SESH framework: A Space Exploration Framework for GPU Application and Hardware Codesign

Joo Hwan Lee
School of Computer Science
Georgia Institute of Technology
Atlanta, GA, USA
joohwan.lee@gatech.edu

Jiayuan Meng
Leadership Computing Facility
Argonne National Laboratory
Argonne, IL, USA
jmeng@alcf.anl.gov

Hyesoon Kim
School of Computer Science
Georgia Institute of Technology
Atlanta, GA, USA
hyesoon@gatech.edu

***Abstract*—Graphics processing units (GPUs) have become increasingly popular accelerators in supercomputers, and this trend is likely to continue. With its disruptive architecture and a variety of optimization options, it is often desirable to understand the dynamics between *potential* application transformations and *potential* hardware features when designing future GPUs for scientific workloads. However, current codesign efforts have been limited to manual investigation of benchmarks on microarchitecture simulators, which is labor-intensive and time-consuming. As a result, system designers can explore only a small portion of the design space. In this paper, we propose SESH framework, a model-driven codesign framework for GPU, that is able to automatically search the design space by simultaneously exploring *prospective* application and hardware implementations and evaluate *potential* software-hardware interactions.**

I. Introduction

As demonstrated by the supercomputers Titan and Tianhe-1A, graphics processing units (GPUs) have become integral components in supercomputers. This trend is likely to continue, as more workloads are exploiting data-level parallelism and their problem sizes increase.

A major challenge in designing future GPU-enabled systems for scientific computing is to gain a holistic understanding about the dynamics between the workloads and the hardware. Conventionally built for graphics applications, GPUs have various hardware features that can boost performance if carefully managed; however, GPU hardware designers may not be sufficiently informed about scientific workloads to evaluate specialized hardware features. On the other hand, since GPU architectures embrace massive parallelism and limited L1 storage per thread, legacy codes must be redesigned in order to be ported to GPUs. Even codes for earlier GPU generations may have to be recoded in order to fully exploit new GPU architectures. As a result, an increasing effort has been made in codesigning the application and the hardware.

In a typical codesign effort, a set of benchmarks is proposed by application developers and is then manually studied by hardware designers in order to understand the potential. However, such a process is labor-intensive and time-consuming. In addition, several factors challenge system designers' endeavors to explore the design space. First, the number of hardware configurations is exploding as the complexity of the hardware increases. Second, the solution has to meet several design constraints such as area and power. Third, benchmarks are often provided in a specific implementation, yet one often needs to attempt tens of transformations in order to fully understand the performance potential of a specific hardware configuration. Fourth, evaluating the performance of a particular implementation on a future hardware may take significant time using simulators. Fifth, the current performance tools (e.g., simulators, hardware models, profilers) investigate either hardware or applications in a separate manner, treating the other as a black box and therefore, offer limited insights for codesign.

To efficiently explore the design space and provide first-order insights, we propose SESH, a model-driven framework that automatically searches the design space by simultaneously exploring *prospective* application and hardware implementations and evaluate *potential* software-hardware interactions. SESH recommends the optimal combination of application optimizations and hardware implementations according to user-defined objectives with respect to performance, area, and power. The technical contributions of the SESH framework are as follows.

1) It evaluates various software optimization effects and hardware configurations using decoupled workload models and hardware models.
2) It integrates GPU's performance, area, and power models into a single framework.
3) It automatically proposes optimal hardware configurations given multi facet metrics in aspects of performance, area, and power.

We evaluate our work using a set of representative scientific workloads. A large design space is explored that considers both application transformations and hardware configurations. We evaluate potential solutions using various metrics including performance, area efficiency, and energy consumption. Then, we summarize the overall insights gained from such space exploration.

The paper is organized as follows. In Section II, we provide an overview of our work. In Section III, we introduce the integrated hardware models for power and area. Section IV describes the space exploration process. Evaluation methodology and results are described in Sections V and VI, respectively. After the related work is discussed in Section VII, we conclude.

II. Overview and Background

The SESH framework is a codesign tool for GPU system designers and performance engineers. It recommends the optimal combination of hardware configurations and application implementations. Different from existing performance models or architecture simulators, SESH considers how applications may transform and adapt to potential hardware architectures.

A. Overall Framework

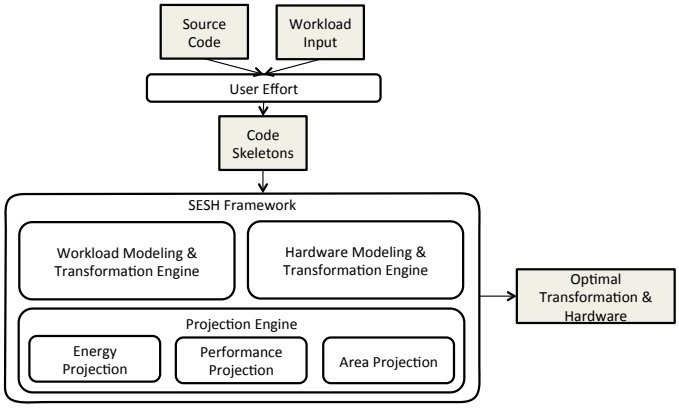

Fig. 1. Framework Overview.

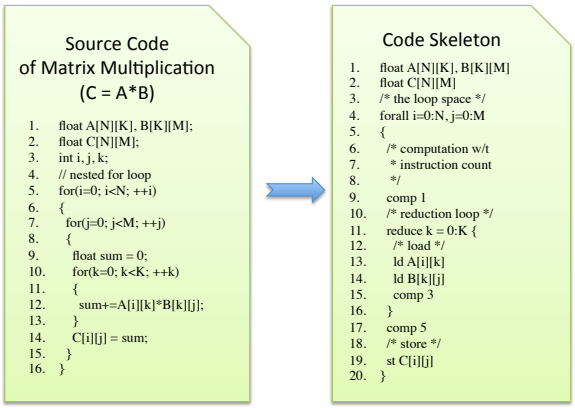

Fig. 2. A pedagogical example of a code skeleton in the case of matrix multiplication. A code skeleton is used as the input to our framework.

As Figure 1 shows, the major components of the SESH framework include (i) a workload modeling and transformation engine, (ii) a hardware modeling and transformation engine, and (iii) a projection engine. Using the framework involves the following work flow:

1) The user abstracts high level characteristics of the source code into a *code skeleton* that summarizes control flows, potential parallelism, instruction mix, and data access patterns. An example code skeleton is shown in Figure 2. This step can be automated by SKOPE [1], and the user can amend the output code skeleton with additional notions such as `for_all` or `reduction`.
2) Given the code skeleton and the specified power and area constraints, SESH explores the design space by automatically proposing potential application transformations and hardware configurations.
3) SESH projects the energy consumption and execution time for each combination of transformations and hardware configurations, and recommends the best solution according to user-specified metrics, without manual coding or lengthy simulations. Such metrics can be a combination of power, area, and performance. For example, one metric can be "given an area budget, what would be the most power efficient hardware considering potential transformations?".

The SESH framework is built on top of existing performance modeling frameworks. We integrated GROPHECY [2] as the GPU workload modeling and transformation engine. We also adopted area and power models from previous work on projection engine. Below we provide a brief description about them.

B. GROPHECY-Based Code Transformation Engine

GROPHECY [2], a GPU code transformation framework, has been proposed to explore various transformations and to estimate the GPU performance of a CPU kernel. Provided with a code skeleton, GROPHECY is able to transform the code skeleton to mimic various optimization strategies. Transformations explored include spatial and temporal loop tiling, loop fusion, unrolling, shared memory optimizations, and memory request coalescing. GROPHECY then analytically computes the characteristics of each transformation. The resulting characteristics are used as inputs to a GPU performance model to project the performance of the corresponding transformation. The best achievable performance and the transformations necessary to reach that performance are then projected.

GROPHECY, however, relies on the user to specify a particular hardware configuration. It does not explore the hardware design space to study how hardware configurations would affect the performance or efficiency of various code transformations. In this work, we extend GROPHECY with parameterized power and area models so that one can integrate it into a larger framework that explores hardware configurations together with code transformations.

C. Hardware Models

We utilize the performance model from the work of Sim et al. [3]. We make it tunable to reflect different GPU architecture specifications. The tunable parameters are *Register file entry count*, *SIMD width*, *SFU width*, *L1D/SHMEM size* and *LLC cache size*. The model takes several workload characteristics as its inputs, including the instruction mix and the number of memory operations.

We utilize the work by Lim et al. [4] for chip-level area and power models. They model GPU power based on McPAT [5] and an energy introspection interface (EI) [6] and integrate the power-modeling functionality in MacSim [7], a trace-driven and cycle-level GPU simulator. McPAT enables users to configure a microarchitecture by rearranging circuit-level models. EI creates pseudo components that are linked to McPAT's circuit-level models and utilizes access counters to calculate the power consumption of each component. We use this model to estimate the power value for the baseline

architecture configuration. Then we adopt simple heuristics to estimate the power consumption for different hardware configurations.

III. Exploratory, Multi facet Hardware Model

In order to explore the hardware design space and evaluate tradeoffs, the SESH framework integrates performance, power and area models, all parameterized and tunable according to the hardware configuration. In this section, we first describe the process to prepare the reference models about the NVIDIA GTX 580 architecture. These models include power and area models for chip. We then integrate these models and parameterize them so that they can reflect changes in hardware configurations.

A. The Reference Model for Chip-Level Power

To get reference values for chip-level power consumption, we use the detailed power simulator in Section II-C. As Hong and Kim [8] showed, the GPU power consumption is not significantly affected by the dynamic power consumption except the DRAM power values. Furthermore, the static power is often the dominant factor for on-chip power consumption. Hence, to get the first-order approximation of the GPU power model, we focus on the static power only.

To estimate the static power for a baseline architecture of NVIDIA GTX 580, we collect power statistics from a detailed power simulator [4]. The Sepia benchmark [9] is used as an example input for the simulation. Note that the choice of the benchmark does not affect significantly the value of static power. In Sepia's on-chip power consumption, the static power consumes 106.0 W while the dynamic power consumes 50.0 W. The DRAM power is 52.0 W in Sepia, although it is usually more than 90.0 W in other benchmarks. As a result, the dynamic power accounts for 24.0% of Sepia's the total power consumption of the chip and DRAM. Taking the dynamic power into account will be included in our future work.

The variation in power consumption caused by thermal changes is not modeled for the purpose of simplicity. As measured in [8], we assume a constant temperature of 340 K (67 °C) in the power model, considering that this operation-time temperature is higher than the code-state temperature of 57 °C [8].

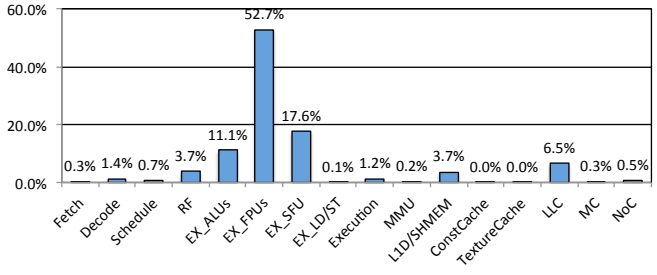

Fig. 3. Power consumption for non-DRAM components.

Figure 3 shows how much total power is consumed by each component according our model. The GPU's on-chip power is decomposed into two categories; SM-private components and SM-shared components. The total on-chip power consumption is 156.0 W. SM-private components accounts for 93 % (144.7 W) of the overall power, and shared components between SMs account for 7% (11.4 W) of the overall power. From all the SM-private components we model EX_ALUs and EX_FPUs consume the most power (52.7% and 11.1%, respectively). SFU (17.6%), LLC (6.5%), and RF (3.7%) also account for large portions of the overall power consumption.

B. The Reference Model for Chip-Level Area

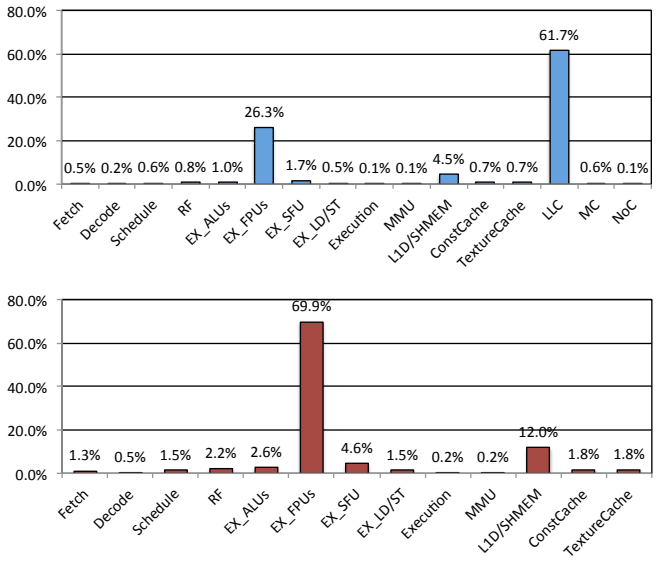

Fig. 4. Area consumption for all non-DRAM components(top) and for SM-private components(bottom).

For the area model, we utilize the area outcome from energy introspection integrated with MacSim [4]. The energy introspection interface in MacSim utilizes area size to calculate power. It also estimates area sizes for different hardware configurations. we use the area outcomes that are based on NVIDIA GTX 580.

Figure 4 (top) shows the area breakdown of GTX 580 based on our area model. The total area consumption of the chip is 3588.0 mm^2. LLC (61.7%) accounts for the majority of the chip area. Figure 4 (bottom) shows the breakdown of the area for SM-private components. The main processing part, 32 SPs (EX_ALUs and EX_FPUs), occupy the largest portion of the area (69.9% and 2.6%, respectively). The L1D / SHMEM is the second largest module (12.0%). SFU (4.6%) and RF (2.2%) also account for a large portion of area consumption.

C. Integrated, Tunable Hardware Model

$$Area_{target} = Area_{baseline} \times \frac{knob_{target}}{knob_{baseline}} \quad (1)$$

$$Power_{target} = Power_{baseline} \times \frac{knob_{target}}{knob_{baseline}} \quad (2)$$

To estimate how changes in hardware configurations affect the overall power consumption, we employ a heuristic that the per-component power consumption and area scales linearly

TABLE I. HARDWARE COMPONENTS THAT ARE MODELED AS TUNABLE KNOBS.

Stage	KNOB	Default(NVIDIA GTX 580)
Per-SM		
RF	Register file entry count	32,768 / SM
ALU	SIMD width	32 / SM
FPU	SIMD width	32 / SM
SFU	SFU width	4 / SM
L1D/SHMEM	L1D size + SHMEM size	64KB / SM
Shared		
LLC	L2 Cache size	768KB

TABLE II. HARDWARE COMPONENTS THAT ARE MODELED WITH CONSTANT AREA AND POWER CONSUMPTION.

Category	Stage
Per-SM(w/ fixed number of SMs)	Fetch, Decode, Schedule, Execution(except ALU, FPU, SFU), MMU, Const$, Tex$
Shared	1 MemCon, 1 NoC, 1 DRAM

with the size and the number of components (e.g., doubling the shared memory size would double its power consumption and also area). Given a target hardware's configuration, we can compute the per-component area and power according to Equations (1) and (2), respectively, where $knob$ refers to the size or number of the component in the corresponding architecture. The baseline data is collected as described in Sections III-A and III-B. The per-component metrics are then aggregated to project the system-level area and power consumption.

According to our analysis in Figures 3 and 4, the major components consuming power and area include the register files, ALUs, FPUs, SFUs, L1 cache size, and the last level cache size. The quantities of these components become tunable variables, or knobs, in the integrated model. Table I lists the knobs and the value of $knob_{baseline}$ in Equations (1) and (2). The area and power consumption of other components are approximated as constant values obtained from modeling NVIDIA GTX 580. These components are summarized in Table II.

D. DRAM Power Model

DRAM power depends on memory access patterns; the number of DRAM row buffer hits/misses can affect power consumption noticeably. However, these numbers can be obtained only from the detailed simulation, which often takes several hours for each configuration (100s of software optimizations $\times$ 10s of hardware configurations easily create 1000s of different configurations). To mitigate the overhead, we use a simple empirical approach to compute the first-order estimation of DRAM power consumption values.

$$P_{DRAM} = MaxDynP \times \frac{Trans\_Intensity}{Max\_Trans\_Intensity} + StatP \quad (3)$$

The total DRAM power (P_{DRAM}) is computed by adding up the static power ($StatP$) and dynamic power [8]. The dynamic power is computed as a fraction of the maximum dynamic power ($MaxDynP$), which can only be reached in the theoretical condition where every instruction generates a DRAM transaction. The number of DRAM transactions per second on each SM is referred to as *DRAM transaction intensity*, whose value is $Max\_Trans\_Intensity$ under the aforementioned theoretical condition.

$$Trans\_Intensity = \frac{\#DRAM\_Accesses}{Exec\_time} \quad (4)$$

In this work, the actual DRAM transaction intensity, $Trans\_Intensity$, is approximated by Equation (4). The total number of DRAM transactions per SM ($\#DRAM\_Accesses$) and the execution time in seconds ($Exec\_time$) are estimated values given by the performance model.

In order to construct Equation (3) as a function of the workload characteristics, the values of $StatP$ and $\frac{MaxDynP}{MAX\_Trans\_Intensity}$ need to be obtained as constant coefficients. We therefore use the power simulator to obtain the DRAM power for SVM and Sepia in the Merge benchmarks [9] and solve for the values of these two coefficients. Equation (5) represents the resulting DRAM model.

$$P_{DRAM} = \alpha \times Trans\_Intensity + \beta \quad (5)$$

where α = 1.6 $\times$ 10^{-6} and β = 24.4

IV. SPACE EXPLORATION

Application transformations and hardware configurations pose a design space that is daunting to explore. First, they are inter-related; different hardware configurations may prefer different application transformations. Second, there are a gigantic number of options. In our current framework, we explore each of them independently and then calculate which combination yields the desired solution. Note that this process is made possible largely because of the fast evaluation enabled by modeling.

The application transformations explored include spatial and temporal loop tiling, unrolling, shared memory optimizations, and memory request coalescing. The hardware configurations explored include SIMD width and shared memory size, which play significant roles in performance, area, and power. We plan to explore more dimensions in our future work.

To compare different solutions, we utilize multiple objective functions that represent different design goals. Those objective functions include the followings.

1) Shortest execution time
2) Minimal energy consumption
3) Largest performance per area

V. METHODOLOGY

A. Workloads

The benchmarks used for our evaluation and their key properties are summarized in Table III. HotSpot and SRAD are applications from the Rodinia benchmark suite [10]. Stassuij is extracted from a DOE INCITE application performing Monte Carlo calculations to study light nuclei [11], [12]. It has two kernels: IspinEx and SpinFlap. We separately evaluate each kernel of Stassuij and also evaluate both kernels together. The

TABLE III. WORKLOAD PROPERTIES

Benchmark	Key Properties	Input Size
HotSpot	Structured grid. Iterative, self-dependent kernels. A deep dependency chain among dependent kernels	1024 X 1024
SRAD	Structured grid. Data dependency involves multiple arrays; each points to different producer iterations	4096 X 4096
IspinEx	Sparse linear algebra, A X B	A : 132 X 132, B : 132 X 2048
SpinFlap	Irregular data exchange similar to spectral methods	132 X 2048
Stassuij	Nested loops. Dependent parallel loops with different shapes. Dependency involves indirectly accessed arrays	-

sizes of matrices in Stassuij are according to real input data. To reduce the space of code transformations to explore, for each benchmark we set a fixed thread block size large enough to saturate wide SIMD.

HotSpot: HotSpot is an ordinary differential equation solver used in simulating microarchitecture temperature. It has a stencil computation kernel with structured grid. Kernels are executed iteratively, and each iteration consumes a neighborhood of array elements. As a result, each iteration depends on the previous one. Programmers can utilize shared memory(ShM) by caching neighborhood data for inter-thread data sharing. Folding, which assigns multiple tasks to one thread, improves data reuse by allowing a thread to process more neighborhood-gathering tasks. Fusing loop partitions across several iterations can be applied to achieve better locality and reduce global halo exchanges. We also provide a hint that indicates only one of the arrays used in double buffering is the necessary output for the fused kernel. In our experiments, we fuse two dependent iterations and use a 16×16 partition for the consumer loop. The thread block size is set to 16×16.

SRAD: SRAD performs spectral removal anisotropic diffusion to an image. It has two kernels: the first generates diffusion coefficients and the second updates the image. We use a 16×16 thread block size and indicate the output array that needs to be committed.

IspinEx: IspinEx is a sparse matrix multiplication kernel which multiplies a 132×132 sparse matrix of real numbers with a 132×2048 dense matrix of complex numbers. We provide a hint that the average number of nonzero elements in one row of the sparse matrix is 14 in order to estimate the overall workload size. Because the numbers of elements associated with different rows may vary. we force a thread to process all elements in columns to balance the workload among threads. We treat the real part and imaginal part of the complex number as individual numbers and employ a 1×64 thread block size in our evaluation. Due to irregularity in sparse data accesses, we provide a hint about the average degree of coalescing, which is obtained from offline characterization of the input data.

SpinFlap: SpinFlap exchanges matrix elements in groups of four. Each group is scattered in a butterfly pattern in the same row, similar to spectral methods. Which columns are to be grouped together is determined by values in another array. SpinFlap is a memory-bounded kernel. By utilizing shared memory, programmers can expect performance improvement. There is data reuse by multiple threads on the matrix, which are used for indirect indices for other matrices. The performance can also be improved by folding. Performance is highly dependent on the degree of coalescing, and it varies according to values of indirect indices. To assess the degree of coalescing, we profiled the average data stride of indirect indices and provide this hint to the framework. We assume a 12×16 thread block size with no folding.

Stassuij(Fused): Fusion increases the reuse in shared memory. But since data dependency caused by indirect indices in SpinFlap requires IspinEx to be partitioned accordingly, the loop index in IspinEx now becomes a value pointed by indirect accesses in the fused kernel, introducing irregular strides that can become un-coalesced memory accesses. We assume a thread block size of $16 \times 4 \times 2$ and provide a hint that indicates the output array.

B. Evaluation Metric

To study how application performance is affected by code transformations and architectural changes, we utilize metrics from previous work [3] to understand the potential optimization benefits and the performance bottlenecks.

1) B_serial : Benefits of removing serialization effects such as synchronization and resource contention.
2) B_itilp : Benefits of increasing inter-thread instruction-level parallelism (ITILP). ITILP represents global ILP (ILP among warps).
3) B_memlp : Benefits of increasing memory-level parallelism (MLP)
4) B_fp : Benefits of improving computing efficiency. Computing efficiency represents the ratio of the floating point instructions over the total instructions.

VI. EVALUATION

While we have explored a design space with both code transformations and hardware configurations, we present our results according to SIMD widths and shared memory sizes in order to shed more light on hardware designs. We model 64 transformations for HotSpot, 64, 128 transformations for two kernels of SRAD, 576, 1728 and 2816 transformations for IspinEx, SpinFlap and Stassuij(Fused) respectively.

A. SIMD Width

We evaluate different SIMD widths from 16 to 128. Figure 5 represents the execution time, energy consumption, possible benefits and performance per area for optimal transformation for HotSpot with increasing SIMD width. The performance-optimal SIMD width is 64 and the optimal SIMD width for minimal energy consumption and maximal performance per area is 32, which is the same as for NVIDIA GTX 580. The reason is that the increased inefficiency in power and area is bigger than the benefit of shorter execution time, even though minimal execution time helps reduce the energy consumption in general. Performance increases from 16 to 64 but decreases from 64 to 128. The application becomes more memory bound with increased SIMD width, and the benefit of less computation time becomes smaller, which we can see from increasing B_memlp.

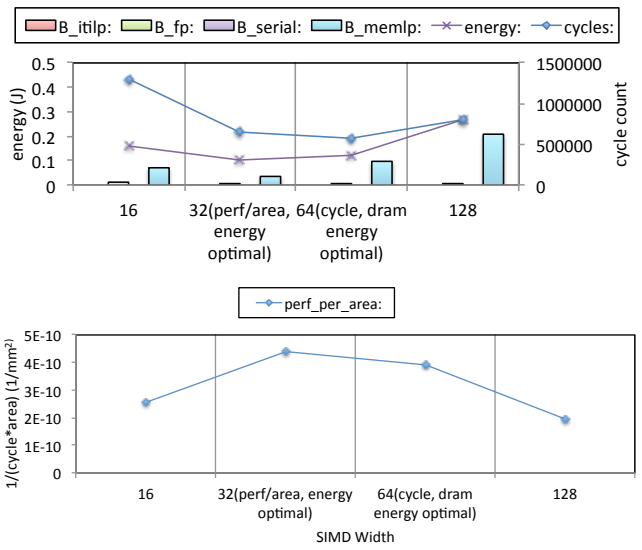

Fig. 5. Execution time, energy consumption, possible benefits and performance per area for optimal transformation for HotSpot on increasing SIMD width.

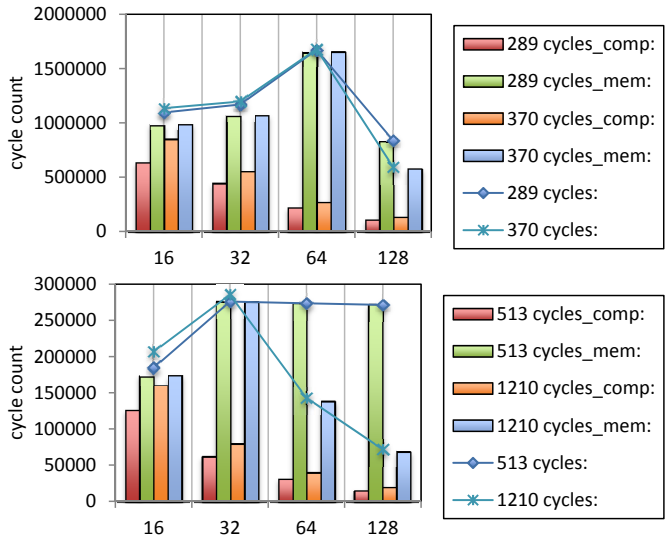

Fig. 6. Comparison between optimal transformation with increasing SIMD width for IspinEx(top) and SpinFlap(bottom).

For HotSpot, SRAD, and Stassuij, the optimal transformation remains the same regardless of SIMD width and objective function. However, depending on SIMD width, optimal transformation changes from 289 (16, 32, 64) to 370 (128) for IspinEx and from 513 (16, 32) to 1210 (64, 128) for SpinFlap. Figures 6 compares the optimal transformation with increasing SIMD width for IspinEx and SpinFlap. The optimal transformation on a narrow SIMD width is selected because it incurs less computation, even though it incurs more memory traffic due to un-coalesced access on a wide SIMD width than optimal transformation on a wide SIMD width. However, the application becomes more memory bound with increased SIMD width, therefore reducing the benefit of less computation.

The transformation ID we use in this paper can be different

```
1  def ispinex() {
2    ...
3    forall j=0:nt, ir=0:ns*2 {
4      ...
5      reduce (float, +) n = 0:avg_j_ntdt {
6        ...
7        ld cr[njp][ir]  // Different on 289 & 370
8        ...
9      }
10     ...
11   }
12 }
```

Fig. 7. Comparison of transformations 289 & 370 for IspinEx.

TABLE IV. OPTIMAL SIMD WIDTH REGARDING MINIMAL EXECUTION TIME, MINIMAL ENERGY CONSUMPTION AND MAXIMAL PERFORMANCE PER AREA.

Benchmark	Performance	Energy	Perf/Area
HotSpot	64	32	32
SRAD(first)	32	32	32
SRAD(second)	32	16	16
IspinEx	128	16	16
SpinFlap	128	16	128
Stassuij	32	16	32

depending on the degree of loop tiling, loop fusion, unrolling, shared memory optimizations, and memory request coalescing. Figure 7 compares transformations 289 and 370 for IspinEx. Those two have same code structure; the only difference is the decision of which loads to be cached or not. Transformation 370 utilizes shared memory for the load *ld cr[njp][ir]*, while transformation 289 doesn't. The difference between transformations 513 and 1210 for SpinFlap is also which loads utilize shared memory or not.

The optimal SIMD width is different depending on workload objective functions. Table IV represents optimal SIMD width for HotSpot, SRAD, IspinEx, SpinFlap and Stassuij regarding minimal execution time, minimal energy consumption and maximal performance per area. We also find strong correlation between minimal energy consumption and largest performance per area. Except for SpinFlap and Stassuij, the optimal SIMD width for minimal energy consumption and the one for largest performance per area are the same.

Considering source code transformation or not changes the optimal SIMD width for SpinFlap. Table V compares the optimal SIMD width for SpinFlap when using optimal transformation on NVIDIA GTX 580 or using optimal transformation on each SIMD width. The optimal SIMD width for performance and performance per area is 128 and the energy optimal SIMD width is 16 when we consider source code transformation. However, 16 is the optimal SIMD width for all objective functions when we do not consider source code transformation and use the optimal transformation on NVIDIA GTX 580 instead. Figure 8 compares the execution time, energy consumption and possible benefits for SpinFlap

TABLE V. OPTIMAL SIMD WIDTH FOR SPINFLAP USING FIXED TRANSFORMATION OR OPTIMAL TRANSFORMATION ON EACH SIMD WIDTH.

Benchmark	Performance	Energy	Perf/Area
Fixed	16	16	16
Variable	128	16	128

with increasing SIMD width considering source code transformation or not.

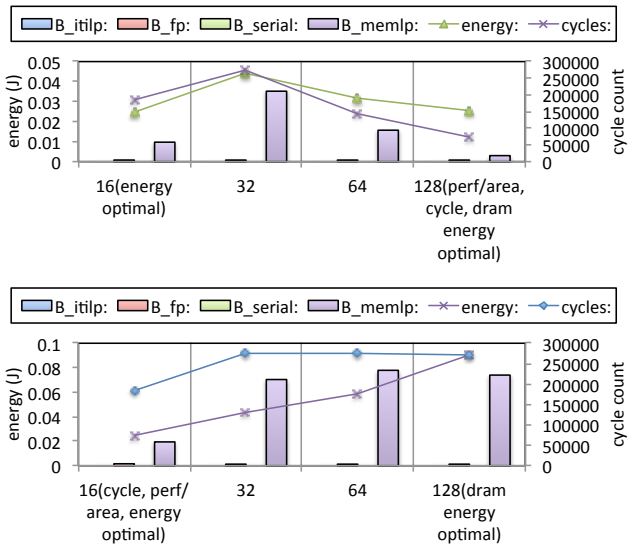

Fig. 8. Execution time, energy consumption and possible benefits for optimal transformation for SpinFlap with increasing SIMD width: (top) considering source code transformation; (bottom) using fixed transformation.

In summary, increasing SIMD width helps performance. But the benefit of large SIMD width degrades because of increased inefficiency in power and area, and the application becomes more memory bound with increased SIMD width. The optimal SIMD width is depends on workload objective functions, with strong correlation between minimal energy consumption and largest performance per area. The optimal transformation changes depend on SIMD width for IspinEx and SpinFlap. Considering source code transformation or not changes the optimal SIMD width for SpinFlap.

B. Shared Memory Size

GPU's shared memory is a software-managed L1 storage on an SM. A larger shared memory would enable new transformations with more aggressive caching. We evaluate shared memory sizes from 16 KB to 128 KB. The values of other parameters remain constant.

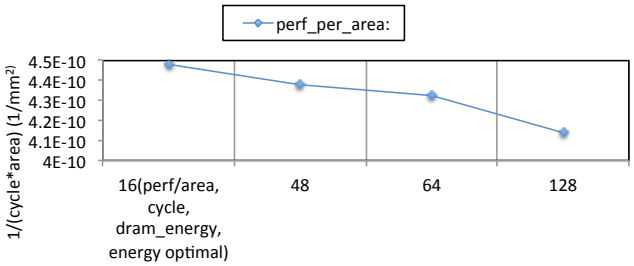

Fig. 9. Performance per area for optimal transformation for HotSpot with increasing shared memory size.

The optimal transformation for HotSpot, SRAD and Stassuij and their performance remain the same regardless of shared memory size. The reason is that shared memory usage

TABLE VI. NUMBER OF TRANSFORMATIONS AVAILABLE FOR ISPINEX, SPINFLAP, AND STASSUIJ DEPENDING ON SHARED MEMORY SIZE.

Benchmark	16 KB	48 KB	64 KB	128 KB
IspinEx	120	192	192	288
SpinFlap	432	1304	1304	1728
Stassuij	2240	2240	2240	2816

per SM for optimal transformations for these applications is already less than 16 KB. Figure 9 represents the performance per area for optimal transformation for HotSpot on increasing shared memory size.

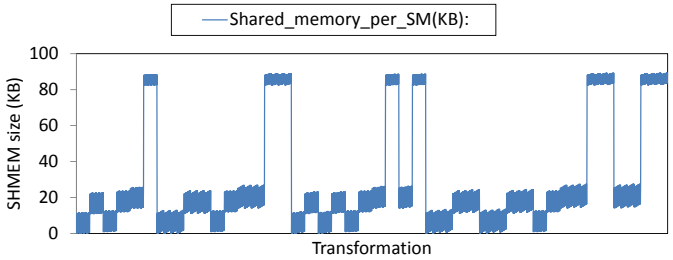

Fig. 10. Shared memory requirement for transformations for Stassuij.

For HotSpot and SRAD, none of the transformations are disabled even when the shared memory size per SM is reduced to 16 KB. Such is not the case for IspinEx, SpinFlap and Stassuij. Figure 10 presents the shared memory requirement for all transformations for Stassuij. Some transformation require less than 20 KB of shared memory, but other transformations require more than 80 KB of shared memory. Therefore the number of valid transformations is different depending on the shared memory size. Table VI presents the number of transformations available for IspinEx, SpinFlap and Stassuij depending on shared memory size.

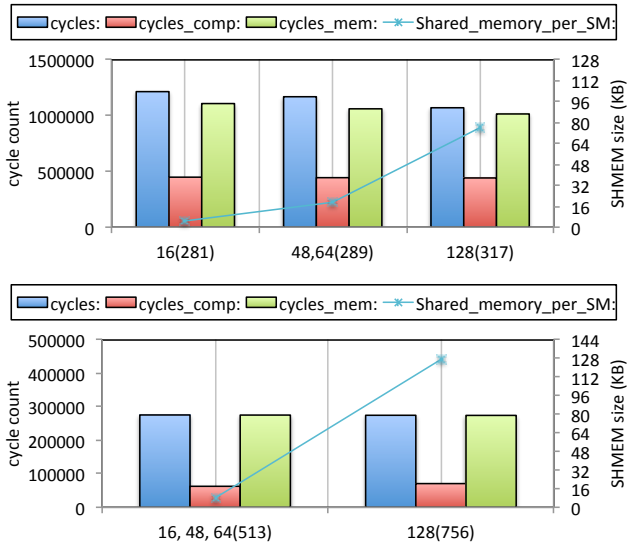

Fig. 11. Comparison between the optimal transformations when increasing shared memory size for IspinEx(top) and SpinFlap(bottom).

The optimal transformation for Stassuij remains the same regardless of shared memory size since shared memory usage

per SM for the optimal transformation is less than 9 KB. However, the optimal transformation changes depending on shared memory size for IspinEx and SpinFlap. New transformations become available as we increase the shared memory size. The optimal transformation changes from 281 (16 KB) to 289 (48, 64 KB), 317 (128 KB) for IspinEx, and it changes from 513 (16, 48, 64 KB) to 756 (128 KB) for SpinFlap. Figure 11 compares the optimal transformation with increasing shared memory size for IspinEx and SpinFlap. The difference between those transformations is which loads utilize shared memory.

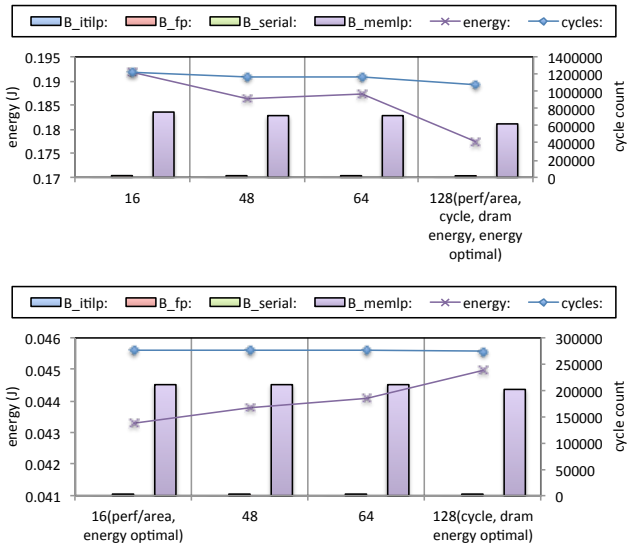

Fig. 12. Execution time, energy consumption and possible benefits for optimal transformation on increasing shared memory size for IspinEx(top) and SpinFlap(bottom).

Figure 12 represents the execution time and energy consumption for the optimal transformations of IspinEx and SpinFlap with increasing shared memory size. When we consider code transformations, the optimal shared memory size for all objective function for IspinEx is 128 KB. Without considering transformations however, the optimal shared memory size is 48 KB. For SpinFlap, the optimal shared memory size for energy and performance per area is 16 KB and the performance-optimal shared memory size is 128 KB when we consider source code transformation. Without considering transformations, the optimal shared memory size remains 16 KB for all objective functions.

In summary, shared memory sizes determine the number of possible transformations in terms of how the shared memory is used. For IspinEx, SpinFlap and Stassuij, some transformations are disabled because of limitation of shared memory size. These applications prefer either small shared memory or very large shared memory, as we can see in Figure 10. For IspinEx and SpinFlap, the optimal transformation changes depending on shared memory size since new transformations become available with increased shared memory size. Considering source code transformations or not changes the optimal shared memory size for IspinEx and SpinFlap.

C. Discussion

The lessons learned from our model-driven space exploration is summarized below.

1) For a given hardware, the code transformation with minimal execution time often leads to minimal energy consumption as well. This can be observed from Figure 13, which represents execution time and energy consumption of possible transformations of each application on NVIDIA GTX 580.
2) The optimal hardware configuration depends on the objective function. In general, performance increases with more resources (wider SIMD width or bigger shared memory size). However, the performance benefit of more resources may be outweighed by the cost of more resources in terms of energy or area. Therefore, the SIMD width and shared memory size that are optimal for energy consumption and performance per area are smaller than those for performance. We also observe that the SIMD width and shared memory size that minimize energy consumption also maximize performance per area.
3) The optimal transformation can differ across hardware configurations. The variation in hardware configuration has two effects on code transformations: it shifts the performance bottleneck, or it enables/disables potential transformations because of resource availability. In the examples of IspinEx and SpinFlap, a computation-intensive transformation becomes memory-bound with wide SIMD; a new transformation that requires large L1 storage is enabled when the shared memory size increases.
4) The optimal hardware configuration varies according to whether code transformations are considered. For example, when searching for the performance-optimal SIMD width for SpinFlap, the legacy implementation would suggest a SIMD width of 16, while the performance-optimal SIMD width is 128 if transformations are considered and would perform 2.6 $\times$ better. The optimal shared memory size would also change for IspinEx and SpinFlap by taking transformations into account.
5) In order to gain performance, it is generally better to increase the SIMD width rather than the shared memory size. Larger SIMD width increases performance at the expense of area and power consumption. Larger shared memory size does not have a significant impact on performance until it can accommodate a larger working set, therefore enabling a new transformation; however, we found that a shared memory size of 48 KB is already able to accommodate a reasonably sized working set in most cases. This coincides with GPU's hardware design trends from Tesla [13] to Fermi [14] and Kepler [15]. Moreover, we found that energy-optimal shared memory size for all evaluated benchmarks is less than 48 KB, which is the value for current Fermi architecture.

Our work can be improved to enable broader space exploration in less time. A main challenge in space exploration is the large number of possible transformations and architectural parameters. Instead of brute-force exploration, we plan to

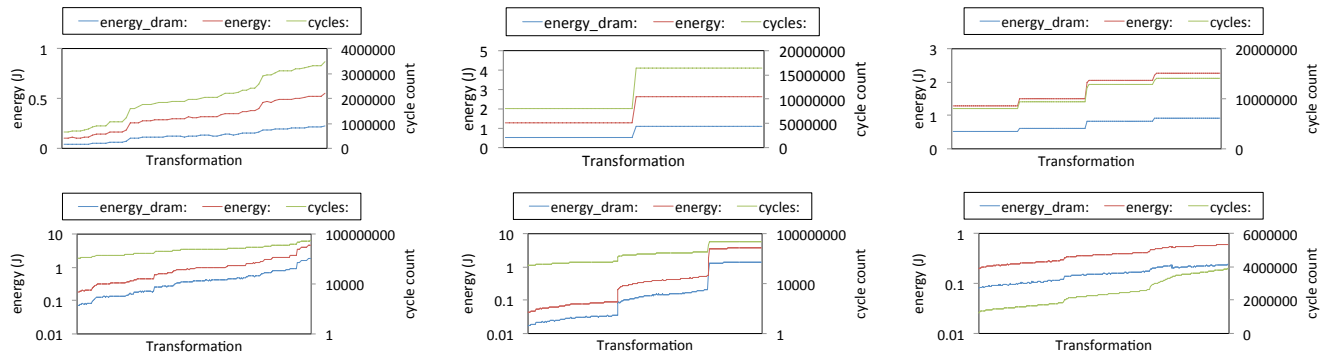

Fig. 13. Execution time and energy consumption of possible transformations of each application on NVIDIA GTX 580. From top to bottom, HotSpot, SRAD (first) SRAD (second), IspinEx, SpinFlap and Stassuij (Fused).

build a feedback mechanism to probe the space judiciously. For example, if an application is memory bound, the SESH framework can try transformations that result in fewer memory operations but more computation, or hardware configurations with large shared memory and smaller SIMD width.

VII. Related work

Multiple software frameworks are proposed to help GPU programming [16], [17]. Workload characterization studies [18], [19] and parallel programming models including PRAM, BSP, CTA and LogP are also relevant to our work [20], [21]. These techniques do not explore the hardware design space.

To the best of our knowledge, there has been no previous work to study the relationships between GPU code transformation and power reduction. Valluri et al. [22] performed quantitative study of the effect of the optimizations by DEC Alpha's *cc* compiler. Brandolese et al. [23] explored source code transformation in terms of energy consumption using the SimpleScalar simulator.

Modeling power consumption of CPUs has been widely studied. Joseph's technique relies on performance counters [24]. Bellosa et al. [25] also used performance counters in order to determine the energy consumption and estimate the temperature for a dynamic thermal management. Wu et al. [26] proposed utilizing phasic behavior of programs in order to build a linear system of equations for component unit power estimation. Peddersen and Parameswaran [27] proposed a processor that estimates its own power/energy consumption at runtime. CAMP [28] used the linear regression model to estimate activity factor and power; it provides insights that relate microarchitectural statistics to activity factor and power. Jacobson et al. [29] built various levels of abstract models and proposed a systematic way to find a utilization metric for estimating power numbers and a scaling method to evaluate new microarchitecture. Czechowski and Vuduc [30] studied relationship between architectural features and algorithm characteristics. They proposed a modeling framework that can be used for tradeoff analysis of performance and power.

While architectural studies and performance improvements with GPUs have been explored widely, power modeling of GPUs has received little attention. A few works use functional simulator for power modeling. Wang [31] extended GPGPUSim with Wattch and Orion to compute GPU power. PowerRed [32], a modular architectural power estimation framework, combined both analytical and empirical models; they also modeled interconnect power dissipation by employing area cost. A few GPU power modeling works use a statistical linear regression method using empirical data. Ma et al. [33] dynamically predicted the runtime power of NVIDIA GeForce 8800 GT using recorded power data and a trained statistical model. Nagasaka et al. [34] used the linear regression method by collecting the information about the application from performance counters. Tree-based random forest method was used on the works by Chen et al. [35] and by Zhang et al. [36]. Since those works are based on empirical data obtained from existing hardware, they do not provide insights in terms of space exploration.

Simulators have been widely used to search the hardware design space. Generic algorithms and regression have been proposed to reduce the search space by learning from a relatively small number of simulations [37]. Our work extends their work by considering code transformations, integrating area and power estimations, and employing models instead of simulations. Nevertheless, their learning-based approach is complementary to our approach and may help SESH framework prune the space as well.

VIII. Conclusions

We propose the SESH framework, a model-driven framework that automatically searches the design space by simultaneously exploring *prospective* application and hardware implementations and evaluate *potential* software-hardware interactions. It recommends the optimal combination of application optimizations and hardware implementations according to user-defined objectives with respect to performance, area, and power. We explored the GPU hardware design space with different SIMD widths and shared memory sizes, and we evaluated each design point using four benchmarks, each with hundreds of transformations. The evaluation criteria include performance, energy consumption, and performance per area. Several codesign lessons were learned from the framework, and our findings point to the importance of considering code transformations in designing future GPU hardware.

REFERENCES

[1] J. Meng, X. Wu, V. A. Morozov, V. Vishwanath, K. Kumaran, V. Taylor, and C.-W. Lee, "SKOPE: A Framework for Modeling and Exploring Workload Behavior," Argonne National Laboratory, Tech. Rep., 2012.

[2] J. Meng, V. Morozov, K. Kumaran, V. Vishwanath, and T. Uram, "GROPHECY: GPU Performance Projection from CPU Code Skeletons," in *SC'11*.

[3] J. W. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "GPUPerf: A Performance Analysis Framework for Identifying Performance Benefits in GPGPU Applications," in *PPoPP '12*.

[4] J. Lim, N. Lakshminarayana, H. Kim, W. Song, S. Yalamanchili, and W. Sung, "Power Modeling for GPU Architecture using McPAT," Georgia Institute of Technology, Tech. Rep., 2013.

[5] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO 42*.

[6] W. J. Song, M. Cho, S. Yalamanchili, S. Mukhopadhyay, and A. F. Rodrigues, "Energy introspector: Simulation infrastructure for power, temperature, and reliability modeling in manycore processors," in *SRC TECHCHON '11*.

[7] "MacSim," http://code.google.com/p/macsim/.

[8] S. Hong and H. Kim, "IPP: An Integrated GPU Power and Performance Model that Predicts Optimal Number of Active Cores to Save Energy at Static Time," in *ISCA '10*.

[9] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: a programming model for heterogeneous multi-core systems," in *ASPLOS XIII*.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using CUDA," *Journal of Parallel and Distributed Computing*, 2008.

[11] S. C. Pieper and R. B. Wiringa, "Quantum Monte Carlo calculations of light nuclei," in *Annu. Rev. Nucl. Part. Sci. 51, 53*, 2001.

[12] S. C. Pieper, K. Varga, and R. B. Wiringa, "Quantum Monte Carlo calculations of A=9,10 nuclei," in *Phys. Rev. C 66, 044310-1:14*, 2002.

[13] N. Corporation, "GeForce GTX 280 specifications," 2008. [Online]. Available: http://www.nvidia.com/object/product_geforce_gtx_280_us.html

[14] NVIDIA, "Fermi: Nvidia's next generation cuda compute architecture," http://www.nvidia.com/fermi.

[15] "NVIDIA's next generation CUDA compute architecture: Kepler GK110," *NVIDIA Corporation*, 2012.

[16] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic CPU-GPU communication management and optimization," in *PLDI '11*.

[17] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August, "Dynamically managed data for cpu-gpu architectures," in *CGO '12*.

[18] K. Spafford and J. Vetter, "Aspen: A domain specific language for performance modeling," in *SC '12*.

[19] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM - A Direct Path to Dependable Software*, 2009.

[20] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, 1990.

[21] R. M. Karp and V. Ramachandran, "A survey of parallel algorithms for shared-memory machines," EECS Department, University of California, Berkeley, Tech. Rep., 1988.

[22] M. Valluri and L. John, "Is Compiling for Performance == Compiling for Power," in *INTERACT-5*.

[23] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto, "The Impact of Source Code Transformations on Software Power and Energy Consumption," *Journal of Circuits, Systems, and Computers*, 2002.

[24] R. Joseph and M. Martonosi, "Run-time power estimation in high performance microprocessors," in *ISLPED '01*.

[25] F. Bellosa, S. Kellner, M. Waitz, and A. Weissel, "Event-driven energy accounting for dynamic thermal management," in *COLP '03*.

[26] W. Wu, L. Jin, J. Yang, P. Liu, and S.-D. Tan, "A systematic method for functional unit power estimation in microprocessors," in *DAC '06*.

[27] J. Peddersen and S. Parameswaran, "CLIPPER: Counter-based Low Impact Processor Power Estimation at Run-time," in *ASP-DAC '07*.

[28] M. Powell, A. Biswas, J. Emer, S. Mukherjee, B. Sheikh, and S. Yardi, "CAMP: A technique to estimate per-structure power at run-time using a few simple parameters," in *HPCA '09*.

[29] H. Jacobson, A. Buyuktosunoglu, P. Bose, E. Acar, and R. Eickemeyer, "Abstraction and microarchitecture scaling in early-stage power modeling," in *HPCA '11*.

[30] K. Czechowski and R. Vuduc, "A theoretical framework for algorithm-architecture co-design," in *IPDPS '13*.

[31] G. Wang, "Power analysis and optimizations for GPU architecture using a power simulator," in *ICACTE '10*.

[32] K. Ramani, A. Ibrahim, and D. Shimizu, "PowerRed: A Flexible Power Modeling Framework for Power Efficiency Exploration in GPUs," in *GPGPU '07*.

[33] X. Ma, M. Dong, L. Zhong, and Z. Deng, "Statistical Power Consumption Analysis and Modeling for GPU-based Computing," in *HotPower '09*.

[34] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, and S. Matsuoka, "Statistical power modeling of GPU kernels using performance counters," in *IGCC 10*.

[35] J. Chen, B. Li, Y. Zhang, L. Peng, and J. kwon Peir, "Tree structured analysis on gpu power study," in *ICCD '11*.

[36] Y. Zhang, Y. Hu, B. Li, and L. Peng, "Performance and Power Analysis of ATI GPU: A Statistical Approach," in *NAS '11*.

[37] W. Wu and B. C. Lee, "Inferred Models for Dynamic and Sparse Hardware-Software Spaces," in *MICRO '12*.