15745 Final Project
Instruction Scheduling and Token Allocation for Modular Time-Multiplexing on CGRAs

Mitchell Fream(mfream@andrew.cmu.edu)
Xuesi Chen(xuesic@andrew.cmu.edu)

***Abstract*—Compilers for coarse-grained reconfigurable arrays (CGRA) often focus on accelerating regular workloads, such as: inner loop kernels. Recent works on energy-minimal dataflow compiler and architecture co-design successfully achieved mapping irregular workloads on CGRAs, but at the expense of performance and hardware utilization. We propose breaking down the dataflow graph (DFG) of irregular workload, into dataflow blocks, which are segments of regular executions within an irregular workload. To ensure energy efficiency, we further statically schedule instructions in a dataflow block into Stages. The main focus of this class project is to instruction schedule within dataflow blocks and do token analysis to refine instruction scheduling that would minimize the number of output queues required per processing element (PE).**

I. Introduction

A. Backgound

Extreme edge device deployment on chip-scale satellites, wildlife monitoring and medical devices are expected to operate on ultra-low power (<1mW) for years [11], [15], [3]. Given the low energy budget and diverse applications extreme-edge processing needs to accommodate, the computer platform should be energy efficient, programmable, and as performant as possible [7].

The CGRAs, consisted of a grid of processing elements (PEs) connected by on-chip networks (NoC), which is a good execution platform for energy-minimal dataflow architecture. Prior CGRA-based energy-minimal dataflow architectures [6], [4] avoid control and data movement overheads intrinsic to von Neumann architecture, allowing operands to fire in-order upon data arrivals. Unlike application-specific integrated circuits (ASICs), the programmability of CGRAs allow computations in diverse applications (e.g. machine learning, signal processing) [5] with multiple generations of algorithmic updates [8].

B. Related Works

Previous CGRAs [9], [1], [16], [13] use time-multiplexing to improve performance, but at the cost of programmability or energy efficiency, making them unsuitable to extreme edge computing. HyCUBE [9], RE-VAMP [1], ULP-SRP [10], IPA [2] execute statically scheduled instructions, allowing both spatial and temporal instruction mapping to PEs to increase performance. However, their applications are limited to inner loop kernel accelerations.

While Revel[16] and Fifer [13] did address nested loops and other irregular control, they either employs energy draining queues for inter-stage data transfers [13] or dataflow PEs with costly instruction buffers and register files [16]. Thus, they are not suitable for energy-minimal edge computing.

The state-of-art energy-minimal order-dataflow CGRA architecture, RipTide, has the ability to execute arbitrary code written in C. RipTide adopts many techniques that trade area and performance for energy efficiencies. As a result, RipTide reduces energy so much that it is unable to use all available energy in certain environments [3] (e.g. from a solar panel under direct sunlight). Hence, it makes sense to ask: **how can we improve performance with a small cost in energy?**

One key energy-saving technique RipTide uses is mapping only one operand to each PEs to avoid reconfiguration. Scheduling instructions across space but not time introduces spatial inefficiency that ultimately hurts area and

performance. Often times, instructions corresponding to the outer loop execution is left idle while waiting for the inner loop execution to finish. Thus, we believe introducing time-multiplexing to RipTide will render a performant CGRA edge processor that has the programmability to execute arbitrary code written in C. Time-sharing PEs can also decrease the number of PEs needed for execution, fitting complex programs that previously need larger fabrics or allowing multi-tenancy.

C. Our Approach

We propose a compiler and architecture co-design that divides a DFG of any arbitrary program into dataflow blocks in the compiler. A dataflow block is a sequence of related dataflow CGRA stages, such as sub-computations of the inner loop. Within each dataflow block, compiler schedules stages that specify what instructions will be time-multiplexed to the CGRA upon reconfiguration. During execution, the architecture rotates between stages within a block until it runs out of work to do. Static-routed NoC is chosen for the execution model for energy efficiency. As a result of choosing Static-routed NoC, instructions within a dataflow block needs to be statically scheduled by the compiler, which the part of the problem we are tackling with this project.

To compile a program for a time-multiplexing dataflow machine, several additional constraints must be met by the compiler:

- all instructions that receivers of tokens produced outside of the current dataflow block are scheduled in the first stage. This is so that we can reduce hardware complexity and design energy-efficient CGRAs.
- no instructions which use the same hardware queue can allow their tokens to interfere. This is due to the nature of ordered dataflow [16].

Our project investigates the requirements to compile programs for a time-multiplexed CGRA based dataflow processor, specifically the means for scheduling instructions within dataflow blocks and the resource utilization constraints for communication between processing elements.

D. Contributions

We present a method for compiler scheduled stages within a dataflow-block. The question of breaking the dataflow graph apart into multiple dataflow blocks is not the scope of the project and is already well-addressed by prior preliminary study. We show that these dataflow blocks are suitable for use as the smallest unit of dynamic scheduling and that the stages inside them require only static scheduling, reducing dynamic scheduling overhead in energy. We show that by using dataflow-block-based time-multiplexing and compiler-automated static instruction scheduling, performance per area improvements of 2.5x to 3x can be achieved. Next, we explain how time-multiplexing complicates both hardware design and resource allocation. We then show how this problem can be ameliorated for acyclic dataflow graphs through an analog to register allocation which decreases the total number of hardware resources required. We give some insight into why minimizing hardware resources is not always optimal. Next, we present a hardware mechanism which allows for this solution to be extended to cyclic graphs as well and show that the number of hardware queues required can be decreased by 41% to 55%, which decreases the chip size required to run a given workload. Finally, we show that the number of hardware queues can be decreased by 20% to 37% without any loss in performance by further constraining instruction placement to limit the number of stages in each dataflow block.

II. Design

A. Instruction Scheduling

The first instructions to be executed in most schedulers are those with no predecessors. In our case, the first stage of a dataflow block is populated with instructions that have inputs from outside this dataflow block. The hardware execution model ensure that this dataflow block will not be scheduled until all the inputs from other blocks are present, so these instructions will always be ready to fire when the dataflow block is mapped. From here, successive stages are built by looking at the immediate successors of instructions. The dataflow hardware

works to hide instruction latency, meaning that even in cases where there is unequal instruction latency, scheduling should still be done considering immediate successors rather than instruction latencies. This strategy creates stages with no regard to the total number of hardware processing elements available, however, so the final step is to move instructions out of overfilled stages into neighboring stages until all constraints are satisfied. Because the dataflow engine allows tokens to be queued for an arbitrary amount of time and for instructions which are mapped to only fire when their inputs are available, there is no correctness concern when moving instructions between stages within a dataflow block.

Algorithm 1 Instruction Scheduling Algorithm

```
Require: a dataflow block (instr, E), number of PEs
  currStage = instr that are receivers of const or
  outputs of other dataflow blocks;
  mark all currStage instrs as visited by parents;
  stages = {}

  while ! all instrs are visited by all of their parent
  instrs do
      push currStage to the end of stages;
      mark all child instrs visted by the currStage
  instrs;
      currStages = {};
      add instrs visited by all parents instrs in the
  currStage;
  end while

  II = number of instrs % number of PEs
  for all instrs in Stages that exceeds II do
      targetStageIdx = stageIdx % II;
      if targetStageIdx has available PEs then
          insert instr at the end of targetStage;
      else
          insert instr to a closest stage with an available
  PE;
      end if
  end for
  erase all stages that exceeds II;
```

B. Token Allocation

Our hardware model can make use of existing register allocation algorithms, but requires a differently constructed interference graph. Because a dataflow program does not encode a specific schedule, a more conservative interference graph is required. In particular, where a von Neumann processor can explicitly order instructions and therefore guarantee that a given value lifetime ends before another begins, a dataflow program may not be able to specify. This leads us to develop the following definition for the register coloring interference graph in a dataflow system as well as a sufficient criterion for noninterference in acyclic dataflow graphs.

Definition II.1 (Interfere)**.** Two instructions interfere if and only if there exists a schedule where the live range of one instruction reaches the other instruction.

Definition II.2 (Acyclic noninterference Criterion)**.** Two instructions in an acyclic dataflow graph do NOT interfere if there is a path in the DFG from all the uses of one instruction to the other instruction.

The acyclic noninterference criterion shows that two instruction do not interfere because it ensures that all the uses of the first instruction occur before the definition of the second instruction occurs. Paths through a dataflow graph indicate true dependencies and therefore guarantee instruction ordering in any schedule. This allows us to compute interference without needing to iterate over all valid schedules, and is the definition most directly implemented in our code. It is necessary to specify an acyclic dataflow graph because there may be multiple dynamic instances of an instruction in a dataflow graph containing cycles, in which case precedence would not ensure that they are not reordered with one another. Unfortunately, our set of benchmarks does not contain any acyclic dataflow graphs upon which to test this code. In fact, the benchmarks we used do not contain any noninterfering edges, as they all have highly parallel loops which permit many possible schedules. For this reason, we investigated a hardware mechanism to allow for more precise queue reuse.

A concrete algorithm for determining interference edges is presented in Algorithm 2.

Algorithm 2 Queue Interference Detection Algorithm

Require: A dataflow graph (instr, E)
  for all edges e in dataflow graph **do**
    if e points to a Carry or Invariant instruction **then**
      Remove e
    end if
  end for
  Mark all instructions unvisited
  Initialize descendants of all instructions to the empty set
  while Not all instruction are visited **do**
    let i be an instruction whose successors have all been visited
    for all successors j of instruction i **do**
      copy all descendants of j to i
    end for
    Mark i as a descendant of itself
    Mark i as visited
  end while
  Now every instruction is annotated with all of its descendants
  for all pairs i,j of instructions in that dataflow graph **do**
    Presumptively mark i,j as interfering
    if i and all of i's children have j as a descendant **then**
      mark i,j, as noninterfering
    end if
    if j and all of j's children have i as a descendant **then**
      mark i,j, as noninterfering
    end if
  end for
  Now use any graph coloring algorithm to obtain a coloring from the interference graph

In fact, coloring the graph is not sufficient to place instructions in this system even given an optimal schedule. A coloring of instructions tells us which instruction would benefit from sharing the same processing element, but it does not fully specify how we should have those instructions mapped. One solution would be to simply add dataflow block stages to ensure that every color can be placed entirely on a single processing element. This would yield the best results in terms of queue usage, but would decrease processing element utilization by leaving more processing elements unmapped in an average cycle. Thus, the number of colors required for a function provides us with a lower bound on the number of queues required in hardware to support that function, but does not necessarily provide us with the optimal implementation for any hardware with additional queues. This motivates a second strategy which is to minimize the number of queues used without adding any stages. Due to the parallel and fully connected nature of our hardware, moving instructions between processing elements within a single stage should never change the total execution time of a program. Thus, given that our original schedule performed well, we can find a schedule which performs equally well but uses fewer hardware queues. An algorithm to achieve this is presented here as algorithm 3.

Algorithm 3 Greedy Queue Reduction Algorithm

Require: A schedule of all instructions into stages
  Mark all instructions as not fixed
  for all Pairs of instructions i,j which are the same color and in different stages **do**
    if i and j are mapped to different processing elements **then**
      Find instruction k in the same stage as j which is mapped to the same PE as i
      Swap the processing elements for instructions j and k unless either is fixed
      Mark i and j as fixed
    end if
  end for

This serves to greedily group same color instructions to the same processing elements without ever slowing down the program. This mapping is certainly not optimal, as grouping instructions this way actually amounts to a second graph coloring problem, which would of course not be fully optimized by a greedy algorithm. Nevertheless, the results we achieved with this level of sophistication were quite promising.

In order to work with cyclic graphs, we make the assumption that we can break all edges leading to Carry or Invariant gates. This solution is, in essence, tagged dataflow. By tagging each token with its loop iteration number, hardware can disambiguate multiple instances of a single static instruction. If we then replace our queues with random access memories, we can map each dynamic iteration number to an address in the memory. This fully removes any problematic reordering across loop iterations (i.e. in DFG cycles) by more precisely allocating memory resources to

each iteration. Doing this allows us to transform our cyclic dataflow graphs into acyclic graphs by breaking edges where hardware tag changes are enforced. In RipTide, order among different loop iterations is maintained by Carry and Invariant instructions; these instructions are the logical choice for where to change tags, and so we break DFGs at the inputs to Carry and Invariant gates.

III. Experimental Setup

Our execution model was based on the RipTide processor. In particular, each processing element is equipped with a number of output buffers which is uses to communicate with other processing elements. The depth of these buffers is not essential to our contribution, but as per the original paper it was set at 4. In order to allow for time-multiplexing, it is required that each processing element have multiple output buffers, as it may be used for multiple instructions which need to send tokens to different destinations.
We used our lab group's dataflow simulator to run program binaries and count the fraction of processing elements which were in use in each cycle. time-multiplexing enabled binaries used fewer processing elements, so they were more efficient in terms of firing processing elements per clock cycle.

In order to evaluate the effectiveness of the queue reuse algorithm, we compare the number of queues needed by the naive mapping to the number required by our optimized mapping. Because we investigated small kernels and because of the extreme constraints that dataflow places on the allocation algorithm, we do not see very large amounts of queue reuse.

IV. Experimental Evaluation

We verified the correctness of the instruction scheduling using dense matrix vector multiplication, dense matrix matrix multiplication and dense matrix convolution benchmarks with various input sizes.

In our first experiment, we targeted a dataflow processor with an unlimited number of hardware queues. This means that the compiler needs only to optimize for performance

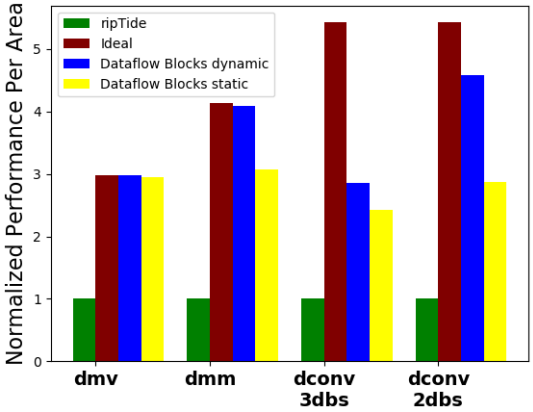

Fig. 1. Performance Per Area Compared to RipTide and Ideal

per unit area, as there is no failure criterion for our mapping if it uses too many queues. To this end, we use algorithm 1. In this case, we find that all of our workloads show a large improvement in performance per area over RipTide and that our output only falls short of the ideal performance per area by a small amount. This suggests that there is a large potential for time-multiplexed execution on smaller hardware, but only if the memory resources for execution can be scaled down similarly to the compute resources.

In a second experiment, we consider a queue limited machine. To this end, we use algorithms 2 and 3 to calculate an upper bound on the minimum number of queues required if we allow for performance losses and an upper bound on the number of queues required to achieve the same performance as in experiment 1.

In a third experiment, we manually construct a dataflow graph which trades one additional dataflow stage for one less queue in dmv. By selecting an instruction to move in an outer loop, we only incur 101 cycles of extra execution time ($<$0.2% cycle overhead) but move from 12 queues required to 11. This demonstrates that, with proper selection, it is possible to trade very little performance loss for decreased queue usage. This points to a need for algorithmic exploration of the design space between full queue count optimization and full performance optimization.

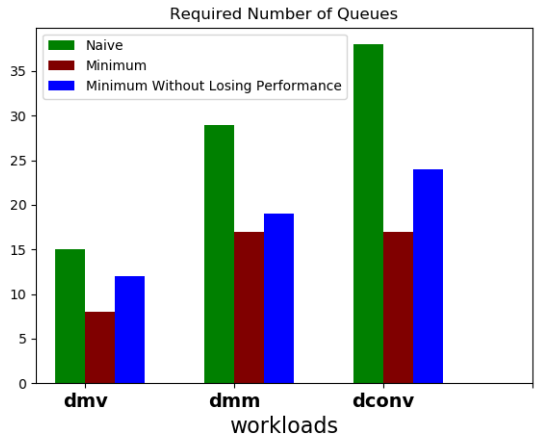

Fig. 2. Number of Queues Required for Different Optimization Criteria

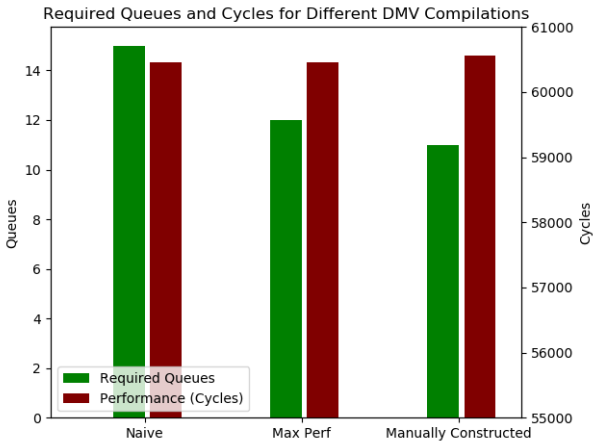

Fig. 3. Comparison of Required Hardware Resources and Required Execution Time

V. Surprised and Lessons Learned

One large surprise in this work was how often nodes interfere for the purpose of register allocation in a dataflow architecture. Because dataflow architectures enforce no particular ordering on instruction execution other than the true dependencies in the dataflow graph, there are many possible execution orders for a given set of instructions. This already limits the cases where registers (or in our case, queues) can be reused, as there are fewer guarantees about liveness. In particular, consider the following code snippet.

```
int a,b,c,d = ...;
int tmp1 = a + b;   //(1)
int e = tmp1 * 5;   //(2)
int tmp2 = c + d;   //(3)
int f = tmp2 * 10;  //(4)
```

A von Neumann execution model would require explicitly scheduling these statements in some order. In particular, if the operations were scheduled in program order then tmp1 and tmp2 would not interfere, as the last use of tmp1 happens before tmp2 is defined. This is not the case in a dataflow processor, however. Under the dataflow execution model, there is no static schedule ordering the execution of these statements other than the true dependences $1 \rightarrow 2$ and $3 \rightarrow 4$. This means that in our machine, there are no non-interfering edges for the purpose of register allocation. This situation gets even worse for cyclic dataflow graphs. Consider the following snippet.

```
while(i<N) {
     int tmp1 = arr_in[i]; //(1)
     int tmp2 = tmp1 * 5;  //(2)
     int tmp3 = tmp2 + 7;  //(3)
     arr_out[i] = tmp3;
     i += 1;
}
```

This example appears at first glance to be more hopeful. The true dependencies between statements ensure that statements within each loop iteration will execute in order, and therefore that tmp1, tmp2, and tmp3 could be allocated to the same register, but this analysis fails to consider additional loop iterations. Because our machine has queues connecting processing elements, it is possible that as many as 4 iterations of statement 1 have occurred before a single iteration of statement 2. This means that we still cannot reuse registers in this case. In fact, this is a general property of loops in our case unless those loops have true dependencies between every consecutive statement as well as a true dependency between the last and first statement. In the case of the snippet above, there is only a write after write dependence (which is a false dependence) ordering successive executions of statement 1, so there is no opportunity for optimization here given the Riptide hardware model. This caused us to reevaluate our hardware model and introduce a limited notion of tagging.

It was a pleasant surprise how little per-

formance was lost in our hand constructed DMV mapping. We expected, based on previous literature's focus on decreasing initiation interval, that adding stages in such a way that initiation interval increased would have disastrous effects on performance. We attribute the very small loss in performance to the fact that we selected an infrequently executed dataflow block to slow down, specifically one in an outer loop.

VI. Conclusions

We have shown that CGRA based dataflow architectures can achieve massive improvements in performance per area by decreasing the number of processing elements on a chip and time-multiplexing instructions. We show that breaking a dataflow graph apart into statically schedulabled dataflow blocks achieves a 2.5x to 3x performance per area increase over pure RipTide without time-multiplexing, and that this scheme achieves nearly the performance per area of an optimally scheduled execution. We have also shown that the size of programs which can be mapped to a given hardware configuration can be improved by reusing hardware queues through an analog to register allocation, which can lead to a 41% to 55% decrease in the number of hardware queues needed for a given workload. Taken together, this implies a massive overprovisioning in existing extreme edge CGRA based dataflow machines. Existing workloads could run as the same performance on CGRA architectures with one third as many processing elements and half as many communication channels through smarter compilation.

VII. Future Work

In order to make better use of our register allocation scheme, future work should explore tagged token dataflow more deeply for area limited applications. By using a limited form of tagging, write after write dependencies can be removed, which allows for much better register allocation results in cases which contain loops. We expect that further optimizations could decrease the number of queues needed for some programs by taking advantage of other facets of tagged dataflow, improving the size of programs which can be run on a given hardware configuration.

In addition, there are many interesting tradeoffs that future research could investigate in regards to CGRA based dataflow. The general problem that large programs will face in an increase in the number of required hardware queues. The queue pressure in a dataflow machine increases faster than the register pressure in a comparable von Neumann machine because of the lack of a static schedule, which causes many more instructions to interfere with one another for the purpose of queue coloring. This points to a possible solution, however. Excessive parallelism, that is, having more ready instructions than available processing elements, does not improve performance. Thus, in some cases, explicitly limiting the scope of possible schedules could improve queue allocation pressure without costing any performance. To do this, a compiler would simply need to add extra ordering edges to the dataflow graph. This would cause some instructions to now be guaranteed to execute before other instructions which they previously were not, which would decrease the number of interference edges in the queue coloring graph, leading to a reduction in the number of required hardware resources.

Conversely, there is no benefit to using fewer hardware queues than are available. In the same way that a von Neumann machine would gain no benefit by only using 12 out of 16 registers, there is no reason to minimize the number of queues used in a dataflow machine. Because allocating two instructions to use the same output buffer constrains them to be placed on the same processing element, it often comes with a performance cost. Increasing the number of stages in a dataflow block to map multiple instructions to the same processing element requires increasing the initiation interval for that dataflow block, which has a direct impact on latency and throughput. Thus, the decision about which dataflow blocks to slow down given the need to reuse queues is nontrivial. A starting point for future research would be to add stages preferentially to dataflow blocks which are outside of loops, saving the majority of the

hardware queues for inner loop computations. This is similar to how it is preferable to spill values outside of loops in a von Neumann processor to leave as many registers available as possible inside the loop. We showed a hand crafted example of this in experiment 3, but leave a full exploration of programmatically deciding where to add stages for future work.

In the same way that there is no performance benefit to leaving hardware queues empty, there is no benefit to leaving processing elements idle. This points to the possibility of loop unrolling as a way to increase PE utilization, especially in inner loops. Future work would do well to explore the trade off between better PE utilization at the cost of increased queue usage due to static instruction bloat caused by loop unrolling.

Lastly, there is significant room for exploration in the space of memory spilling for CGRA based architectures. Similarly to von Neumann architectures, it may be necessary or beneficial to store values in memory rather than taking up valuable hardware queues with them. The means to efficiently realize this is nontrivial, however, as the additional load and store instructions would cause a greater increase in register pressure in a CGRA based dataflow machine than in a von Neumann machine, offsetting a portion of the benefit of spilling.

VIII. Project Logistics

We feel that credit should be evenly divided.

References

[1] T. K. Bandara, D. Wijerathne, T. Mitra, and L.-S. Peh, "Revamp: A systematic framework for heterogeneous cgra realization," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 918–932.

[2] S. Das, D. Rossi, K. J. Martin, P. Coussy, and L. Benini, "A 142mops/mw integrated programmable array accelerator for smart visual processing," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2017, pp. 1–4.

[3] B. Denby and B. Lucia, "Orbital edge computing: Nanosatellite constellations as a new class of computer system," in *ASPLOS 25*, 2020.

[4] G. Gobieski, A. O. Atli, K. Mai, B. Lucia, and N. Beckmann, "Snafu: an ultra-low-power, energy-minimal cgra-generation framework and architecture," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 1027–1040.

[5] G. Gobieski, N. Beckmann, and B. Lucia, "Intermittent deep neural network inference," in *SysML Conference*, 2018, pp. 1–3.

[6] G. Gobieski, S. Ghosh, M. Heule, T. Mowry, T. Nowatzki, N. Beckmann, and B. Lucia, "A programmable, energy-minimal dataflow compiler and architecture," pp. 546–564, 2022.

[7] G. Gobieski, B. Lucia, and N. Beckmann, "Intelligence beyond the edge: Inference on intermittent embedded systems," in *ASPLOS*, 2019.

[8] N. P. Jouppi, D. H. Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma *et al.*, "Ten lessons from three generations shaped google's tpuv4i: Industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 1–14.

[9] M. Karunaratne, A. K. Mohite, T. Mitra, and L.-S. Peh, "Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[10] C. Kim, M. Chung, Y. Cho, M. Konijnenburg, S. Ryu, and J. Kim, "Ulp-srp: Ultra low-power samsung reconfigurable processor for biomedical applications," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 3, pp. 1–15, 2014.

[11] B. Lucia, V. Balaji, A. Colin, K. Maeng, and E. Ruppel, "Intermittent computing: Challenges and opportunities," *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, 2017.

[12] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler, "A design space evaluation of grid processor architectures," in *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*. IEEE, 2001, pp. 40–51.

[13] Q. M. Nguyen and D. Sanchez, "Fifer: Practical acceleration of irregular applications on reconfigurable architectures," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1064–1077.

[14] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ilp, tlp, and dlp with the polymorphous trips architecture," in *Proceedings of the 30th annual international symposium on Computer architecture*, 2003, pp. 422–433.

[15] F. Tavares, "Kicksat 2," May 2019. [Online]. Available: https://www.nasa.gov/ames/kicksat

[16] J. Weng, S. Liu, Z. Wang, V. Dadu, and T. Nowatzki, "A hybrid systolic-dataflow architecture for inductive matrix algorithms," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 703–716.