL II progression over 10 iterations.

IIs in order to increase throughput as much as possible. This information may ultimately be obtained from automated simulation of the application prior to mapping, but is currently manually annotated.

From the data provided, the algorithm starts by allocating each kernel a minimum number of resources. This is limited by the maximum II supported by the hardware, as well as providing a sufficient quantity of resource types such as memories or stream ports. This first solution not only represents the minimum number of resources absolutely necessary to execute the application, but will also be the slowest solution in terms of performance because operations must each have an issue slot in the schedule. Note that we assume that memory operations on different arrays can be packed into the same physical memory. For simplicity of explanation, we will assume the target device supports an unlimited maximum II so for the IPL application, the initial allocation of resources to each kernel is just one resource each.

From the initial resource allocation, the resource limited II is calculated on lines 4-5 of Figure 5. This is calculated as the number of operations divided by the number of resources allocated rounded up, and is the minimum schedule depth needed to provide every operation a resource and time slot to execute. On the first iteration, this will be equal to the total number of operations that need to execute for each kernel, since only one resource is available to each kernel. With each kernel's II, the next stage calculates the absolute rates at which values would be produced or consumed at each port (lines 6-7) of each kernel assuming input streams always have data available and output streams are never full. Thus, if a kernel has an II of 4, and produces 2 values per iteration on a port, the port would produce at an absolute rate of 0.5 running unconstrained. For IPL, all kernels produce and consume data at the same rate, so their port rates are the reciprocal of their respective resource constrained IIs.

Now the algorithm evaluates each stream in the application and assigns the stream rate to the value of the "slower" end (lines 8-9). For example, if one end of the stream is trying to produce results every cycle, and the other end can only consume once every 5 cycles, the stream rate will be 0.2 data elements per cycle. While this local processing puts an upper limit on the rates of each channel and kernel, we have to also model the more global behavior. The faster end of the stream will slow down to match the stream rate through stalling, and transitively this will slow the other ports of this kernel. Other kernels may then be slowed, until a steady-state is reached. For the IPL example on the first iteration, streams connected to the INT kernel will be set to the INT port rates because it is the slowest kernel on this iteration.

With the kernels, ports and streams annotated with local information, the next phase of the algorithm begins (lines 10-15). For every kernel, the ports are evaluated by comparing the port rate to the stream rate the port is connected to. If the port rate does not match the stream rate, this means that the port wants to operate faster than its partner on the other end of the stream, but this is not allowed since the slower port dictates the maximum rate. When this condition is detected, the port rate is changed to the stream rate to match the other end. The change in port rate is then propagated to the kernel II itself (which will no longer be resource limited) and to all other ports of that kernel. The process of evaluating all ports of the system continues until no further changes are made to any port rates after all kernels have been evaluated. Again on the first iteration for IPL, the progression will be the port rates of the INT kernel propagate to the DC and LPF kernels and the LPF rates propagate to the ED kernel making so that at the end of the iteration, all four kernels are operating at the INT kernel rate.

Once the algorithm has arrived at the rates for all kernels and ports in the design, the kernels have their slack calculated. Slack is the difference between the originally calculated ideal rate and the rate the kernel was assigned during execution of the algorithm. Zero slack means that the kernel is operating as fast as possible given the

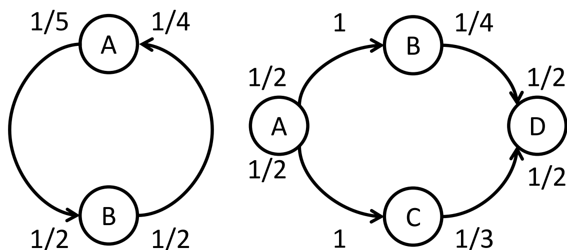

Figure 8. Unbalanced loop and re-convergence.

resources allocated to the kernel. In the IPL case this will be the INT kernel. The kernel is therefore limiting performance of the overall application. There may be more than one limiting kernel if multiple kernels have zero slack. The limiting kernel or kernels are then provided with the minimum increment of resources required to reduce their II from the present value, and thereby increase throughput (line 16). With the resource allocation changed, the process begins again.

Termination conditions for the algorithm are as follows. If at least one of the limiting kernels is already operating at its recurrence II, the performance cannot be improved because this value is a lower bound on the schedule depth for the kernel, so additional resources will not improve it further. At this point the algorithm returns to the last solution. Alternatively, if the limiting kernel(s) can benefit from more resources but the sum of all allocated resources would exceed the device capacity then the algorithm will also return to the last solution. Figure 6 and Figure 7 show the incremental solutions for the IPL application in terms of resources and kernel II respectively as generated by the Resource Allocation algorithm. Note that we assume the device supports a maximum II sufficient for every kernel to execute on a single resource.

It is also possible that the system has no legal solution. Our model of computation for floorplanning currently only allows blocking reads. If the production and consumption rates around a loop or where the flow of data diverges and then re-converges are unbalanced, then somewhere in the system a buffer associated with a stream will either become full or empty such that the execution will deadlock. Figure 8 shows an example of each condition with the port rates labeled. We detect this condition by limiting the number of iterations of the inner loop of the resource division phase to the number of ports in the design. Intuitively, if the algorithm is propagating a change due to a particular port more than once, then there is an unbalanced loop and the algorithm terminates. These conditions ultimately mean there is no steady state behavior of the system with bounded buffers between kernels.

Our algorithm provides the best possible application throughput for a given device capacity and the supported production and consumption model. A proof of optimality is presented in appendix at the end of the paper. To summarize the proof, this algorithm progresses through the set of Pareto optimal solutions from the smallest and slowest to the fastest and largest terminating when no further improvement is possible or the available resources are exhausted.

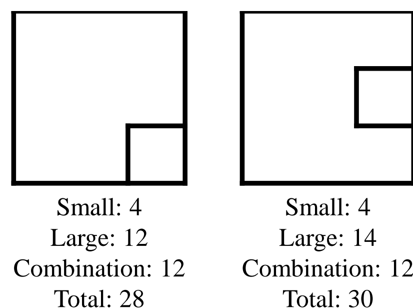

Figure 9. Perimeter calculation examples.

The resource allocation phase is very fast even for the most complex multi-kernel benchmarks such as the 18 kernel discrete wavelet transform (Wavelet) application which completes this phase in no more than 2 seconds. Even if the approach scales poorly with the number of kernels, its execution time is dwarfed by the Macah compiler and SPR.

4.2. Global Placement

The global placement phase takes the quantity of resources assigned to each kernel in the resource allocation phase and uses simulated annealing to place these resources in the device. The cost function works to keep the resources for each kernel together while also placing communicating kernels close together to reduce resource utilization and maintain routability.

After the division of resources has been established, the algorithm moves on to the coarse placement of kernels on the device in order to minimize routing resources dedicated to communication between kernels. The global placement is a simulated annealing based placement algorithm with a specialized cost function geared toward the foorplanning problem. Each resource assigned to a kernel is a separate moveable object. Moves are made by selecting two locations at random and swapping the resources assigned to these locations. Swaps of resources in two locations from the same kernel are useless and are not allowed. The objective of the placement is to minimize the distance between resources that will communicate, and can be broken down into minimizing intra-kernel communication, and minimizing inter-kernel communication.

There are two cost measurements used, which are applied to both internal and external kernel communication. The first is perimeter. Perimeter is evaluated by visiting each resource associated with a kernel and checking its neighbors. An adjacent position that is not another member of the same kernel counts as one unit of perimeter. Placements with a lower perimeter translates to a tightly packed cohesive block, while a large perimeter cost means

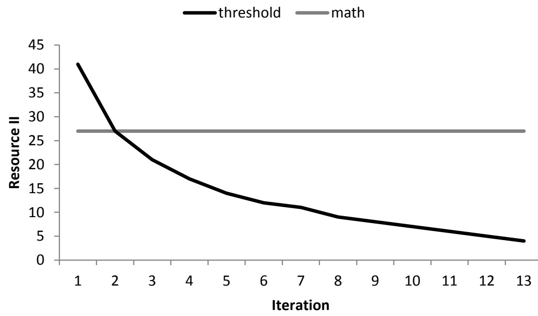

Figure 10. PET kernel II progression.

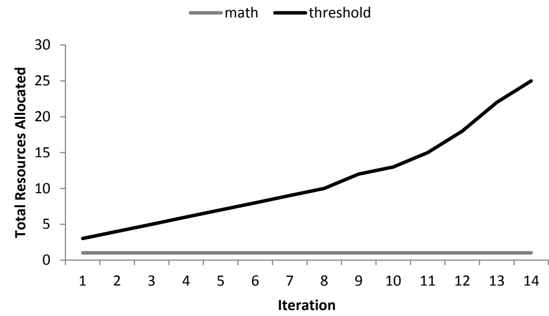

Figure 11. PET resource assignment progression.

the elements are spread out more, or even separated. The second measure used is the bounding box perimeter. This is simply the perimeter of the smallest rectangle encompassing all members of the kernel. The overall cost assigned to the kernel is the larger of the actual perimeter and bounding box perimeter. The bounding box helps guide separated elements back together. For example, a two element kernel where the resources are not adjacent will have the same perimeter cost regardless of their separation, so the bounding box dominates in this case to help drive them back together.

The same basic approach is applied to inter-kernel communication as well. The difference in this case is that each pair of communicating kernels is treated as a single super kernel for the purposes of calculating the aforementioned cost. Thus, in this case the perimeter is only counted if an element is not adjacent to another element from the same kernel or the kernel on the other end of the stream currently being evaluated. All of the same goals apply here as well: minimizing the perimeter will minimize the area so it should be tightly packed, and the bounding box will help drive together separated regions. The total cost function for the system is then the sum of the individual kernel costs and each super kernel representing each inter-kernel stream in the application. The VPR [11] cooling schedule is used to control temperature adjustments for the annealing.

Table 2. Global Placement Results.

App	Min Cost	Generated Cost	Cost Ratio	Avg WL	Max WL	Kernels
DWT	100	108	1.08	1.0	1.0	3
PET	44	44	1.00	1.0	1.0	2
Bayer	176	182	1.03	1.0	1.0	5
IPL	156	164	1.05	1.0	1.0	4
Wavelet	476	522	1.10	1.2	5.0	18

Figure 9 shows a simple two kernel example similar to the PET event detection benchmark with two different placements to illustrate the different components of the overall cost metric. The small kernel itself has a perimeter of 4 in either case, while the large kernel cost changes depending on whether it has a concave shape. Since the two kernels communicate, their resources are pooled together to calculate the perimeter once again, which is the same in either case here. In these two cases, the bounding box option is not used because it is never greater than the perimeter calculation for this example.

Even without any optimizations for calculating an incremental cost function per move, the placement phase of the floorplanning executes for no more than 40 seconds on a modest desktop for the most complex benchmark, again a runtime dwarfed by other tools in the Mosaic flow.

5. RESULTS

We present results for multi-kernel benchmarks written in Macah to demonstrate the floorplanning flow. For the global placements, the best of ten runs is shown to demonstrate the effectiveness of the approach. The Resource Allocation progression for the PET application is shown Figure 10 and Figure 11 which demonstrates optimizing the application with different port rates.

While the Resource Allocation process is optimal as shown in the appendix, Global Placement is based on a heuristic. Five multi-kernel benchmarks were run through the floorplanner with the results summarized in Table 2. The Min Cost field is the theoretical minimum placement cost achievable for the given resource allocation. This is calculated as the sum of the minimum rectangular regions for each kernel and pair of communicating kernels, similar to the cost function used in the actual placement. This minimum is generally unachievable in practice, since the placement of different kernels interact. The generated cost is for the best of ten runs of the benchmark through the Global Placement phase with the Cost Ratio indicating the increase over the theoretical minimum. Avg WL (wire length) is the average minimum distance between pairs of communicating kernels as defined in the application while the Max WL is the largest distance.

As can be seen, the placer achieves layouts within 10% of the lower bound in all cases, with a geometric mean of 1.05. Inter-kernel signals are almost always of length 1, meaning communicating kernels are adjacent for all but 1 signal in Wavelet.

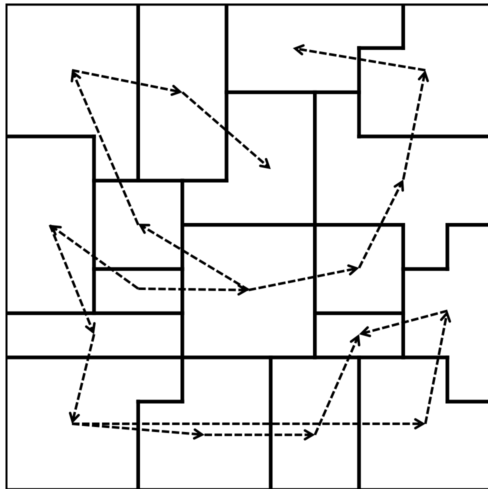

Figure 12. Wavelet kernel placement.

Detailed floorplans for several interesting cases are shown in Figure 3, Figure 4, and Figure 12. As you can see the results are well packed, communications are short, and individual kernels have reasonable shapes.

6. CONCLUSION

We have presented an algorithm for floorplanning multi-kernel applications on CGRAs. From a description of the inter-kernel communication pattern and basic parameters of the kernels, the algorithm divides the available resources among the kernels in order to maximize throughput. It then provides a high level placement of the kernel resources in order to facilitate global routing. This in turns enables detailed scheduling, placement and routing of each kernel to efficiently map multi-kernel applications onto the reconfigurable fabric.

7. ACKNOWLEDGEMENTS

The authors would also like to thank Nathaniel McVicar for his support in benchmark development.

8. REFERENCES

[1] Mosaic Research Group. http://www.cs.washington.edu/research/lis/mosaic/.

[2] Knight, A., Multi-Kernel Macah Support and Applications, M.S. Thesis, University of Washington, Dept. of EE, 2010. https://www.ee.washington.edu/people/faculty/hauck/publications/KnightMS.pdf.

[3] Brian Van Essen, Improving the Energy Efficiency of Coarse-Grained Reconfigurable Arrays, Ph.D. Thesis, University of Washington, Dept. of CSE, 2010. http://www.ee.washington.edu/faculty/hauck/publications/dissertation-vanessen.pdf

[4] Ylvisaker, B., "C-Level" Programming of Parallel Coprocessor Accelerators, Ph.D. Thesis, University of Washington, Dept. of CSE, 2010. http://www.ee.washington.edu/faculty/hauck/publications/macahThesis.pdf

[5] Friedman, S., Carroll, A., Van Essen, B., Ylvisaker, B., Ebeling, C., and Hauck, S. 2009. SPR: an architecture-adaptive CGRA mapping tool. In Proceeding of the ACM/SIGDA international Symposium on Field Programmable Gate Arrays (FPGA '09). ACM, New York, NY, 191-200.

[6] Xilinx, PlanAhead 13.2 User Guide. http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_2/PlanAhead_UserGuide.pdf

[7] Lei Cheng; Wong, M.D.F.; , "Floorplan design for multi-million gate FPGAs," Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on , vol., no., pp. 292- 299, 7-11 Nov. 2004

[8] John M. Emmert and Dinesh Bhatia. 1999. A methodology for fast FPGA floorplanning. In Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays (FPGA '99).

[9] Gordon, M. I., Thies, W., and Amarasinghe, S. 2006. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In Proceedings of the 12th international Conference on Architectural Support For Programming Languages and Operating Systems. San Jose, California, USA, October 21 - 25, 2006. ASPLOS-XII. ACM, New York, NY, 151-162.

[10] Butts M., Jones A., Wasson P., A Structural Object Programming Model, Architecture, Chip and Tools for Reconfigurable Computing. In Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '08), pages 55–64, April 2008.

[11] Betz, V. and Rose, J. 1997. "VPR: A new packing, placement and routing tool for FPGA research". *7th international Workshop on Field-Programmable Logic and Applications* (September 01 - 03, 1997).

[12] Panda, R.; Hauck, S.; "Dynamic Communication in a Coarse Grained Reconfigurable Array," Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on , vol., no., pp.25-28, 1-3 May 2011.

9. APPENDIX – OPTIMALITY PROOF

An application is modeled as a directed graph. Nodes correspond to Macah kernels and edges represent streams between kernels. Each edge terminal represents a port connecting a kernel to a stream. The application has the following properties that are inputs to the resource allocation algorithm.

- $RecII_k$ is the kernel's intrinsic minimum recurrence initiation interval. The kernel cannot

execute faster than this rate of a fixed number of cycles per iteration. This entails loop-carried dependencies or other sequencing constraints in that kernel.

- Ops_k is the number of operations necessary to execute kernel k. If the kernel has Ops_k operations, and is assigned R_k resources, then it cannot execute faster than *ceiling*(Ops_k/R_k) cycles per iteration due to the limits of time-multiplexing those R_k resources.
- Min_k: The minimum size of a kernel is the number of stateholding elements or other resources that cannot be time-multiplexed. Also, if the device has a maximum supported II, Min_k must be at least *ceiling*(Ops_k/MaxII).
- $Rate_{k,i}$: The rate at which data is produced or consumed by kernel k at port i will need to be reconciled to match the port-rate at the other end of the stream. Thus, a value of 1 means the port transfers a value once each iteration of the kernel.
- TotRes is the total number of resources available in the device. We must ensure that the total number of resources assigned across the kernels is no larger than this value.

The system has bounded-sized buffers, and the kernels communicate on the streams via handshaking. That means that if a kernel attempts to read from an empty stream or write to a full stream (a stream whose buffer is already filled with data), that kernel stalls until the precipitating condition is resolved.

We assume the system is connected, which means any kernel is linked to another kernel in the design.

The system computes the following derived properties as it performs the resource allocation:

- R_k: The number of resources assigned to kernel k. To be legal, we know that $R_k >= Min_k$. Also, $TotRes \geq \sum_k R_k$. If $TotRes < \sum_k Min_k$ then no legal solution exists.
- II_k: This is the achieved II of kernel k. Achieved II is the average II of that kernel, once stalls are taken into account, of the system operating at steady-state. To be legal, we know $II_k \geq RecII_k$ and $II_k \geq$ *ceiling*(Ops_k/R_k).

9.1. Goal

The goal is to assign resources to kernels to maximize steady-state performance. We can define performance as the rate data is sent on some designated stream, but as we will show later, under our model, maximizing the rate of data sent on any specific stream maximizes the rate of data sent on all streams in the system.

9.2. Implications of steady-state

We define steady-state in the device as the point where the production and consumption rates on the streams in the system stabilize. When the program begins operating, some kernels will stall because they do not have any data to start operating, and other kernels may produce data at a higher than sustainable rate because some stream buffers have not yet filled. However, over time the system will reach an equilibrium, where the average production and consumption rates on each of the streams in the system will stabilize. We define these average rates as the steady-state performance.

Consider some kernel k_1 in the system, with at least two ports p_1 and p_2. Given our assumptions, at steady-state we know that each of these ports has a fixed data transaction rate relative to II_{k1}, and thus they have a fixed data transaction rate relative to each other. That is, for every data transaction on p_1, there are $Rate_{k1,p2}/Rate_{k1,p1}$ data transactions on p_2.

A similar relationship holds between ports of a given stream. At steady-state, we know that the rate of data written to the stream and data read from the stream must be the same. Otherwise the stream will either empty causing stalls on the consumer, or fill causing stalls on the producer.

At steady-state there is a fixed relationship between the port rates of both ends of a stream, between the ports on a given kernel, and between a kernel's II and its port's rates. There is also at steady-state a fixed relationship between the II of kernels connected directly by a stream. Via transitivity, any two kernels that are connected by streams, either directly or indirectly, must have a fixed relative II at steady-state. Since we only consider connected designs, this means that, at steady-state the IIs of all kernels and the communication rates of all streams have a fixed relative relationship.

The major implication of this discussion is that a resource allocation with a higher throughput on any stream has a higher throughput on all streams, and has a lower achieved II on all kernels.

9.3. Proof of optimality

The algorithm starts with a minimum allocation of resources to each kernel. Based on this allocation, the II of each kernel can be calculated as well as the rates of data produced and consumed at the boundaries of each kernel. With this information and the constant relationships between kernels connected by streams, the kernel or kernels limiting the overall throughput of the application is

determined. Since these kernels are limiting throughput, the algorithm works to improve performance of these kernels by allocating more resources, specifically the minimum number of resources necessary to reduce the II of the kernels in question. No intermediate allocation of resources is needed since these kernels cannot improve in performance until the II can be reduced. The analysis of which kernels impact throughput is performed again and the process continues until the limiting kernel has reached its recurrence II or the device is full, meaning that no further improvement is possible.

It is also important to note that allocating more resources to a kernel will never increase the II nor reduce the throughput of the kernel. The relationship between resource allocation and II of the kernel is a monotone decreasing function.

With this information about the system the proof is framed as a contradiction.

The resource allocation algorithm is run assuming unlimited resources with the understanding that the recurrence minimum II will be reached before the resource-constrained minimum II. Our algorithm produces a series of solutions S_{min}, … S_{max}, with corresponding total resource usages T_{min}, … T_{max}. S_{min} is the first solution produced by our algorithm, which assigns the minimum legal number of resources to each kernel.

Assume that there is a division of resources S with a total number of resources T that is the solution generated by the resource allocation algorithm.

For the solution found by the algorithm we have $T \geq T_{min}$, since if $T < T_{min}$ there is some kernel given fewer resources than is legal, and if $T = T_{min}$ then $S = S_{min}$ since there is only one legal solution with T_{min}. We also know that $T \leq T_{max}$, because at T_{max} at least one kernel is running as fast as its $recII_k$ allows (otherwise the algorithm would produce another solution), which means the highest possible performance of any solution is S_{max}. Therefore $T_{min} \leq T \leq T_{max}$.

If it were proposed that a different solution S' exists that would improve S, the solution determined by our algorithm, there would be a different resource allocation to at least one kernel in the application.

We now consider two cases: (1) any kernel which is assigned fewer resources in S' than in S; and (2) any kernel which is assigned more resources in S' than in S.

(1) any kernel assigned fewer resources in S' than in S

There are two possibilities. Either the kernel $k_{reudced}$ being considered is at its minimum resource allocation as in S_{min}, or it had been allocated additional resources sometime during the execution of the algorithm. If $k_{reduced}$ was at its minimum resource allocation, S cannot legally assign fewer resources to it. On the other hand, if the kernel was allocated more resources during the run, then it was allocated n more resources for a total of $R_{previous} + n = R_{improve}$ resources, where $R_{previous}$ was the number of resources allocated before being increased. $R_{improve}$ is the number of resources allocated incorporating the increment n to increase throughput. Also, kernel $k_{reduced}$ would only have had its resources increased if at $S_{previous}$ it was never stalling. In other words, there was a solution $S_{previous}$ whose performance was limited by setting $k_{reduced}$ to have $R_{previous}$ resources, and it could only be sped up by adding at least n more resources. Thus, S' must have a throughput of at most $S_{previous}$, which has a worse throughput than S. This is a contradiction.

(2) any kernel assigned more resources in S' than in S

Note that in our algorithm the only kernels that have their resource assignments changed are those that never stall on a given iteration. Any kernel that does stall would have the same resources assigned in both S' and S. The kernel must not have stalled if it was assigned more resources, and thus is a member of the set of kernels limiting performance or reached its maximum resource allocation as in S_{max}.

The algorithm initiates the R_k of every kernel to begin at S_{min} and increments R_k of each limiting kernel to improve throughput. The search of the solution space is monotonically increasing in terms of the resource allotments to those kernels which limit performance. Furthermore, only the minimum number of additional resources is allocated to improve performance. Our algorithm terminates when the next increment to all the limiting kernels exceeds TotRes or if a limiting kernel is operating at its recurrence constrained II. Thus, if any kernel in S' were allocated a greater number of resources than in S, then S' would allocate more resources than necessary to achieve the same performance as in S and such a solution would not be generated by our algorithm.

If it were supposed that, at maximum resource usage, a kernel with greater R_k in S' than in S meant there existed another kernel with lower R_k in S' than in S, we know this would contradict the result in (1).

9.4. Conclusion

Via proof by contradiction, we have demonstrated that for any given resource constraint, there are no solutions that provide a better throughput than the ones given by our algorithm. Thus, the resource allocation algorithm generates the Pareto-optimal set of solutions for this problem, and selects the highest throughput design that meets the resource constraints of the target chip. Therefore, given the problem definition and assumptions specified, our resource allocation algorithm is optimal.