Register and Thread Structure Optimization for GPUs

Yun Liang[1], Zheng Cui[2], Kyle Rupnow[2,3], Deming Chen[2,4]

[1]Center for Energy-Efficient Computing and Applications, School of EECS, Peking University, China
[2]Advanced Digital Sciences Center, Singapore
[3]Nanyang Technological University, Singapore
[4]University of Illinois at Urbana-Champaign, USA
[1]{ericlyun}@pku.edu.cn, [2]zheng.cui@adsc.com.sg, [3]k.rupnow@ntu.edu.sg, [4]dchen@illinois.edu

Abstract— GPUs are an increasingly popular implementation platform for a variety of general purpose applications from mobile and embedded devices to high performance computing. The CUDA and OpenCL parallel programming models enable easy utilization of the GPU's resources. However, tuning GPU applications' performance is a complex and labor intensive task. Software programmers employ a variety of optimization techniques to explore tradeoffs between the thread parallelism and performance of a single thread. However, prior techniques ignore register allocation, a significant factor in single thread performance and, indirectly affects the number of simultaneously active threads. In this paper, we show that joint optimization of register allocation and thread structure has great potential to significantly improve performance. However, the design space for this joint optimization can be large; therefore, we develop performance metrics appropriate for evaluation within a compiler's inner loop and efficient design space exploration techniques that use the metrics to narrow the search space. Across a range of GPU applications, we achieve average performance speedup of 1.33X (up to 1.73X) with design space exploration 355X faster than the exhaustive search.

I. Introduction

Continually increasing demand for performance, throughput, and energy efficiency has led to a proliferation of heterogeneous computing platforms. One common, low-cost heterogeneous platform couples graphics processing units (GPUs) with CPUs; GPUs efficiently execute highly data-parallel functions while CPUs execute control-heavy code. Programming models such as CUDA (Compute Unified Device Architecture) [1] and OpenCL (Open Computing Language) [3] have made this platform design popular and accessible to a wide range of programmers. GPU-CPU platforms are now widely used in supercomputers, desktop and laptop, and embedded platforms. Programmers for these platforms write applications that use GPUs to accelerate the data-parallel functions, called *kernels*. The GPUs typically contain hundreds of processing cores, which allow faster and more energy efficient execution of data-parallel applications such as image processing, graph algorithms, and biomedical applications [11].

Although the CUDA and OpenCL programming models have made it easy to map data-parallel kernels to GPU hardware, it remains complex and labor intensive to optimize the kernels to efficiently use the GPU hardware [12]. GPUs are composed as a hierarchy of compute units; in the NVIDIA terminology, multiple streaming processors (SPs) are grouped together with shared memory, registers, and thread program counters into a streaming multi-processor (SM). Because the memory, registers, and program counters are shared among SPs within a SM, resource limitations affect both single thread performance and the number of active threads.

In CUDA, threads are organized into thread blocks, which are further organized into grids. We define this thread block and grid organization as the *thread structure*. The performance of GPU applications depends on single thread performance and the number of simultaneously active threads.[1] Thread structure and register allocation are interrelated — thread structure affects both single thread performance and the number of active threads, and similarly register allocation affects both single thread performance and the number of active threads. Changes in thread structure affects the total number of threads, the amount of work per thread and the granularity of assigning threads to SMs; changes in register use per thread affects the instruction counts, dependencies, memory accesses of one thread and the number of threads that can be simultaneously active in the SM [2].

In this paper, we demonstrate the acceleration potential of joint optimization of register allocation and thread structure compared to the default compiler settings. The design space for this joint optimization is large and impractical for exhaustive exploration within the compiler. Thus, we develop efficient design space exploration techniques based on novel fast and simple performance metrics to quickly identify the optimal or near-optimal solution with a combination of analytical modeling and limited empirical measurement. This paper contributes to the state-of-the-art as follows:

- We propose a joint register and thread structure optimization framework for GPUs that achieves an average 1.33X (up to 1.73X) kernel performance speedup over the default setting on NVIDIA GTX480 across a range of GPU applications.
- We propose efficient performance metrics and design space exploration algorithms suitable for integration into the CUDA compiler, achieving design space exploration runtime speedup of 355X over an exhaustive search.

[1]The thread structure specifies the total number of threads, but the number of simultaneously active threads is determined by GPU hardware resource limitations.

978-1-4673-3030-5/13/$31.00 ©2013 IEEE

II. Overview of CUDA and GPU Architecture

The CUDA programming model is based on ANSI C with extensions to support CUDA's data parallel execution. In this model, software programmers write separate sections for host (CPU) and kernel (GPU) codes. The GPU is used as a coprocessor for accelerating data-parallel kernels. In CUDA kernels, groups of 32 threads are organized into warps, and one or more warps are grouped together into a thread block. Then thread blocks are further organized into a grid structure. Thus, thread structure can be specified by thread block and grid size. Thread blocks are scheduled to execute on one of the GPU SMs, and the thread block's warps execute in a SIMD fashion on that SM. SMs perform zero overhead scheduling to interleave warps within a thread block and hide memory access latency. We can tune the total threads by varying the thread structure. A more detailed description of CUDA can be found in [1].

GPUs are many-core architectures; the NVIDIA GTX480 used in this paper has 15 SMs (480 cores). Each SM has private registers and memory which are shared among the threads and thread blocks running on the SM. In recent generations of GPUs, architectures have consistently increased the shared resources per SM. These increases improve performance potential and flexibility in algorithm implementation, but this flexibility also increases the design space for implementation. Thus, the joint optimization of register allocation and thread structure will continue to gain importance as these design space parameters continue to grow.

In this paper, we explore a joint design space between register allocation and thread structure with particular emphasis on feasibility of integrating this optimization into the CUDA compiler. In particular, the design space we consider includes three factors as follows:

- reg: the number of registers allocated per thread.
- $blkSize$: the number of threads per thread block.
- $gridSize$: the number of thread blocks of the kernel.

The total number of threads of the kernel is just the product of $blkSize$ and $gridSize$. We can vary the maximum allocated registers between 16 and 63 per thread using the NVIDIA compiler's (nvcc) register limit parameter. When a GPU kernel is invoked in CUDA, thread structure ($blkSize$ and $gridSize$) are the first two arguments passed to the kernel. Most GPU kernels support multiple $gridSize$ and $blkSize$ settings without code modification. Collectively, these three factors form a large design space.

III. Performance Metrics

GPU architecture simulators [5], analytical models [9, 4, 14, 7], and micro-benchmarking methods [13] have all been used to analyze the performance bottlenecks of GPU architectures. However, these models are inappropriate for our problem and integration into a CUDA compiler for several reasons:

- Architectural simulation is orders of magnitude too slow: simulation of one test configuration may be greater than 100X slower than measurement.
- Some analytical models and simulations are based on CUDA's intermediate representation, PTX (Parallel Thread Execution) code, but register allocation is performed when PTX is compiled into the architecture's binary (CUBIN).
- Prior methods aim for perfect performance prediction fidelity at the cost of runtime — methods that can be integrated into a compiler's inner loop may need to sacrifice prediction fidelity to a certain extent for evaluation speed.

Our metrics can estimate the performance effects of register allocation and thread structure to the first order. Then, the design space exploration algorithms use the metrics to narrow the search space, as explained in the next section.

Performance. The performance of GPU kernels depends on the performance of a single thread and the number of threads that can be active at the same time. Our performance metric is designed to model both aspects quickly enough to integrate such a performance model into a GPU compiler.

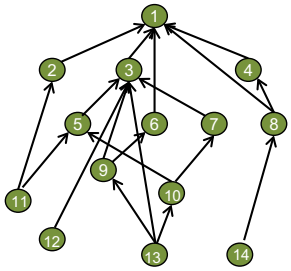
To estimate the performance of a single thread, we analyze the kernel's detailed machine code generated using cuobjdump [1]. Cuobjdump is a recently released official disassembler for NVIDIA's machine-level instructions. For different compiler settings such as register allocation limits, or CUDA API, the generated machine code will be different. These differences in the generated machine code will result in different single thread performance. By default, nvcc attempts to allocate a large amount of registers per thread in an attempt to maximize single thread performance. The $blkSize$ and $gridSize$ also influence performance as different $blkSize$ and $gridSize$ values imply different amounts of threads using the shared resources and a different granularity in assigning threads to SMs.

Based on the generated machine code, we first construct the control flow graph (CFG). Then for each basic block, we model read after write (RAW), write after read (WAR) and write after write (WAW) data hazards and schedule instructions in the basic block to ensure that no instruction violates those hazards. We do not use register renaming to eliminate data hazards because this performance model is intended to differentiate between register allocations. We use the micro-benchmarking methods as in [13] to determine the execution latency of arithmetic and memory instructions. For the control and other miscellaneous instructions, we assume 1 cycle latency. Finally, different from the older GPUs that only have shared memory as on-chip memory, GTX 480 architecture (Fermi) contains caches in the memory hierarchy. Thus, the access latency of load/store operations depend on whether they are cache hits or misses. We empirically measure the cache hit ratio using the default compiler settings for all studied applications and then compute an average memory access latency. Using these latencies and the instruction schedule, we compute the estimated latency of each basic block.

In Figure 1, we show an example of instructions of a basic block from *BlackScholes* and its instructions dependency graph. In this basic block, there are 14 instructions (Figure 1(a)). The first operand followed by the operator specifies the destination and the remaining operands specify the sources according to the manual of cuobjdump [1]. The data dependencies between instructions are shown in Figure 1(b). As previously described, all the three types of dependencies are considered. In each clock cycle, ready instructions are scheduled.

```
1.  MOV   R5,    R4;
2.  F2F   R4,    R4;
3.  FFMA  R52,   R55,   c[xxx],  R5;
4.  FMUL  R53,   R5,    xxx;
5.  MOV   R55,   xxx;
6.  FSETP P0,    xxx,   R5,    xxx;
7.  F2F   R56,   R52;
8.  FMUL  R53,   R5,    R53;
9.  FMUL  R5,    R52,   xxx;
10. FFMA  R57,   R56,   c[xxx], R55;
11. FFMA  R4,    R4,    c[xxx], R55;
12. FSETP P1,    xxx,   R52,   xxx;
13. FMUL  R56,   R52,   R5;
14. FMUL  R58,   R53,   xxx;
```

(a) Disassembled code



(b) Instruction dependency graph

Fig. 1. Disassembled code and instruction dependency graph of basic blocks. In sub-figure (a), xxx denotes either memory address or processor status register.

The latency of the basic block is just the completion time of the latest finishing instruction. As we will demonstrate, this latency estimation effectively distinguishes between different register allocations although it does not consider some factors in single thread performance.

We can now compute expected latency of each basic block, but basic blocks execute with different frequency depending on program control flow. For each basic block b, let $Cycle[b]$ be the execution latency as estimated based on the instructions dependence graph and $Freq[b]$ be the execution frequency collected through profiling. Then, we estimate the single thread execution latency as follows

$$Cycle(kernel) = \sum_{b \in \mathbf{B}} Cycle[b] \times Freq[b] \qquad (1)$$

where $\mathbf{B}$ is the set of all basic blocks in the kernel.

Occupancy. Occupancy is a metric to measure how well a source code takes advantage of GPUs' ability to execute many threads in parallel. Occupancy is defined as the ratio of simultaneously active threads to the maximum number of threads supported on one SM [2]. Occupancy can be easily computed based on resource usage per thread (e.g., reg, shared memory), $gridSize$, $blkSize$. The NVIDIA nvcc compiler provides an interface to obtain the occupancy at compile-time [2].

Figure 2 shows the speedup over default setting and occupancy on GTX480 of the *BlackScholes* program for a few selected $blkSize$ values and fixed $gridSize$ value; the default and optimal settings are indicated. For each $blkSize$, we vary the reg value. As shown, high occupancy does not correlate with high performance. The optimal performance setting has 70% occupancy while settings with 100% occupancy have performance 10% to 50% slower than the optimal. Furthermore, as shown, the performance of GPU applications critically depends on both register allocation and thread structure.

Intuitively, the reason for these results is that occupancy ignores single thread performance. Settings with small register usage ($16 \leq Reg \leq 20$) achieve high occupancy because those settings allow the most simultaneously active threads. However, this reduced register usage also seriously degrades single thread performance, and the increases in active threads may not compensate for reduced single thread performance. On the other hand, if we allocate significantly more registers per thread (e.g., the default setting), the improvement in single thread performance may not compensate for the reduction in active threads. Overall kernel performance depends on both single thread performance and occupancy, therefore we now combine these factors in order to estimate overall kernel performance.

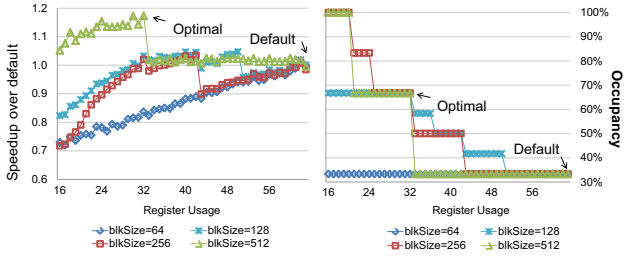


Fig. 2. Subset of the register and thread structure design space of *BlackScholes*. Optimal setting is ($blkSize = 512$, $gridSize = 720$, $Reg = 32$) and the default setting is ($blkSize = 256$, $gridSize = 720$, $Reg = 63$). Speedup is over the default setting.

Register and Occupancy (RO). A very simple metric of single thread performance is the register use; the $RegRatio$ is the ratio of registers used per thread to the maximal allowed registers per thread. $RegRatio$ models the performance as a linear function of register allocation. RO metric combines $RegRatio$ with occupancy as $Occupancy \times RegRatio$ to estimate the overall GPU kernel performance. Note that $0 < RegRatio \leq 1$ and $0 < Occupancy \leq 1$. Thus, the product of $Occupancy$ and $RegRatio$ is also a value between 0 and 1, where high values represent good tradeoffs between single thread performance and many threads simultaneously active. In the next section, we will describe how we use this metric for design space exploration.

Performance and Occupancy (PO). The performance and occupancy metric combines our more detailed performance model with the occupancy. The PO metric is a 2-tuple: (T, C), where T denotes the remaining space for active threads on one SM and C denotes the estimated latency in cycles of a single thread. T is computed using $MaxThreads \times (1 - Occ)$, where $MaxThreads$ denotes the maximal allowed threads per SM and Occ denotes the occupancy. C is computed using Equation 1. In the next section, we will describe design space exploration methods that determine which candidate 2-tuples are most likely to be the optimal solution.

IV. Design Space Exploration

In this section, we present several design space exploration algorithms that offer different GPU kernel performance and design space exploration runtime tradeoff results. The algorithms use the performance metrics to prune the search space and empirical search (e.g., measuring the execution time on the GPU) to evaluate selected candidate solutions. We use exhaustive search to obtain the optimal setting. But it is not feasible to integrate exhaustive search into compiler as it is too slow. We compare our design space exploration algorithms with exhaustive search in terms of the final GPU kernel performance and design space exploration runtime. Our ultimate goal is to identify an efficient and effective design space exploration algorithm that offers the best kernel performance with minimum exploration overhead.

Exhaustive Search (ES). For each combination of reg, $blkSize$, and $gridSize$ in the design space, compile the kernel, execute, and measure actual performance. Select the setting that yields the best kernel performance.

RO Search (ROS). Using the RO metric from section III, choose the combinations that maximize the RO value. If more than one solution has a maximal RO value, ROS automatically measures the performance of all these solutions to tie-break.

PO Search (POS). Using the PO metric from section III, find the set of candidate solutions. The design space exploration is described in Algorithm 1. We first derive all the possible values for total threads given various thread structures. Given a particular value of total threads, we formulate the problem as a Pareto-optimal problem [8]. For each setting $(gridSize, blkSize, reg)$, the value of the PO metric is (T, C), where T indirectly models the number of active threads and C models the performance of a single thread, as described in section III. Given the occupancy, T can be easily computed (line 11). C is computed using Equation 1 (line 12). For every candidate solution (T, C), if no other candidate solution is better in both T and C, it is part of the Pareto-optimal set; otherwise, if there is another candidate that has both better T and better C, it is a dominated solution and can be pruned without loss. For example, consider two candidates, a and b. If candidate a has both more active threads ($T_a < T_b$) and better performance of a single thread ($C_a < C_b$), then candidate a dominates candidate b and b can be pruned from the design space. In contrast, if candidate a has more active threads than b, but lower single thread performance, then both a and b are Pareto-optimal and should be evaluated. The Pareto-optimal solutions $\{(T_1, C_1), ..., (T_n, C_n)\}$ represent good candidates, which trade between the number of active threads and the performance of a single thread. In the POS algorithm, for each possible value of total threads, we compute the Pareto-optimal points because different values of total threads imply different workload per thread, and it is thus difficult to directly compare their T and C values. We empirically search the Pareto-optimal points and select the best solution.

Algorithm 1: Performance and Occupancy Search Algorithm

```
1  Let MaxThreads be the maximal threads allowed on one SM;
2  Let Threads be the set of possible total number of threads;
3  Global_Sol = ∅ ;
4  foreach T ∈ Threads do
5      Sol = ∅ ;
6      Config = {⟨gridsize, blksize⟩ | gridsize × blksize = T} ;
7      foreach ⟨gridSize, blkSize⟩ ∈ Config do
8          foreach reg ∈ Reg do
9              Let s be ⟨gridSize, blkSize, reg⟩;
10             Occ = compute_occupancy(s);
11             T = MaxThreads × (1 − Occ);
12             C = estimate_cycle(s);
13             Sol = Sol ∪ (T, C);
14
15     Select the set of Pareto-optimal solutions P from Sol;
16     Global_Sol = Global_Sol ∪ P ;
17
18 foreach sol ∈ Global_Sol do
19     Let ⟨gridSize, blkSize, reg⟩ be the setting of sol;
20     time = measure(gridSize, blkSize, reg);
21     if time < best_time then
22         best_time = time;
23         best_setting = ⟨gridSize, blkSize, reg⟩;
24
25
```

Figure 3 shows an example of Pareto-optimal curve and dominated points for *MarchingCubes* benchmark. The design space is significantly pruned (by 96.4% for the dominated points) by only considering the Pareto-optimal points. However, due to the non-linear performance effect of active threads and single thread performance on GPU performance, POS algorithm selects the best solution by evaluating each Pareto-optimal solution on the GPU.

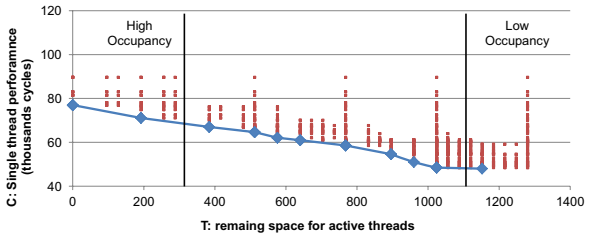


Fig. 3. Pareto-optimal curve for *MarchingCubes*. The points on the Pareto-optimal curve are Pareto-optimal solutions. The other points (red) are dominated solutions.

PO Search Filter (POSF). In POS algorithm, there are some Pareto-optimal points unlikely to be the globally optimal solution. For example, although occupancy does not directly correlate to performance, very low occupancy is an indicator that insufficient threads will be active to keep the hardware busy. As shown in Figure 2, extreme high occupancy is an indication that the individual thread's resource use is extremely low and thus single thread performance may be affected. These solutions are less likely to be the optimal solution. Therefore, we filter the Pareto-optimal design candidates to the set of solutions within the occupancy range most likely to produce the best candidate. Figure 3 highlights the high (left corner) and low occupancy (right corner) ranges as an example. The high and low occupancy range are selected empirically based on the tradeoff between performance and design space exploration runtime.

In total, we present three design space exploration techniques with different GPU kernel performance and design space exploration runtime tradeoffs; ROS chooses the setting based on the product of register allocation ratio and occupancy, and only uses measurement when multiple candidates are equal in terms of the metric. POS and POSF model active threads and single thread performance, and only measure the performance for the set of Pareto-optimal solutions.

Compiler Integration and Portability. All of the proposed algorithms rely on techniques that are feasible for integration in a compiler: the register allocation per thread (reg) setting and occupancy value are all already available within the NVIDIA GPU compiler (nvcc); thread structure ($blkSize$, $gridSize$) are GPU kernel call arguments. Thus, our techniques are suitable for compiler integration and portable to any GPU architecture without user intervention. Our analysis and performance metrics narrow the design space in a more robust manner than pure analytical models [9, 4] which are only accurate for the machines with the exact modeled hardware features and not portable to other architectures.

V. Experimental Results

We evaluate our techniques on NVIDIA GTX480. We select a set of benchmarks from CUDA SDK [1], Rodinia [6], and a real-world application for 3D sound localization [10], as shown in Table I. The MC, NB, Par, and CFD benchmarks have fixed total number of threads. Thus, we vary $blkSize$ and update $gridSize$ correspondingly ($\frac{totalThreads}{blkSize}$). For Aud, we vary $blkSize$ only as the $gridSize$ is fixed for this particular

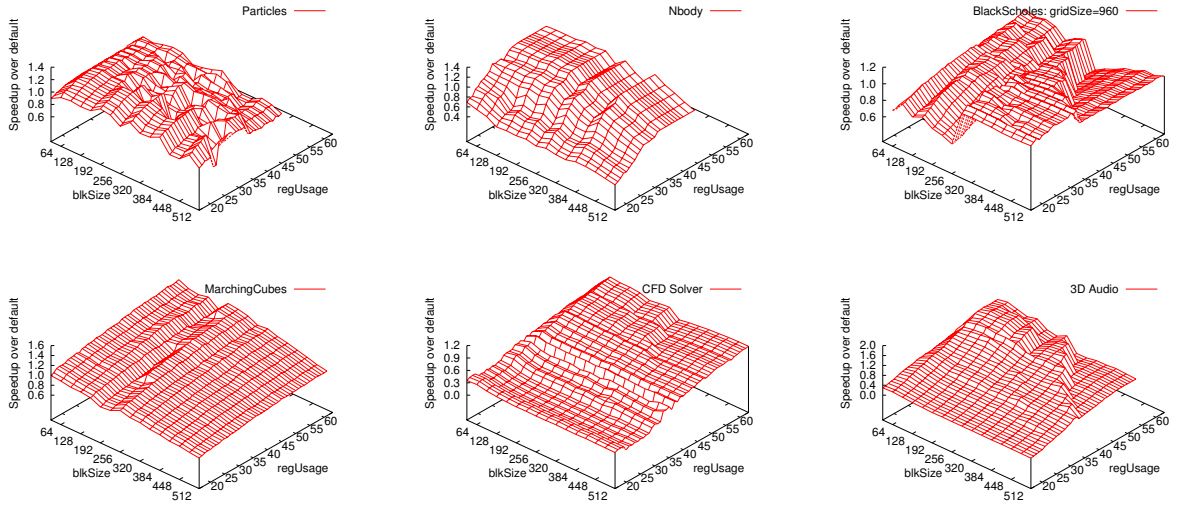


Fig. 4. The register and thread structure design space. Speedup is over the default setting. For *Particles* and *Nbody*, the maximum register use per thread is 45. For *3D Audio*, the maximum register use per thread is 53.

TABLE I
BENCHMARKS AND RUNTIME COMPARISON.

Benchmark	Source	DSE runtime (sec)			
		ES	RO	POS	POSF
BlackScholes (BS)	CUDA SDK [1]	14472	55	693	244
MarchingCubes (MC)	CUDA SDK [1]	3888	47	465	169
Nbody (NB)	CUDA SDK [1]	11665	95	667	64
Particles (Par)	CUDA SDK [1]	338	8	416	76
3D Audio (Aud)	[10]	5236	21	1649	274
CFD Solver (CFD)	Rodinia [6]	4364	21	70	34

application. For BS, we vary both $blkSize$ and $gridSize$, so it is a good example to demonstrate performance variation along both dimensions.

Although nvcc's register allocation algorithm is unknown, the *maxrregcount* parameter can be used to specify the maximum register use per thread in the range of 16 to 63 registers. In theory the actual register use may be less than the maximum; therefore, we use nvcc's post-compilation analysis interface to verify the actual register use for each compilation setting. We compute the performance speedup of settings chosen by our design space exploration algorithms by comparing to the default setting. In the default setting, we use the original thread structure and do not specify a register limit. By default, nvcc attempts to use a large amount of registers per thread. Thus, in practice, the register limit we set is often less than the default register use.

The POS algorithm requires additional basic block execution frequency information for its thread performance modeling. Many GPU kernels avoid using complex control flow that might cause thread divergence [7] (i.e., basic block execution frequencies are identical per thread without divergence). Thus, for most GPU kernels we can derive the exact basic block execution frequency via static program analysis. We focus on the speedup potential of the joint register and thread structure optimization in this work. Hence, for GPU kernels with complex control flow, we rely on GPGPU-Sim [5] to collect the exact basic block frequencies. If there is divergence, we use the maximum execution frequency encountered for each basic block. Finally, for the POSF algorithm (POS with occupancy filtering), we empirically select the occupancy filtering range to be 0.3-0.5 based on the tradeoff between performance and design space exploration runtime.

All the experiments are performed on an NVIDIA GTX480 hosted on a machine with an Intel quad-core i5-750 2.67GHz CPU and 3GB of RAM. We use the CUDA profiler [1] to measure the kernel performance on GPU. In the following, we will first present the performance variation across the register allocation and thread structure design space. Then, we will compare our design space exploration algorithms (ROS, POS, and POSF) in terms of selected kernel setting performance, design space exploration runtime and explored search space.

Design Space. Threads are scheduled for execution in warps (a group of 32 threads). Thus, if $blkSize$ is not a multiple of 32, the last warp will contain fewer than 32 threads and waste compute resources. Therefore, $blkSize$ values are multiples of 32 between 32 and 512. Similarly, the GTX480 allows register allocation between 16 and 63 registers. In Figure 4, for each application, we show the speedup over default setting for all the points in the design space. We use exhaustive search to measure the actual performance of all the points in the search space on the GPU. In Figure 4, we show *BlackScholes* with one of the $gridSize$ value. The remaining $gridSize$ settings are not shown here due to space limitation, but similar variation has been observed.

As shown, the kernel performance is very sensitive to the register allocation and thread structure for all the applications. Exploring only one or the other can easily end with a setting trapped in a local optima. To find the global optimal setting, designers have to explore the joint design space simultaneously. For all of the tested applications, the default setting is not optimal; on average, the kernel can be accelerated by 1.36X by jointly optimizing register allocation and thread structure.

Comparison of Exploration Algorithms. Figure 5 compares the speedup of the settings identified by the design space exploration algorithms over the default setting. ES identifies the optimal solution. ROS achieves speedup from 0.38 to 1.31; ROS can select poor solutions because the algorithm assumes

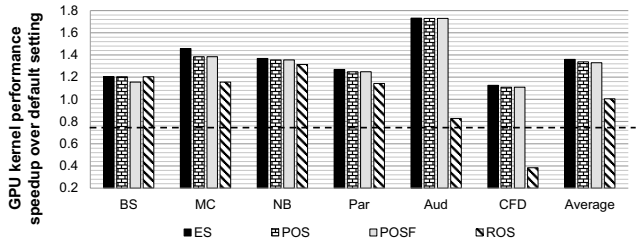


Fig. 5. The comparison of different design space exploration algorithms in terms of GPU kernel performance speedup.

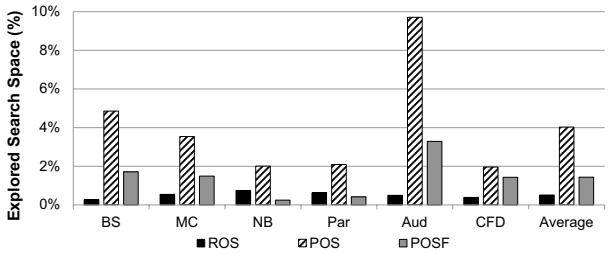


Fig. 6. The comparison of different design space exploration algorithms in terms of size of the explored design space.

that both register use ratio and occupancy are equally important and that small increases to one variable (that don't affect the other) will always yield performance improvement. However, we can see in Figure 2 that there are plateaus where increases in register allocation or occupancy do not affect overall performance. POS performs better thanks to the single thread performance model used together with occupancy and Pareto-optimal candidate selection. POS improves performance by up to 1.73X, with average 1.34X speedup, compared to average 1.36X performance speedup achieved by ES. Similarly, POSF performs well even with a filtered design space: it achieves average 1.33X performance speedup. The settings chosen by POS and POSF are either optimal or close to the optimal (on average, within 2% of the optimal speedup).

Table I compares the design space exploration runtime overhead for all the algorithms to the exhaustive search. ES is slow because it measures all the design points on the GPU and chooses the best one. ROS significantly reduces the search space, with total exploration time reduced from hours to seconds because ROS only measures kernel performance to break ties between settings with equal metric estimations. POS searches more design points than ROS because there tend to be more Pareto-optimal design points, but POS also achieves better performance kernel speedup and still achieves average 60X design space exploration runtime reduction. POSF further improves that design space exploration runtime reduction to average 355X by filtering some of the Pareto-optimal points unlikely to be the optimal.

Figure 6 compares the size of the explored design space of different algorithms. On average across all the kernels, ROS only explores 0.5% of the search space. Intuitively, ROS seems an attractive solution because it effectively improves exhaustive search by pruning the search space using the metric. However, as shown in Figure 5, it fails to find the optimal or near-optimal solution for most of the benchmarks. Although, POS and POSF explore 4% and 1.4% of the search space on average, respectively, they can find the optimal or near-optimal solutions. It is also important to note that these percentages treat each design point equally. However, poor design points may take significantly longer to execute on GPU (and thus measure). The total runtime in Table I includes the measurement latency, so an algorithm may test an equal number of design candidates, but have lower total latency if on average it only evaluates good candidates.

In summary, ROS effectively prunes the design space, but the chosen designs are not necessarily close to the optimal. Both POS and POSF provide better performance of the selected kernel settings (1.34X and 1.33X speedup, respectively) while still reducing average design space exploration runtime by 60X and 355X respectively.

VI. Conclusion

Performance optimization is a critical component of the GPU kernel design process. In this paper, we demonstrate that joint optimization of register allocation and thread structure has large potential to improve GPU kernel performance, and the trend in GPU architectures towards more SMs and more shared resources in each SM will make this joint optimization more important in the future. However, the design space can be large. Therefore, we developed several metrics to efficiently estimate the performance and design space exploration algorithms that use the metrics to narrow down the search space. The Performance and Occupancy Search with occupancy Filtering (POSF) algorithm performs a smart exploration of likely Pareto-optimal design points to improve kernel performance by 1.33X (up to 1.73X) on average with design space exploration runtime 355X faster than the exhaustive search.

VII. Acknowledgments

The Advanced Digital Sciences Center is funded by A*STAR Singapore under the Human Sixth Sense Project.

References

[1] NVIDIA. NVIDIA CUDA Programming Guide, Version 3.2.

[2] NVIDIA. Occupancy Calculator. `http://developer.nvidia.com/object/cuda_3_2_toolkit_rc.html`.

[3] OpenCL. `http://www.khronos.org/opencl`.

[4] S. S. Baghsorkhi et al. An adaptive performance modeling tool for GPU architectures. In *PPoPP*, 2010.

[5] A. Bakhoda et al. Analyzing CUDA worloads using a detailed GPU simulator. In *ISPASS*, 2009.

[6] S. Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.

[7] Z. Cui et al. An accurate GPU performance model for effective control flow divergence optimization. In *IPDPS*, 2012.

[8] K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, 2001.

[9] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ISCA*, 2009.

[10] Y. Liang et al. Real-time implementation and performance optimization of 3D sound localization on GPUs. In *DATE*, 2012.

[11] J. Owens et al. GPU computing. *Proceedings of the IEEE*, 96:879 – 899, 2008.

[12] S. Ryoo et al. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP*, 2008.

[13] H. Wong et al. Demystifying GPU microarchitecture through microbenchmarking. In *ISPASS*, 2010.

[14] Y. Zhang and J. D. Owens. A quantitative performance analysis model for GPU architectures. In *HPCA*, 2011.