Multilevel Granularity Parallelism Synthesis on FPGAs

Alexandros Papakonstantinou[1], Yun Liang[2], John A. Stratton[1], Karthik Gururaj[3],
Deming Chen[1], Wen-Mei W. Hwu[1], Jason Cong[3]
[1]Electrical & Computer Eng. Dept., University of Illinois, Urbana-Champaign, IL, USA
[2]Advanced Digital Science Center, Illinois at Singapore, Singapore
[3]Computer Science Dept., University of California, Los-Angeles, CA, USA.
[1]{apapako2,stratton,dchen,w-hwu}@illinois.edu, [2]{eric.liang}@adsc.com.sg, [3]{karthikg,cong}@cs.ucla.edu

Abstract— **Recent progress in High-Level Synthesis (HLS) techniques has helped raise the abstraction level of FPGA programming. However implementation and performance evaluation of the HLS-generated RTL, involves lengthy logic synthesis and physical design flows. Moreover, mapping of different levels of coarse grained parallelism onto hardware spatial parallelism affects the final FPGA-based performance both in terms of cycles and frequency. Evaluation of the rich design space through the full implementation flow - starting with high level source code and ending with routed netlist - is prohibitive in various scientific and computing domains, thus hindering the adoption of reconfigurable computing. This work presents a framework for multilevel granularity parallelism exploration with HLS-order of efficiency. Our framework considers different granularities of parallelism for mapping CUDA kernels onto high performance FPGA-based accelerators. We leverage resource and clock period models to estimate the impact of multi-granularity parallelism extraction on execution cycles and frequency. The proposed Multilevel Granularity Parallelism Synthesis (ML-GPS) framework employs an efficient design space search heuristic in tandem with the estimation models as well as design layout information to derive a performance near-optimal configuration. Our experimental results demonstrate that ML-GPS can efficiently identify and generate CUDA kernel configurations that can significantly outperform previous related tools whereas it can offer competitive performance compared to software kernel execution on GPUs at a fraction of the energy cost.**

Keywords-FPGA; High-Level Sytnthesis; Parallel Computing; Design Space Exploration

I. INTRODUCTION

Field Programmable Gate Array (FPGA) devices have been receiving increased attention and gaining growing market during the past two decades. Some of their key advantages include re-programmability (compared to ASICs), low power (compared to multicores and GPUs), long-term reliability (i.e. higher MTBF due to low temperature operation), real-time execution characteristics (i.e. do not depend on operating systems, caches or other non-deterministic latency sources) and high spatial parallelism that can be tailored to each application. Nevertheless, they suffer from usability challenges which consist of i) complex programming models (i.e. low level hardware description languages, e.g. VHDL, Verilog) and ii) complex and high-latency synthesis flows. As a consequence, few beyond hardware design experts adopt FPGA programming in practice.

Ongoing developments in the field of high-level synthesis (HLS) have led to the emergence of several industry [1-3] and academia powered [4,5] tools that can generate device-specific RTL descriptions from popular High-Level programming Languages (HLLs). Such tools help raise the abstraction of the programming model and constitute a significant improvement in FPGAs' usability. However, the sequential semantics of traditional programming languages greatly inhibit HLS tools from extracting parallelism at coarser granularities than instruction level parallelism (ILP). Even though parallelizing optimizations such as loop unrolling may help extract coarser-granularity parallelism at the loop level [6,7], the rich spatial hardware parallelism of FPGAs may not be optimally utilized, resulting in suboptimal performance.

Recent trends of CPU and GPU designs toward coarser grained parallel architectures (e.g. MultiCores and ManyCores) have led to a growing general usage of several new programming models [8-11] that explicitly expose coarse-grained parallelism. Some of these programming models, such as OpenMP, streaming languages, etc., have been adopted as programming interfaces for mapping application parallelism onto FPGA [12-14]. Moreover, the recently introduced CUDA (Compute Unified Device Architecture) [8] programming model by NVIDIA which provides a multi-threaded SPMD model for general purpose computing on GPUs has been selected as the FPGA programming model in the FCUDA framework [15].

All of these HLS works take advantage of the abundant spatial parallelism in FPGAs to exploit the inherent application parallelism. However, in most cases, application parallelism is extracted only from a single level of granularity (e.g. loop [13,6,7], stream pipeline [14] or procedure granularity [15]). Moreover, the impact of additional parallelism on frequency is either ignored (only cycles are reported) or dealt with via worst case synthesis conditions (i.e. target very low frequency). Exposing more parallelism will improve the number of execution cycles but may severely affect (more than 10X) the clock period due to wire routing and fan-out side effects. Thus, neglecting frequency may lead to a worse design in terms of both performance and area.

In this paper, we propose a Multilevel Granularity Parallelism Synthesis (ML-GPS) HLS-based framework for mapping CUDA kernels onto FPGAs. We leverage parallelism extraction at four different granularity levels: i) *array* ii) *thread*, iii) *core* and iv) *core-cluster*. By tuning parallelism extraction across different granularities, our goal

is to find a good balance between execution cycles and frequency. ML-GPS is based on existing HLS tools and provides an automated framework for i) considering the effect of multilevel parallelism extraction on both execution cycles and frequency and ii) leveraging HLL code transformations (such as unroll-and-jam, procedure call replication and array partitioning) to guide the HLS tools in multilevel granularity parallelism synthesis.

Exploration of several configurations in the hardware design space is often restricted by the slow synthesis and place-&-route (P&R) processes. HLS tools have been used for evaluating different design points in previous work [6,7]. Execution cycles and area estimates from HLS were acquired without going through logic synthesis of the RTL. Array partitioning was exploited together with loop unrolling to improve compute parallelism and eliminate array access bottlenecks. Given an unroll factor, all the non dependent array accesses were partitioned. Such an aggressive partitioning strategy may severely impact the clock period, though (i.e. array partitioning results in extra address/data busses, address decoding and routing logic for on-chip memories). In this work, we identify the best array partition degree considering both kernel and device characteristics through resource and clock period estimation models.

Various resource/frequency estimation models have been proposed [16-18], but not in conjunction with multi-granularity parallelism extraction. In this work, we propose resource and clock period estimation models that predict the resource and clock period as a function of the degrees of different parallelism granularities (array, thread, core and core-cluster). Additionally we incorporate physical layout information into the framework by partitioning the design into *physical layout tiles* on the FPGA (each core-cluster is placed in one physical tile). Our clock period estimation model takes into account the design resource usage and layout on the FPGA and predicts the clock period degradation due to wire routing. We combine our resource and period models with HLS tool execution cycle estimations to eliminate the lengthy synthesis and P&R runs during design space exploration. To explore the multi-dimensional design space efficiently, we propose a heuristic which leverages our estimation models along with a binary search algorithm to prune the design space and minimize the number of HLS invocations. Thus the ML-GPS framework can efficiently complete the design space exploration within minutes (rather than days if synthesis and physical implementation were used). More importantly, the design space point selected by the ML-GPS search is shown to provide up to 7X of speedup with relation to previous work [15], while achieving near optimal performance.

The main contributions in this paper are as follows:

- We describe an automated multilevel granularity parallelism synthesis framework for mapping CUDA kernels onto FPGAs.
- We derive accurate resource and clock period estimation models.
- We propose a design space exploration algorithm for fast and near-optimal multi-granularity parallelism synthesis.

II. Background and Motivation

The ML-GPS framework is based on the FCUDA framework [15] (referred to as SL-GPS hereafter) which demonstrates a novel HLS-based flow for mapping coarse-grained parallelism in CUDA kernels onto spatial parallelism on reconfigurable fabric. The SPMD CUDA kernels offer a concise way for describing work to be done by multiple threads which are organized in groups called *thread-blocks*. Each thread-block is executed on a GPU streaming multiprocessor (SM) which comprises of several streaming processors (SP) that execute the individual threads. SL-GPS converts the CUDA kernel threads into *thread-loops* which describe the work of CUDA thread-blocks. Each thread-loop is then converted into a custom hardware processing engine referred to as *core,* hereafter. Each core executes the iterations (aka *threads*) within its corresponding thread-loop in a sequential fashion and multiple cores are instantiated on the FPGA. Thus, coarse grained parallelism is extracted only at the granularity of thread-loops (i.e. CUDA thread-blocks). CUDA thread-blocks have properties that enable efficient extraction of application parallelism onto spatial parallelism. First, they execute independently and only synchronize through off-chip memory across kernel invocations. In SL-GPS this translates to multiple instantiated cores that can execute in parallel without long inter-core communication signals. Second, according to the CUDA programming model, each thread-block is assigned to one GPU SM with a private set of on-chip memories and registers. In SL-GPS this is leveraged by allocating private on-chip BRAMs and registers to each core, thus, eliminating memory access bottlenecks from shared memories and high fan-out loads from shared registers.

However, exposing parallelism at a single level of granularity may result in loss of optimization opportunities that may be inherent in different types and granularities of parallelism. Finer granularities offer parallelism in a more light-weight fashion by incorporating less resource replication at the expense of extra communication. On the other hand, coarser granularities eliminate part of the communication by introducing more redundancy. ML-GPS provides a framework for flexible parallelism synthesis of different granularities. In addition to the *core* granularity, the proposed framework considers the granularities of *thread, array* and *core-cluster*. As mentioned earlier, cores correspond to CUDA thread-blocks and in ML-GPS each core is represented by a procedure (which contains the thread-loop). Concurrent procedure calls are utilized to guide the instantiation of parallel cores by the HLS tool. Threads correspond to thread-loop iterations and are parallelized by unrolling the thread-loops. Array access optimization is facilitated by array partitioning (only for arrays residing in on-chip memories). Finally, core-clusters correspond to groups of cores that share a common data communication interface (DCI) and placement constraints. The placement constraints associated with each core-cluster enforce physical proximity and short interconnection wires between the intra-cluster modules. As shown in Fig. 1, the placement of each cluster is constrained within one *physical tile*.

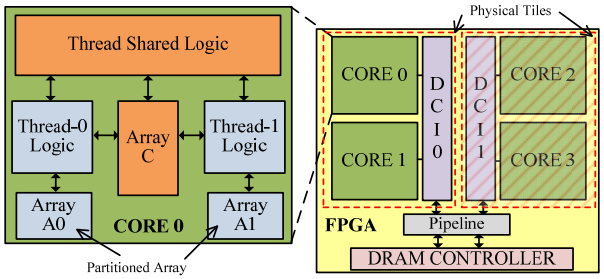


Figure 1. Thread, Core, Core-Cluster and Memory BW granularities

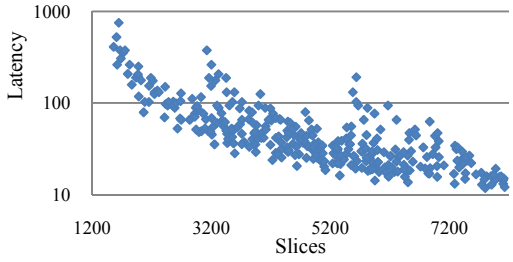


Figure 2. Design space of mm kernel

Both core- and thread-level parallelism extractions contribute to compute logic replication. However threads are more resource efficient (compared to cores) as they allow more sharing opportunities for memories, registers and other resource (Fig. 1). The downside of thread-level parallelism is longer and higher fan-out wires between shared resources and private logic of each thread as the degree of unroll increases. Cores on the other hand require fewer communication paths (only share DCI) at the expense of higher logic replication (Fig. 1). At an even finer granularity, array access parallelism is extracted through array partitioning and it enables more memory accesses per clock cycle, though at the expense of BRAM resources (each partition requires exclusive use of the allocated BRAMs) and addressing logic.

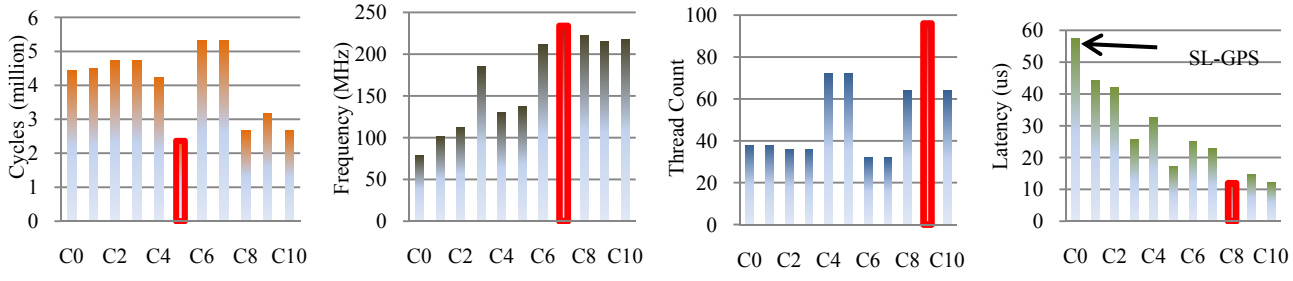
The DCI module includes the logic that carries out the data transfers to/from the off-chip memories through the DRAM controllers. Sharing a single DCI module among all the cores on the FPGA may result in long interconnection wires that severely affect frequency, annulling the benefit of core-level parallelism. As a downside, DCI units consume device resources while providing no execution cycle benefits. Core clustering helps eliminate long interconnection wires by constraining the cluster logic placement within physical tiles. Moreover, pipelining is used at the inter-cluster interconnection level (Fig. 1) to connect the DCI modules with the DRAM controller.

The optimal mix of parallelism extraction at different granularity levels depends on the application kernel characteristics as well as the resource characteristics of the FPGA device. Depending on the application, different granularity levels will affect execution cycles, clock frequency and resource usage in different degrees. Moreover, the absolute and relative capacities of different resource types in the targeted device will determine which granularity of parallelism is more beneficial.

Fig. 2 depicts a 2D plot of the design space for the *mm* kernel in terms of compute latency vs. resource (slices) usage. Each point represents a different configuration (i.e. combination of threads, cores, core-clusters and array partition degree). We observe that performance is highly sensitive to the parallelism extraction configurations. The depicted design space includes about 300 configurations and their evaluation through logic synthesis and P&R took over 7 days to complete. The charts in Fig. 3 offer a more detailed view of a small subset of design points in terms of cycles, clock frequency total thread count and latency, respectively. All of the configurations of the depicted subset have high resource utilization (greater than 75% of device slices) and span a wide range of the design space. The leftmost bar (C0) corresponds to the SL-GPS configuration which leverages only core-level parallelism, whereas the other configurations exploit parallelism in multiple dimensions. In each graph the highlighted bar corresponds to the best configuration with respect to the corresponding metric. As we can observe, C8 is the configuration with minimum latency, whereas different configurations are optimal in different performance related metrics (i.e. cycles, frequency and thread count). The charts demonstrate that i) single granularity parallelism extraction does not offer optimal performance and ii) performance constituents are impacted differently by different parallelism granularities.

III. ML-GPS Framework Overview

Before we introduce the ML-GPS framework, we first describe the corresponding source code transformations leveraged for the different parallelism granularities we consider.



a) Execution cycles b) Clock Frequency c) Hardware concurrency d) Execution Latency

Figure 3. Performance attributes of mm design space configurations

```
matmul_tblock( ...) {
  for(ty=0; ty<bDim.y; ty++)
    for(tx=0; tx<bDim.x; tx++) {
      for (k=0; k<BLK_SIZE; ++k)
        Cs[ty][tx] += As[ty][k] * Bs[k][tx];
    }
}
```

a) Original mm code

```
matmul_tblock( ...) {
  for(ty=0; ty<bDim.y/2; ty++)
    for(tx=0; tx<bDim.x; tx++) {
      for (k=0; k<BLK_SIZE; ++k)
        Cs[ty][tx] += As[ty][k] * Bs[k][tx];
        Cs[ty+bDim.y/2][tx] +=
            As[ty+bDim.y/2][k] *Bs[k][tx];
    } }
```

b) Unrolled thread-loop

```
matmul_tblock( ...) {
  for(ty=0; ty<bDim.y/2; ty++)
    for(tx=0; tx<bDim.x; tx++) {
      for (k=0; k<BLK_SIZE; ++k)
        Cs1[ty][tx] += As1[ty][k] * Bs[k][tx];
        Cs2[ty][tx] += As2[ty][k] * Bs[k][tx];
  }
}
```

c) Arrays A and C partitioned

```
for(by=0; by<gDim.y/2; by++)
   for(bx=0; bx<gDim.x; bx++) {
      matmul( ... )
      matmul( ... )
   }
```

d) Thread-block concurrency

Figure 4. Source code transformations (mm kernel)

Threads: unfolding of thread-loop iterations through unroll-and-jam transformations (Fig 4b).

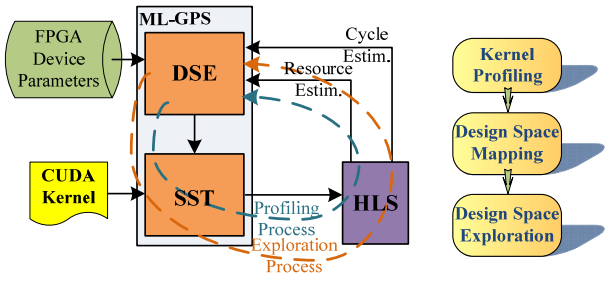
Array: on-chip array access concurrency is controlled by the degree of array partitioning, which divides arrays to separate partitions (Fig. 4c). Each partition is mapped onto a separate BRAM and thus, the array acquires multiple memory ports. In this work, array partitioning is applied only to arrays with affine accesses [19].

Cores: unfolding of threadblock-loop iterations through replication of thread-loop function calls (Fig. 4d). Each function call corresponds to the instantiation of one parallel core.

Core-cluster: the set of thread-blocks is partitioned to subsets, with each subset assigned to one core-cluster.

The ML-GPS framework leverages three main engines as depicted in Fig. 5a: i) a source-to-source transformation (SST) engine ii) a design space exploration (DSE) engine and iii) a HLS engine. The SST engine takes as input the CUDA code along with a set of configuration parameters that correspond to the degrees of the different parallelism granularities to be exposed in the output code. The configuration parameters are generated by the DSE engine which takes as input the target FPGA device data and determines the configurations that should be evaluated during the design space exploration. Finally, the HLS engine synthesizes the generated output code of the SST engine to RTL. In the ML-GPS framework we use a commercial HLS tool [2], which generates highly-optimized RTL code.

The ML-GPS flow involves three automated main steps (Fig. 5b). Initially a kernel profiling step is performed in order to build the resource estimation model for each kernel. Profiling entails feeding the SST engine with a small set of multilevel granularity configurations which are subsequently synthesized by the HLS tool to generate the corresponding resource utilization estimations. A kernel-specific resource model is then built using regression analysis on the HLS resource estimations. The number of the profiled configuration points determines the accuracy of the resource estimation model generated. More configurations result in more accurate resource models, though, at the expense of extra profiling time. In the ML-GPS framework the user can determine the effort spent on profiling.



a) ML-GPS components b) ML-GPS flow

Figure 5. ML-GPS Overview

After profiling, the design space is determined in the second main step. First the total number of core-cluster configurations is determined by considering both the resource estimation model generated in the 1st step (i.e. take into account the kernel characteristics) and the selected FPGA device (i.e. take into account the resource availability and distribution on the device). Subsequently the thread, array partitioning and core dimensions of the design space are determined for each core-cluster configuration with the help of the resource estimation model.

Finally in the third main step, design space exploration is performed using the resource and the clock period estimation models along with cycle estimates from the HLS tool to evaluate the performance of the different design points. A binary search heuristic is used to trim down the number of HLS invocations and prune the design space. The DSE engine's goal is to identify the optimal coordinates in the multi-dimensional parallelism granularity space in order to maximize the performance of the CUDA kernel on the selected FPGA device (i.e. given a fixed resource budget).

IV. Design Space Exploration

As mentioned previously, exploration of the multilevel granularity space is based on estimations of resource, clock period and cycles. We estimate resource and clock period degradation due to routing through regression analysis based equations, whereas we leverage cycle estimations from HLS. The formulas used for resource and clock period estimations are presented in the following section. To optimize the space exploration runtime we employ an efficient search optimization that helps minimize the number of HLS tool invocations during the search process. This is discussed in Section IV.B.

A. Resource and Clock Period Estimation Models

The resource model is built during the profiling step of the flow. A small number of points in the design space are used to generate different configurations of the input kernel exposing different granularities of parallelism. The HLS tool is fed with the profiled kernel configurations and it returns resource estimation results. We classify the resource estimations based on the degrees of parallelism exposed at the core (*CR*), thread (*TH*), and array-partitioning (*AP*) dimensions. Using linear regression we then evaluate the R_0, R_1, R_2 R_3 and R_4 coefficients of (1):

$$R = R_0 + R_1 \times CR + R_2 \times CR \times TH + R_3 \times CR \times AP + R_4 \times TH \times AP \quad (1)$$

Conceptually, the model characterizes the resource usage of a core-cluster based on the core number (R_1), count of threads (R_2), array partitioning (R_3), and the interaction between unrolling and array partitioning (R_4). For each type of resource (LUT, Flip-Flop, BRAM and DSP) we construct a separate equation which represents the core-cluster resource usage as a function of the different parallelism granularities. These equations are kernel-specific and are used during the design space exploration phase for resource budgeting as well as for estimating the clock period. The total resource count, R_{FPGA}, is equal to the product of the core-cluster resource estimation, R, and the number of physical tiles, CL, (i.e. number of core-clusters): $R_{FPGA}=R \times CL$.

The clock period model aims to capture the clock period degradation resulting from wire routing within the core-cluster. The HLS-generated RTL is pipelined for a nominal clock period defined by the user. However the actual clock period of the placed netlist is often degraded (i.e. lengthened) due to interconnection wire delays introduced during P&R. Through this model we incorporate the effect of different parallelism granularities as well as layout information on interconnection wires (i.e. wires within the core cluster; inter-cluster wires are pipelined appropriately, as mentioned earlier) and thus the clock period. The period estimation model is described by (2) which is pre-fitted offline using synthesis data (synthesized CUDA kernels were used for the period estimation model construction):

$$Period = P_0 + P_1 \times Diag + P_2 \times Util + P_3 \times AP + P_4 \times TH \quad (2)$$

Diag is calculated using (3) and it corresponds to the diagonal length (in slices) of a *virtual tile* with the following properties: i) the total core-cluster slices can fit in the virtual tile, ii) the dimensions of the virtual tile do not exceed the dimensions of the allocated physical tile and iii) the diagonal length of the virtual tile is minimal given the two previous constraints. *Util* in (2) represents the slice utilization rate of the physical tile by the core-cluster logic.

$$Diag^2 = \begin{cases} 2 \times R_{slice} & , if\ R_{slice} \le minDim^2 \\ minDim^2 + \left(\frac{R_{slice}}{minDim}\right)^2 & , if\ R_{slice} > minDim^2 \end{cases} \quad (3)$$

where *minDim* corresponds to the minimum dimension of the physical tile (in slices) and R_{slice} is the slice count of the core-cluster logic. Parameters R_{slice} (hence *Diag*) and *Util* in (2) are calculated by leveraging the resource model described above. Conceptually, parameter *Diag* incorporates the core-cluster resource area and layout information while *Util* incorporates the routing flexibility into the period model. *AP* and *TH* represent the requirement for extra wire connectivity within each core due to array partitioning and thread-loop unrolling.

B. Design Space Search Algorithm

1) Latency Estimation

Following the resource model construction for each kernel, the multi-dimensional design space can be bound given a resource constraint, i.e. an FPGA device target. Our goal is to identify the configuration with the minimum latency, *Lat*, within the bound design space. Latency is a function of all the parallelism granularity dimensions (i.e. thread (*TH*), array partitioning (*AP*), core (CR) and core-cluster (*CL*)) of the space we consider and is estimated using (4):

$$Lat(TH, AP, CR, CL) = Cyc \times \frac{N_{block}}{CR \times CL} \times Period \quad (4)$$

where N_{block} represents the total number of kernel thread-blocks, *Cyc* is the number of execution cycles required for one thread-block and *Period* is the clock period. As was discussed earlier, *Period* is affected by all the design space dimensions and is estimated through our estimation model in (2). On the other hand, *Cyc* is generated by the HLS engine and is only affected by the *TH* and *AP* dimensions (i.e. the HLS engine's scheduling depends on the thread-loop unrolling and array partitioning degrees). Thus for the design subspace that corresponds to a pair of unroll (u) and array-partitioning (m) degrees, *Lat* is minimized when the expression in (5) is minimized. We leverage this observation in tandem with our binary search heuristic to prune the design space and cut down the HLS invocations.

$$E(u, m, CR, CL) = \frac{N_{block}}{CR \times CL} \times Period \quad (5)$$

2) Binary Search Heuristic

The binary search heuristic is guided by the following observation:

Observation 1: For a given loop unroll factor, latency decreases monotonically first with a subsequent monotonic increase as the array partition degree increases.

Fig. 6a depicts the *fwt2* kernel latency for different unroll and array-partition pairs (u, m). For each point in Fig. 6a, its latency is determined by using the core (CR_u^m) and core-cluster (CL_u^m) values that minimize E in (5), and thus minimize *Lat*. We can observe (Fig. 6a) that the value of execution latency as a function of array partition degree for a fixed unroll factor decreases monotonically until a global optimal point, after which it increases monotonically. Intuitively, as the array partition degree increases, on-chip array access bandwidth is improved as more array references

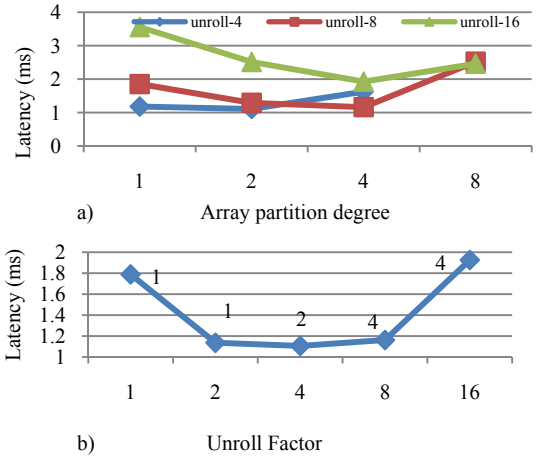


Figure 6. Unroll and Array partition effects on latency

Algorithm 1. Binary Search

```
1  /* search sequence: unroll followed by array partition */;
2  Search (1);

3  Search (dim)
4     if search all the dimensions then
5        Let (u, m) be unroll and array partition pair.
6        lat = Lat(u, m, CR_u^m, CL_u^m);
7        UpdateLatency (lat, u, m);
8        return lat;
9     Let space[] be the design space of dimension dim ;
10    low = 1; high = |Space| ;
11    while low <= high do
12       mid  = (low + high) / 2;
13       Select(dim, mid);
14       res_mid = Search(dim + 1);
15       Select(dim, mid + 1);
16       res_midPlus = Search(dim + 1);
17       if res_mid < res_midPlus then
18             high = mid - 1; /* search left */
19             Update(cur_best, res_mid);
20       else
21             low = mid + 2; /* search right */
22             Update(cur_best, res_midPlus);
23    return cur_best
```

can take place concurrently (i.e. execution cycles decrease). However, after a certain degree (saturation point), any further partitioning does not decrease clock cycles. Additionally it hurts frequency due to increased wire connectivity and higher logic complexity. More importantly, further partitioning may constrain coarse granularity extraction at the core and core-cluster levels as more BRAMS are used by each core. Thus, there exists an optimal array partition degree for each unroll factor. Observation 1 has been verified for other benchmarks as well.

A similar trend has also been observed for unroll factor (Fig. 6b). For each unroll factor u in Fig. 6b, its latency is determined by using its optimal array partition degree m from Fig. 6a and core (CR_u^m) and core-cluster (CL_u^m) values. Intuitively, as the unroll factor increases, more parallelism is exploited, thus improving the execution latency. However, unrolling beyond a certain degree may not be beneficial due to array access bottleneck and frequency degradation (from increased connectivity and fan-out issues). In summary, there is an optimal unrolling degree.

Based on the above observation and the intuition behind it, in Algorithm 1 we propose a binary search algorithm to find the optimal point (unroll factor u and array partition degree m). As shown in Algorithm 1, we search unroll factor first followed by array partition degree. Array *space[]* stores the feasible values for each dimension *dim*, in sorted order (line 9). The size and value of *space[]* are obtained from the resource model. Then, we perform binary search for dimension *dim*. In each round of the binary search (line 11-22), we compare the performance of two middle neighboring points (*mid*, *mid+1*). Function *Select* records the value selected for the dimension *dim*. The comparison result guides the search towards one direction (the direction with smaller latency) while the other direction is pruned away. In the end of the search across each dimension, the best result of the current dimension (in terms of execution latency) is returned. For each point visited during the search (i.e. (u, m) pair), the corresponding execution latency is computed based on (4) (line 6). The function *UpdateLatency* compares the current solution with the global best solution and updates it if the current solution turns out to be better.

Let us consider *fwt2,* shown in Fig 6, as an example. We start searching the unroll degree dimension and compare two neighboring points in the middle (2 and 4). For each unroll factor (2 and 4), its minimal latency is returned by recursively searching next dimension in a binary search fashion. The best solution so far is stored and the latency comparison of unroll factors (2 and 4) will indicate the subsequent search direction. The complexity of our binary search is $log|U| \times log|M|$, where U and M represent the design dimensions of thread and array partition.

V. EXPERIMENTAL RESULTS

The goals of our experimental study are threefold: i) to evaluate the effectiveness of the estimation models and the search algorithm employed in ML-GPS, ii) to measure the performance advantage offered by considering multiple parallelism granularities in ML-GPS versus SL-GPS [15] and iii) to compare FPGA and GPU execution latency and energy consumption.

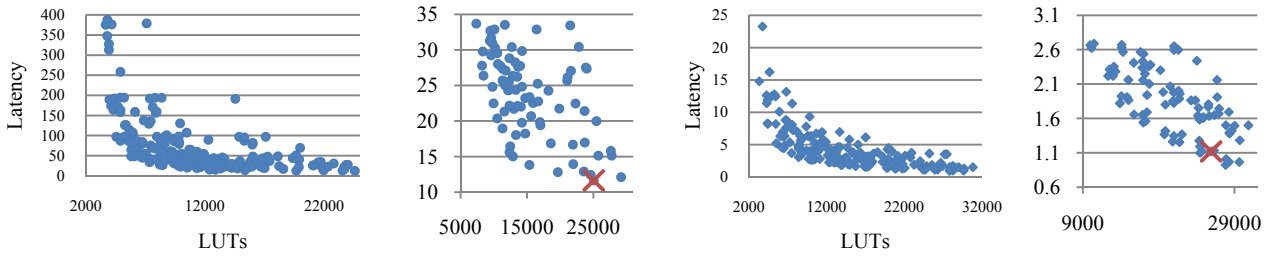
The CUDA kernels used in our experimental evaluations come from the CUDA SDK [8] and Parboil [20] suites and are described in Table I. The 1st column of Table I lists the application names and kernel aliases, the 2nd column details the input/output data dimensions and the 3rd column gives a short description of the application that corresponds to the kernel. In the experiments detailed in the following sections, we focus on integer computation performance and thus have modified the original kernels to include only integer arithmetic. Moreover, we explore the flexibility of the reconfigurable logic by evaluating the effect of different bitwidths on performance.

TABLE I. CUDA KERNELS

Application (Kernel Name)	Data Dimensions	Description
Matrix Multiply (mm)	4096x4096 arrays	Computes multiplication of two arrays (used in many applications)
Fast Walsh Transform (fwt1)	32MB Vector	Walsh-Hadamart transform is a generalized Fourier transformation used in various engineering applications
Fast Walsh Transform (fwt2)		
Coulombic Potential (cp)	512x512 grid, 40000 atoms	Computation of electrostatic potential in a volume containing charged atoms
Discreet Wavelet Transform (dwt)	120K points	1D DWT for Haar Wavelet and signals

A. ML-GPS Design Space Exploration

We have employed a mid-size Virtex 5 device (VSX50T) to explore the exhaustive design space of parallelism extraction in the multi-dimensional space we consider. Fig. 7a and 7c depict the entire design space for *mm* and *fwt2* kernels. Both maps consist of around 200 design points that have been evaluated by running the complete



a) *mm* design space b) *mm* design subspace c) *fwt2* design space d) *fwt2* design subspace

Figure 7. Multi-granularity parallelism design spaces

implementation flow: HLS followed by logic synthesis and P&R. Each design point corresponds to a unique configuration of *thread*, *array*, *core* and *core-cluster* parameters. Fig. 7b and 7d portray a subset of design points that are within 3X of the optimal configuration. The 'X' markers highlight the configuration point identified by the design space exploration (DSE) engine of the ML-GPS framework. Our experiments indicate that the configuration selected by the proposed search framework is, on average, within 10% of the optimal configuration's latency.

As described earlier, the DSE engine employs resource and clock period estimation models and invokes the HLS engine to profile the kernel and acquire cycle estimations. Thus, the design space exploration completes within several minutes compared to running synthesis and P&R which may require several days for multiple configurations.

B. ML-GPS versus SL-GPS

We compare the ML-GPS framework with SL-GPS [15] where parallelism was exposed only across the *core* dimension. Fig. 8 shows the normalized comparison data for a set of kernels. For these experiments we targeted a mid-size virtex5 FPGA device (VSX50T). The ML-GPS space exploration framework was used to identify the best configuration in the multi-granularity design space. Then the identified configuration was compared with the configuration that utilizes the maximum number of cores (SL-GPS) given the device resource budget. The comparison depicted in Fig. 8 is based on execution latencies derived from actual logic and physical synthesis implementations.

For each kernel we have generated two integer versions with different bitwidth arithmetic: 16-bit and 32-bit. Our experimental results show that performance is improved by up to 7X when multi-granularity levels of parallelism are considered. Note that for the *fwt1_16* kernel there is no performance improvement. The reason for this is due to the

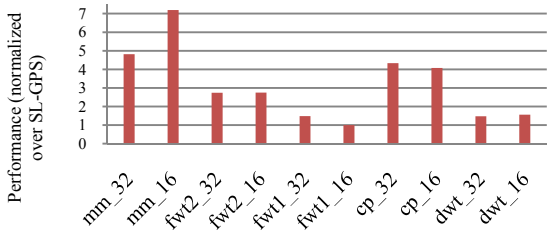


Figure 8. Performance comparison: ML-GPS vs. SL-GPS

multiple access patterns (with different strides) applied on the *fwt1* kernel arrays. This renders static array partitioning infeasible without dynamic multiplexing of each access to the right array partition. As a result, array partitioning is not considered for *fwt1* (in both bitwidth versions), thus impacting the performance contribution of unrolling (i.e. parallelism is exposed mainly across the *core* and *core-cluster* dimensions). The limited degrees of freedom in parallelism extraction result in small performance improvements for the 32-bit version and no performance improvement for the 16-bit version of *fwt1*.

C. ML-GPS versus GPU

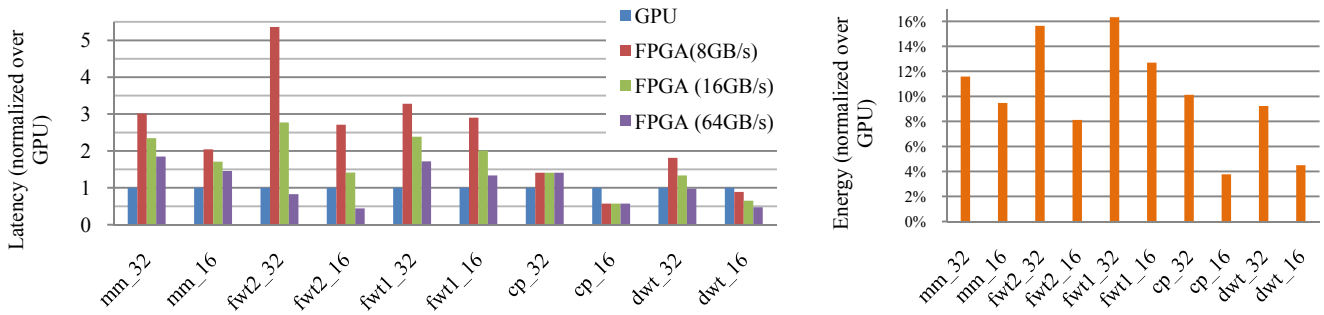
1) Performance

In this set of experiments we compare the performance of the FPGA-based hardware configuration identified by ML-GPS with the software execution on the GPU. For the GPU performance evaluation we use the Nvidia 9800 GX2 card which hosts two G92 devices, each with 128 stream processors. We utilize a single G92 device in our experimental setup. In terms of FPGA device we target one of the largest Xilinx Virtex5 devices (VSX240T) which includes a rich collection of embedded DSP (1056) and BRAM (1032) macros. The FPGA and GPU devices have been selected to ensure a fair comparison with regards to process technology (65nm) and transistor count.

In these comparison results we include both the compute latencies as well as the data transfer latencies to/from off-chip memories. The G92 device offers 64GB/sec peak off-chip bandwidth. For the FPGA device we evaluate three different off-chip bandwidth capacities: 8, 16 and 64GB/sec. Fig. 9a depicts the FPGA execution latencies for the ML-GPS chosen configuration, normalized with regards to the GPU latency. First, we can observe that the 16-bit kernels perform better than the corresponding 32-bit kernel versions on the FPGA (note that the GPU execution latencies are based on the 32-bit kernel versions). This is due to smaller data communication volumes (half-size values), as well as higher compute concurrency (smaller compute units allow higher concurrency). Second, off-chip bandwidth has a significant effect on performance, especially for kernels with high off-chip bandwidth data traffic (e.g. *fwt2*). With equivalent off-chip bandwidths (i.e. 64GB/s), the FPGA is faster than the GPU for half of the kernels.

2) Energy

Using the same FPGA and GPU devices as previously, we evaluate energy consumptions. For the GPU device we



a) Latency comparison

b) Energy consumption comparison

Figure 9. FPGA vs. GPU latency and energy comparison

use the reported power consumption of 170W (i.e. 270W system – 100W board). For the FPGA chip we use the Xilinx Power Estimator tool. The comparison results are depicted in Fig. 9b, normalized with regard to the GPU results. The energy consumed by the FPGA execution is less than 16% of the GPU energy consumption for all of the kernels. The 16-bit kernels show significant energy savings compared to the corresponding 32-bit versions due to fewer DSP and BRAM macro resources utilized per operation and operand, respectively.

VI. CONCLUSIONS

In this paper we present a novel framework, ML-GPS, for automated extraction of multilevel granularity parallelism from CUDA kernels to hardware spatial parallelism on the FPGA. The ML-GPS framework addresses the usability challenges of FPGAs. It combines the higher programming abstraction offered by HLS with multilevel granularity parallelism synthesis enabled by source code transformations (SST engine) and a fast design space exploration (DSE) engine. By leveraging efficient and accurate resource and clock period estimation models the proposed framework guides the design space exploration towards a configuration that is customized for the target FPGA and delivers high compute performance. Our experimental results show that ML-GPS achieves up to 7X speedup compared to SL-GPS [15]. In comparison with GPU, FPGA (ML-GPS) is shown to have competitive performance at a much lower power footprint. The current framework implementation leverages the CUDA programming model (thread and thread-block) and the AutoPilot HLS tool (procedure level parallelism support). Other SPMD-based programming models (e.g. OpenCL [9]) and alternative HLS tools could be easily supported in the future.

ACKNOWLEDGMENT

This work is partially supported by the Gigascale Systems Research Center (GSRC) and the Advanced Digital Sciences Center (ADSC) under a grant from the Agency for Science, Technology and Research of Singapore.

REFERENCES

[1] Mentor Graphics "Catapult C Synthesis", 2010, http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/

[2] Z. Zhang et al., "Autopilot: a platform-based ESL synthesis system," in High-Level Synthesis: from Algorithm to Digital Circuit, P. Coussy and A. Morawiec, Eds. Netherlands: Springer, 2008.

[3] Impulse Accelerated Technologies, "Impulse CoDeveloper," 2010, www.impusec.com

[4] P. Diniz et al., "Automatic Mapping of C to FPGAs with the DEFACTO Compilation and Synthesis System," Microprocessors & Microsystems, vol. 29(2-3), 2005, pp. 51-62.

[5] J. Cong et al , "Platform-Based Behavior-Level and System-Level Synthesis," Proc. IEEE Int. SOC Conf., 2006.

[6] H. Ziegler and M. Hall. "Evaluating Heuristics in Automatically Mapping Multi-Loop Applications to FPGAs," Proc. ACM Int. Symp. Field Programmable Gate Arrays (FPGA'05), 2005.

[7] B. So et al., "Using Estimate from Behavioral Synthesis Tools in Compiler-Directed Design Space Exploration," Proc. IEEE/ACM Design Automation Conf. (DAC), 2003.

[8] NVIDIA, "CUDA Zone", 2011, Accessed Mar 2011, http://www.nvidia.com/object/cuda_home_new.html

[9] Khronos, "The OpenCL specification, version: 1.1," http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf, 2010.

[10] OpenMP, "OpenMP application program interface," http://www.openmp.org/mp-documents/spec30.pdf, 2008.

[11] INTEL, "Sophisticated library for vector parallelism – Inter Array Building Blocks," http://software.intel.com/en-us/articles/intel-array-building-blocks/, 2010.

[12] D. Cabrera et al., "OpenMP Extensions for FPGA Accelerators," Proc. IEEE Int. Conf. Systems, Architecture, Modeling & Simulation (SAMOS'09), 2009.

[13] Y.Y. Leow et al., "Generating Hardware from OpenMP Programs," Proc. IEEE Int. Conf. Field Programmable Technology (FPT), 2006.

[14] A. Hagiescu et al., "A Computing Origami: Folding Streams in FPGAs," Proc. IEEE/ACM Design Automation Conf. (DAC), 2009.

[15] A. Papakonstantinou et al., "FCUDA: Enabling Efficient Compilation of CUDA Kernels onto FPGAs," Proc. IEEE Symp. Application Specific Processors (SASP'09), 2009.

[16] D. Julkarni et al. "Fast Area Estimation to Support Compiler Optimizations in FPGA-based Reconfigurable Systems," IEEE Symp. Field-Programmable Custom Computing Machines (FCCM), 2002.

[17] A. Nayak et al. "Accurate Area and Delay Estimators for FPGAs," Proc. IEEE Conf. Design & Testin Europe (DATE'02), 2002.

[18] S. Bilavarn et al. "Design Space Pruning Through Early Estimations of Area/Delay Tradeoffs for FPGA Implementations," Computer-Aided Design of Integrated Circuits and Systems, Vol. 25(10), 2006.

[19] J.Cong et al., "Automatic Memory Partitioning and Scheduling for Throughput and Power Optimization", Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD'09), 2009.

[20] IMPACT Rresearch Group, "Parboil Benchmark Suite", http://impact.crhc.illinois.edu/parboil.php, Univ. of Illinois, 2010.