A Comparative Study on the Architecture Templates for Dynamic Nested Loops

Jason Cong and Yi Zou
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095, USA

***Abstract*—Loops are the most typical constructs that the FCCM community tries to accelerate. Many loop constructs have dynamic loop bounds and may face load balancing issues in their parallel realizations. In this paper we discuss different architecture templates for these dynamic nested loops, and show the benefits and trade-offs between various implementation strategies. We show that a statically scheduled architecture may perform better if the overhead in banking conflicts overwrites the benefits in load balancing by dynamic scheduled architectures.**

I. Introduction

FPGA is frequently used as a loop accelerator. The primary reason is that these loops typically have a large degree of parallelism where multiple instances of the loop body can execute simultaneously.

A loop typically has an iteration space. Some loops have fixed iteration spaces, and many loop optimization or transform techniques such as iteration space tiling/slicing [1] and affine transforms [2] can be easily applied. When the iteration space is not a fixed one, many of these techniques do not directly apply. We denote these loops as dynamic nested loops.

Actually, these dynamic loops are frequently encountered in practice. For example, in sparse matrix vector multiplication, the dynamic behavior occurs because of the fact that the number of non-zeros in each row/column is not the same. In hypergraph traversal, the number of pins for each node or each hyperedge is different. Recently, there is a growing interest in porting map-reduce framework onto FPGAs [3], [4]. This can also be viewed as a case for dynamic loops. Multiple map (or reduce) instances can run in parallel, and there is no guarantee that these instances shall finish in a constant cycle count.

Let's look at a typical dynamic loop: sparse matrix vector multiplication (code shown in Fig. 1). The sparse matrix here is represented in a compressed row format. Clearly, the inner loop bound is not a constant value. Multiple inner loop instances can execute in parallel.

Each execution of the inner loop is roughly proportional to the loop bound of the inner loop (and they differ row by row). Thus, parallelizing the outer loop in a naive way may cause load balancing issues. We can also leverage fine-grain parallelism to parallelize the inner loop using a tree-adder type of architecture to do the reduction.

For the sake of simplicity, we assume we are dealing with a two-level nested loop similar to Fig. 1. The loop bound of the inner loop varies between instances. There are no write-conflicts or any dependencies between multiple inner loop instances, and their executions can be arbitrary ordered. These conditions can be checked easily using standard compiler techniques.

```
for(i=0;i<n;i++){
  temp=0;
  for (k=rows[i];k<rows[i+1];k++)
      temp+=val[k]*x[col[k]];
  y[i]=temp;
}
```

Fig. 1. Sparse matrix vector multiplication

There are mainly three ways to implement the hardware architecture for these dynamic nested loops: approach A: parallelizing the inner loops (the loops with dynamic bounds); approach B: parallelizing the outer loops with static allocation/scheduling; and approach C: parallelizing outer loop with dynamic allocation/scheduling. In this paper, we mainly want to evaluate three architecture templates and discuss their trade-offs. We find that static allocation, with various kinds of static allocation strategy, is often the best architecture among the three. It is simple to implement and is very effective. The dynamic allocation approach, although attractive, faces many practical difficulties or overheads. These overheads include data access conflicts, serialization in the centralized scheduler and queue structures etc. Efforts to reduce the overhead are also presented.

We use SPMxV as a representative example to make this comparative study, but the discussion presented in our paper is broadly applicable to many other dynamic nested loops. SPMxV is the key computation kernel in a wide range of applications (e.g., quadratic placement in EDA domain, level-set solver for medical image segmentation etc.) We also need to point out that FPGA-based implementations for SPMxV have been heavily studied by the FCCM community in the past. A tree-adder based approach is implemented in [5] and [6]. The work in [7] parallelizes multiple sparse dot products and also discusses load balancing improvement through graph partitioning. The work [8] partitions the matrix into multiple strips and implements a streaming approach.

II. Architecture Templates and Implementations

This section presents the three architecture templates in the general sense, and illustrate their implementations on the SPMxV example. We assume the computation is done in fixed point, and all the data required is already stored on-chip.

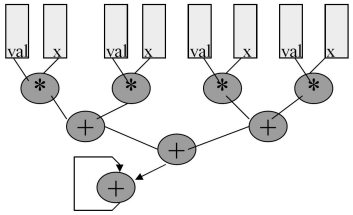

Fig. 2. Tree adder with banked storage

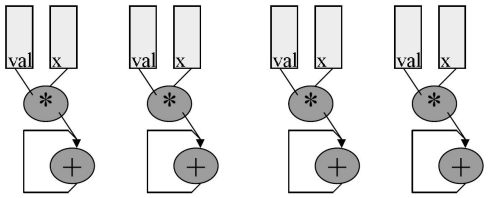

Fig. 3. Multiple PEs with local storage

A. Parallelizing inner loops

Because the loop bound of the inner loop is not fixed, complete loop unroll is not possible. However, partial unroll of the inner loop is possible. For SPMxV, the result of the partial unroll can be viewed as an architecture which uses a tree-adder as the atomic unit. The tree adder has a fixed number of multipliers in the leaf nodes, and a fixed number of adders in the non-leaf nodes.

In most cases, the latency of this atomic unit is larger than one and the unit should be pipelined if possible.

The loop bound of the inner loop may not be a multiple of the partial unroll factor. To cope with this, dummy data (zero) should be added into the input of the atomic unit. Clearly, this brings in the architecture overhead, because the relative resource utilization rate is lower if dummies are added.

Each architecture template requires a specific memory arrangement scheme to achieve efficiency. Because the inner loop is partially unrolled, this unrolling increases the number of memory accesses of the inner loop body. We can use a cyclic memory partitioning scheme so that multiple data accesses (leaf nodes of the tree-adder) always access data in different banks. Fig. 2 shows the tree adder with one final accumulator. val (and col, but not shown in the figure) are banked using cyclic partitioning, while vector x is duplicated because it uses indirect data access.

The inner loop of SPMxV can be parallelized. However, the inner loops of some applications have certain loop-carried dependencies that enforce strictly sequential executions. For example, in random walk simulation, different walks of one trajectory seem to be strictly sequential. The architecture templates that leverage coarse-grain parallelism should be used instead.

B. Multi-PE realization with static allocation

As an alternative, we can unroll the outer loop and parallelize multiple inner loop instances (multiple dot products for SPMxV).

In this architecture, the atomic processing unit is the hardware realization of one inner loop instance. For SPMxV, this is one MAC (Multiply-And-aCcumulate) unit with some control logic for boundary checking. The parallelism comes from instantiating multiple processing units, and each unit is handling different inner loop instances.

The allocation of the inner loop instances into the processing elements affects the overall load balancing. This fact is also discussed in the SPMxV design in [7], but their goal is to mainly reduce communication messages rather than help load balancing.

The optimal allocation that minimizes the makespan of the parallel execution is NP-complete. Four static allocation technique are compared: cyclic (deterministic), random permutation, random and quasi-random. Note that these four techniques are general, and they do not make use of the loop bound of inner loops explicitly. More efficient static allocation can be done based on the analysis on those loop bounds. Cyclic assignment determines the assignment using equation $P(i) = i\%N_{PE}$, where N_{PE} is the number of PEs, i is the subtask ID and $P(i)$ is the assigned PE ID (from 0 to $N_{PE} - 1$) for subtask i. Random permutation generates one random permutation for each group of contiguous N_{PE} subtasks and uses the permutation to do the assignment. Random assignment or quasi-random assignment determine the assignment using equation $P(i) = Rand()\%N_{PE}$ or $P(i) = Qrand()\%N_{PE}$, respectively.

Each PE has its own local memory, and it works on these local memories independently. The data required for the computation is distributed using the allocation strategy. Fig. 3 shows the diagram of this architecture template for the SPMxV case.

C. Multi-PE realization with dynamic allocation

The subtasks that are mapped into the PEs can also be allocated dynamically at runtime. To support the dynamic allocation, we conceptually need a queue of idling PEs and a queue of pending subtasks. When a PE completes the execution of one subtask, its ID shall be added into the idling PE queue. A dynamic scheduler checks the status of these two queues and maps one pending subtask into one idling PE. This is the architecture for the dynamic allocation/scheduling implemented in [4].

A queue is typically implemented through FIFO. The elements in the idle queue are the IDs of PEs. FIFO can accept at most one input each cycle. When multiple PEs complete their subtasks in the same cycle, some arbitration logic (e.g., priority encoder) is needed and the IDs shall be added into the FIFO one by one. And a few more PEs may complete their subtasks in subsequent cycles, which further complicates the process.

Our implementation does not use any queue explicitly. Instead, the scheduler checks a group of (say K) idle signals

from K contiguous PEs every clock cycle, and assigns some new subtasks for the idle PEs among the K PEs. When K is smaller than number of PEs N_{PE}, the scheduler checks the next K contiguous PEs in the next cycle in a circular fashion. The circuity for assigning the subtasks for K contiguous PEs is a simple bit-counting logic. Let us denote b_0 to b_{K-1} as the idle signals for K contiguous PEs. The assignment can be known after $p_i = \sum_{j=0}^{i} b_j$ is computed. If $b_i = 1$, PE i shall get a subtask with ID $M + p_i$, where M is the last subtask ID that is assigned before this cycle.

The data required for the computation needs to be accessible by all the processing elements. If these data are in one big global buffer or in off-chip memory, the bandwidth may be a severe problem since it needs to serve all the PEs. We also statically partition the required data into N_{PE} banks to ensure that the available bandwidth in the dynamic allocation case is at least the same as the static allocation case. The banking is based on a cyclic partitioning scheme.

Currently, the interconnect structure between the banked on-chip storage and the PEs is implemented as full crossbar, where any PE can access any bank immediately if that bank is not accessed by others. Other interconnect structures such as ring can be implemented to further reduce interconnect complexity.

Dynamic allocation determines which subtask each PE shall process at runtime. When one PE is assigned one subtask, it requests the required data from the banked global buffer (or even off-chip memory) and starts computation. This data access or data copy is on the critical path and can not be overlapped, because we can not predict what data is accessed by specific PEs until run-time thus we can not use prefetching to hide data access latency. Because of this, paper [4] only did double buffer for common data path (broadcasted data).

There are two ways to resolve the issue. First, we can force the architecture to use double buffers at local PEs to do the latency hiding. Each PE shall have two (or more) slots of local storage. The dynamic allocation and scheduling determines the assignment and copies data into the slot that is recently used, but each PE still has another slot to process when the data copy occurs. Note the solution of this dynamic assignment is different then the original one. This approach tries to overlap data access and computation in a coarse-grain fashion. This idea is inspired by virtual function unit in the architecture of a super-scalar processor. Second, the computation can be overlapped with data access in a fine-grain fashion. For example, the computation of the sparse dot product is done in parallel when we fetch/access the required data (in a streaming fashion). We use the second approach because our data is already on-chip. However, the first approach is still useful for other dynamic loops if the fine-grain approach fails to overlap the data access and computation completely (for example, the data access needs to be done in a burst mode for off-chip access).

It is possible that one PE will try to access one data bank that is occupied by another PE in the dynamic allocation case. To reduce the overhead due to bank conflicts, we also try to use static allocation at the beginning of the computation, and then switch to a dynamic approach (this is essentially a type of work stealing) when the whole execution is about to finish.

III. Experiment Results

We implemented these architectures using VHDL and simulated them using ModelSim. Because we use fixed-point computation, we do not need to consider the extra complication due to the long latency of floating point units. Floating point SPMxV needs to use one MAC unit to compute multiple dot products in an time-multiplexed interleaved fashion to compensate the long latency of floating point units [7]. We assume a same clock frequency although our preliminary synthesis results suggest that the design with dynamic allocation gets a worse timing. We use 16 multipliers and 16 adders in all the designs ($N_{PE} = 16$). For approach A, this is translated into an adder-tree with 16 leaf nodes. For approach B and C, this is translated into a design with 16 MAC units.

We tested the first ten test matrices from the UFL sparse matrix collection [9]. The lower bound of the cycle count using N_{PE} multipliers and N_{PE} adders to compute a SPMxV with a sparse matrix with nnz entries is $\lceil nnz/N_{PE} \rceil$. The values are shown in column a) of Table I.

The cycle count data for approach A is shown in column b). Because the number of non-zeros in one row is typically small (average about 3 to 4), the overhead due to the dummies added is very large. However, this approach can be an effective solution if the inner loop bound is much larger than N_{PE}.

The total cycle count by approach B depends on the static allocation strategy. The results of cyclic allocation, random permutation allocation, random allocation and quasi-random allocation are shown in column c) to f). All four approaches are not far away from the lower bound (likely due to the central limit theorem). The cyclic or random permutation approaches are almost always better than random or quasi-random approaches. Because the latter two can not guarantee the number of instances allocated in different PEs are more or less the same.

The cycle count for approach C with zero-cycle overhead is shown in column g). Zero-cycle overhead means that the cycle-overheads in data transfer and arbitration are all ignored, and each PE can always get a new subtask (a new row) in zero cycle latency. The values in this column are very close to the lower bound $\lceil nnz/N_{PE} \rceil$, which suggests that dynamic allocation indeed improves the load-balancing considerably. But we can not get to zero-cycle overhead in the practical settings. The actual cycle counts for $K = 1$ and $K = 16$ are shown in column h) and i). We can see that the cycle count for $K = 1$ is very bad. The average cycle count for one subtask (one dot product) is 3 to 4 cycles, but we may need to wait up to 15 cycles to get a new subtask assigned in the $K = 1$ case. The $K = 16$ case is much better because it can assign a new subtask immediately if any PE signals the scheduler it

TABLE I
CYCLE COUNTS FOR TEST MATRICES

	Lower bound	Approach A	Approach B				Approach C			
	a)	b)	c)	d)	e)	f)	g)	h)	i)	j)
1138_bus	254	1139	283	294	312	317	255	1140	392	283
494_bus	105	494	117	120	135	117	106	496	165	114
662_bus	155	662	178	172	202	175	158	668	228	172
685_bus	204	685	223	224	257	249	206	686	298	218
abb313	98	313	103	103	121	139	100	318	134	101
arc130	65	149	99	93	120	120	67	159	100	99
ash219	28	219	28	28	40	36	28	220	28	28
ash292	138	292	158	150	211	180	141	296	197	148
ash331	42	331	42	42	66	56	42	332	42	42
ash608	76	608	76	76	96	86	76	609	76	76

is idle. However, due to the banking conflicts, the cycle count is still worse than static allocation approach for the majority of examples.

The last method makes the static assignment initially (cyclic allocation), but leaves the last $NumRows\%N_{PE}$ rows unassigned and makes decisions at runtime based on the progress of the computation. This is a simple heuristic that combines the benefits of dynamic and static allocation. The data is shown in column j). Further it reduces the cycle count as compared to the cyclic allocation. However, it still is some distance away from the lower bound. Other ways of combination, such as guided scheduling approach which gradually decreases the chunk size of allocation in runtime, can also be implemented.

Note we assume the data are already stored on-chip. This seems a little bit unfair and makes dynamic allocation unnecessary. But this is a valid assumption if we use SPMxV for iterative methods such as conjugate gradient methods. Multiple SPMxV shall be invoked in a sequential fashion. Most of our discussions are still valid if the data we need to fetch resides off-chip. For example, off-chip memory may have multiple memory modules and banks. The bandwidth shall be higher if we could avoid banking conflicts through static allocation. If we only have one bank of off-chip memory, the differences between approaches may become marginal and the off-chip bandwidth becomes the bottom neck for this application.

To summarize, the overhead of approach A is the resource underutilization due to dummies. It should work better if the inner loop bound is significantly larger than N_{PE}. The overhead of approach B is load imbalance, especially if we can not predict the workloads of individual inner loop instances in advance. The overhead of approach C mainly comes from bank conflicts and serialization of the centralized scheduler. Because the job allocation is performed dynamically at runtime, the data distribution can not be done statically and shall also be performed at runtime. It is important that we are aware of these different architecture templates and select a best one that matches a particular application.

IV. CONCLUSIONS AND FUTURE WORK

In this paper we study three architecture templates for dynamic loops. We present some practical issues in implementing dynamic architectures, such as implementing idle queue without FIFO, latency hiding techniques, etc. We show that the static allocation approach gives the best performance for the fixed-point SPMxV example. We summarized the trade-offs of different architecture templates in a general sense.

Our current comparative study is still preliminary, and there are a lot more architecture templates we did not explore. For example, we can first partial unroll the outer loop and then apply approach A. We can also restrict the banks each PE can access to simplify the interconnects for approach C. We should test different matrix examples from either quadratic placements or level-set segmentation, because these are the real applications we are actually targeting for. We shall also investigate more application examples to show the trade-offs for the architecture templates.

V. ACKNOWLEDGEMENTS

This work is partially funded by Center for Domain-Specific Computing (NSF Expedition in Computing Award CCF-0926127), and grants from Nvidia Corp. and Mentor Graphics Corp. under UC Discovery program.

REFERENCES

[1] M. Wolfe, "More iteration space tiling," in *Proc. Conference on Supercomputing*, 1989, pp. 655–664.

[2] A. Lim and M. S. Lam, "Communication-free parallelization via affine transformations," in *Proc. Workshop on Languages and Compilers for Parallel Computing*, 1995, pp. 92–106.

[3] J. H. C. Yeung et al., "Map-Reduce as a programming model for custom computing machines," in *Proc. Symposium on Field Programmable Custom Computing Machines*, Apr. 2008, pp. 149–159.

[4] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang, "FPMR: MapReduce framework on FPGA - a case study of RankBoost acceleration," in *Proc. Symposium on Field Programmable Gate Arrays*, Feb. 2010.

[5] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on FPGAs," in *Proc. Symposium on Field Programmable Gate Arrays*, 2005, pp. 63–74.

[6] J. Sun, G. Peterson, and O. Storaasli, "Sparse matrix-vector multiplication design on FPGAs," in *Proc. Symposium on Field-Programmable Custom Computing Machines*, 2007, pp. 349–352.

[7] M. deLorimier and A. DeHon, "Floating-point sparse matrix-vector multiply for FPGAs," in *Proc. Symposium on Field Programmable Gate Arrays*, 2005, pp. 75–85.

[8] Y. Elkurdi, D. Fernández, E. Souleimanov, D. Giannacopoulos, and W. J. Gross, "FPGA architecture and implementation of sparse matrix vector multiplication for the finite element method," *Computer Physics Communications*, vol. 178, pp. 558–570, Apr. 2008.

[9] T. A. Davis, "University of Florida sparse matrix collection," *NA Digest*, vol. 92, 1994.