Enabling Coordinated Register Allocation and Thread-level Parallelism Optimization for GPUs

Xiaolong Xie[1], Yun Liang[1,2], Xiuhong Li[1], Yudong Wu[1], Guangyu Sun [1,2], Tao Wang[1,2], and Dongrui Fan[3]

[1]*Center for Energy-efficient Computing and Applications, School of EECS, Peking University, China*
{xiexl_pku, ericlyun, lixiuhong, wuyd_pku, gsun, wangtao@pku.edu.cn}
[2]*Collaborative Innovation Center of High Performance Computing, NUDT, China*
[3]*Institute of Computing Technology, Chinese Academy of Sciences, {fandr@ict.ac.cn}*

ABSTRACT

The key to high performance on GPUs lies in the massive threading to enable thread switching and hide the latency of function unit and memory access. However, running with the maximum thread-level parallelism (TLP) does not necessarily lead to the optimal performance due to the excessive thread contention for cache resource. As a result, thread throttling techniques are employed to limit the number of threads that concurrently execute to preserve the data locality. On the other hand, GPUs are equipped with a large register file to enable fast context switch between threads. However, thread throttling techniques that are designed to mitigate cache contention, lead to under utilization of registers. Register allocation is a significant factor for performance as it not just determines the single-thread performance, but indirectly affects the TLP.

The design space of register allocation and TLP presents new opportunities for performance optimization. However, the complicated correlation between the two factors inevitably lead to many performance dynamics and uncertainties. In this paper, we propose Coordinated Register Allocation and Thread-level parallelism (*CRAT*), a compiler-based performance optimization framework. In order to achieve this goal, *CRAT* first enables effective register allocation. Given a register per-thread limit, *CRAT* allocates the registers by analyzing the lifetime of variables. To reduce the spilling cost, *CRAT* spills the registers to shared memory when possible. Then, *CRAT* explores the design space by first pruning the design points that cause serious Ll cache thrashing and register under utilization. After that, *CRAT* employs a prediction model to find the best tradeoff between the single-thread performance and TLP. We evaluate *CRAT* using a set of representative workloads on GPUs. Experimental results indicate that compared to the optimal thread throttling technique, our framework achieves performance improvement up to 1.79X (geometric mean 1.25X).

MICRO-48, December 05-09, 2015, Waikiki, HI, USA
 ISBN 978-1-4503-4034-2/15/12 $15.00
DOI: http://dx.doi.org/10.1145/2830772.2830813

1. INTRODUCTION

The high performance on Graphics Processing Units (GPUs) is realized through massive threading to enable fast context switch between threads and latency hiding. Modern GPUs support concurrently executing thousands of threads. Switching between threads occurs so frequently on GPUs that the execution contexts of threads have to be stored within on-chip memories. Therefore, GPUs employ a large *register file* (RF) to store the variables of threads to provide the maximal context switching capability. For example, each *streaming multiprocessor* (SM) on NVIDIA GTX680 is equipped with a 256 KB register file, which is larger than the total size of L1 cache and shared memory (64 KB) on an SM. More importantly, the register file capacity of GPUs also increases with each new generation [1].

The massive threading of GPUs is a strength for performance acceleration, but places extreme loads on the cache subsystem. The large number of concurrent threads compete for the limited shared cache capacity, leading to high cache contention and thus low cache hit rate [2, 3]. As a result, running with the maximum number of threads does not necessarily give the optimal performance. Thread throttling techniques are designed to mitigate the excessive cache contention by limiting the number of threads that concurrently execute [2, 3]. Recently, thread throttling has been used together with cache bypassing to further improve the cache performance [4]. The granularity of thread throttling can vary from fine-grained (warps) [2] to coarse-grained (thread blocks) [3]. Figure 1 (a) demonstrates the benefits of thread throttling for a variety of applications. In this paper, we define the TLP as the number of concurrently executing thread blocks per SM. *MaxTLP* represents the maximum allowed TLP given the resource and hardware limits; *OptTLP* represents the optimal TLP determined through profiling. For these applications, *OptTLP* uses only about 55% threads of *MaxTLP* on average. As shown, the performance can be improved by 1.42X on average through thread throttling. The achieved performance gain is attributed to the improvement of cache hit rate and reduction of stall caused by cache resource congestion.

However, thread throttling techniques that are designed to preserve the data locality and mitigate cache contention are oblivious to register allocation. More clearly, in the current compilation tool-chain for GPUs, the register per-thread is determined at compile-time. Register per-thread impacts the number of register spills of the thread, which are intrin-

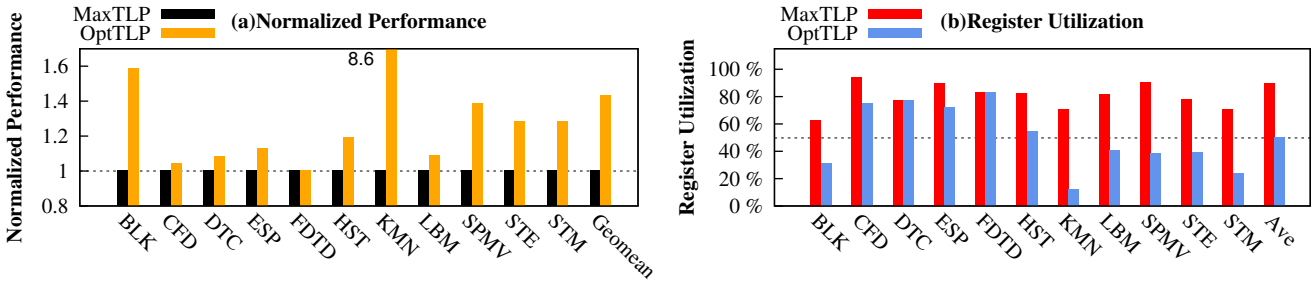


Figure 1: Normalized performance and register utilization for *MaxTLP* and thread throttling technique *OptTLP*.

sically load and store operations. After that, the TLP can be either determined through profiling or adjusted during runtime [2, 3, 5]. However, regardless of how the TLP is determined, register allocation does not have the flexibility to change with the TLP, leading to register resource waste. For example, in Figure 1 (b), we compare the register utilization of *MaxTLP* and *OptTLP* for the applications in Figure 1 (a). Though thread throttling improves the performance, it results in 51.3% register waste.

Neither pure thread throttling nor pure register allocation exploits the entire spectrum of the optimization space. We observe that thread throttling presents new opportunities for register allocation. If thread throttling is coordinated with register allocation, we will not only maintain the TLP that does not cause serious cache contention, but also have the opportunities to utilize the extra register space saved by thread throttling to improve the single-thread performance. By allocating more registers per-thread, we can improve the performance by reducing the number of spills.

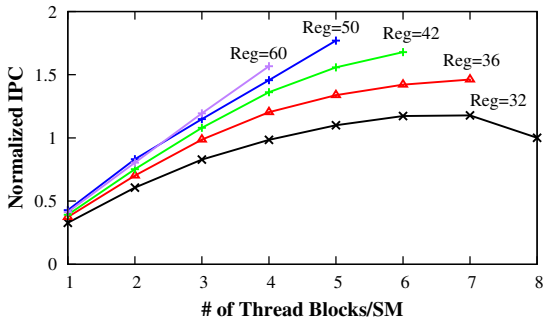


Figure 2: Design space of register per-thread and TLP for application *CFD* on NVIDIA GTX680.

Motivating Example. We illustrate the benefit of the coordinated approach using application *CFD*. Figure 2 presents the design space of TLP and register per-thread for *CFD*. The experiments are performed on NVIDIA GTX680. We vary the register per-thread through the *max regcount* interface built in *nvcc* compiler. In order to vary the TLP, we declare a dummy array in shared memory and vary its size by modifying the source code. The original *CFD* kernel does not use any shared memory and we add simple commands to access the dummy array.

Figure 3 compares a few solutions in Figure 2 in details. Each solution is represented with a 2-tuple:(*reg*,*TLP*), where

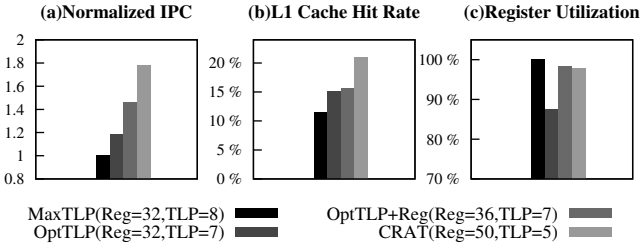


Figure 3: Performance, cache behavior, and register utilization of selected design points for application *CFD*.

reg denotes the register per-thread. Solution *MaxTLP* executes as many thread blocks concurrently as possible using the default register per-thread. The maximal TLP depends on the resource usage per thread block (e.g. registers, shared memory) and the hardware limits (e.g. 8 thread blocks per SM on GTX680). Solution *OptTLP* is the thread throttling technique [3], which limits the TLP and uses the default register per-thread. For application *CFD*, *MaxTLP* and *OptTLP* are (*reg* = 32, *TLP* = 8) and (*reg* = 32, *TLP* = 7), respectively. As shown by Figure 3, *OptTLP* increases the performance by improving the L1 cache performance. However, this also leads to 12.5% register waste. Solution *OptTLP+Reg*,(*reg* = 36, *TLP* = 7) further improves the performance by maintaining the optimal TLP but allocating more registers per-thread. If we continue to increase the per-thread register usage, register will eventually become the limiting resource. Then, the TLP will be penalized as shown by Figure 2. This leads to a non-trivial tradeoff between the TLP and register per-thread. Intuitively, if we allocate more registers per-thread, this will help to improve the single-thread performance by reducing the number of spills. However, this will also reduce the TLP as the number of registers required by a thread block increases. Figure 2 depicts the tradeoff. Solution identified by *CRAT* finds the best tradeoff between the TLP and single-thread performance. For application *CFD*, *CRAT* is (*reg* = 50, *TLP* = 5), which achieves 1.78X performance speedup compared with the *MaxTLP* solution.

Exploring register allocation together with thread throttling has great potential for performance improvement. However, the large design space and the non-trivial performance tradeoff necessitate an automatic analysis and optimization tool. In this paper, we propose Coordinated Register Allocation and Thread-level parallelism (*CRAT*), a compiler performance optimization framework. *CRAT* first enables ef-

fective register allocation by extending GPGPU-Sim compilation framework [6]. Then, *CRAT* explores the design space by pruning the design points that cause serious L1 cache thrashing and register under utilization. For the remaining design points, *CRAT* performs the register allocation and employs a prediction model to compare their performance. To reduce the spilling cost, *CRAT* will spill the registers to shared memory when possible. This paper contributes to the state-of-the-art of GPU optimization techniques as follows:

- We identify the performance bottlenecks of modern GPUs and demonstrate the necessity to optimize the register allocation and TLP coordinately.
- We develop a compiler framework *CRAT* that enables coordinated register allocation and TLP optimization. Our implementation is based on a source-to-source transformation at the *Parallel Thread Execution*(PTX) level. *CRAT* can analyze the lifetime of variables and perform register spilling.
- We develop an optimization algorithm that minimizes the spilling cost by spilling the variables to shared memory and a prediction model that finds the best tradeoff between the TLP and single-thread performance.
- We evaluate our technique using a wide range of applications from Rodinia [7], Parboil [8], and SDK [1]. Evaluations show that compared to the optimal thread throttling technique [3], CRAT improves the performance by up to 1.79X (geometric mean 1.25X).

2. BACKGROUND AND WORKLOAD CHARACTERIZATION

2.1 GPU Architecture

The baseline GPU architecture is shown in Figure 4. GPUs are composed in a hierarchical manner. Processors are grouped into multiple streaming-multiprocessors (SM). Each SM is equipped with warp schedulers, multiple streaming-processors (SP), special function units, load/store units, and on-chip storage including register file, L1 cache, and shared memory. All the SMs share the interconnection network, which connects SMs to off-chip L2 cache and DRAM. Threads are grouped into thread blocks. On each SM, registers, shared memory, threads and thread blocks limits together determine the maximum allowed TLP. More clearly, GPU kernel will launch as many thread blocks concurrently as possible until one or more dimension of resources are exhausted.

In the current compilation tool-chain of GPUs, the register per-thread is determined at compile-time, which subsequently determines the single-thread performance. However, the exact register allocation algorithm has not been disclosed by NVIDIA. In practice, we find that existing register allocation on GPUs is oblivious to thread throttling.

2.2 Workload Characterization

L1 Cache. Caches are introduced to GPUs to broaden the scope of accelerated application. However, distinct from CPU architecture, the cache capacity per-thread on GPUs is very small, leading to poor cache performance [2]. Thread

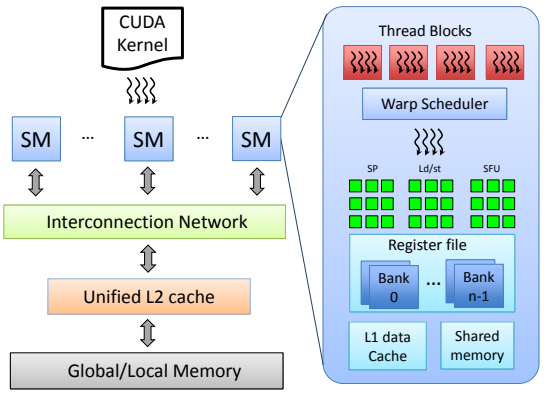


Figure 4: The baseline GPU architecture.

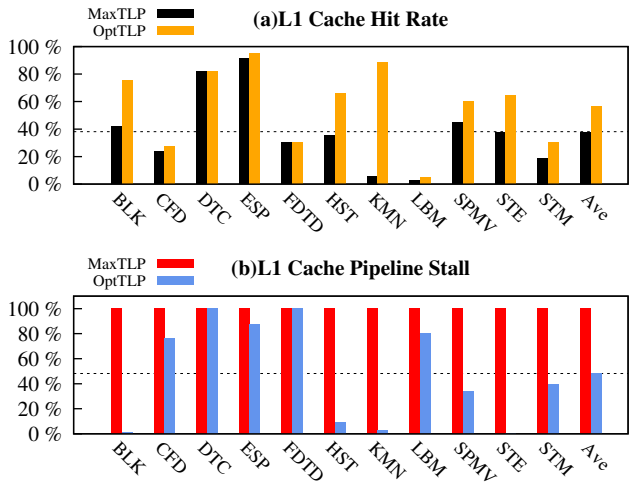


Figure 5: The impact of thread throttling on L1 data cache performance.

throttling techniques are designed to mitigate this problem [3]. First, thread throttling improves the L1 cache hit rate by preserving the data locality as shown by Figure 5(a). Second, thread throttling minimizes the pipeline stall caused by the congestion of cache requests as shown by Figure 5(b). However, as discussed previously, thread throttling techniques can lead to under utilization of registers.

Register Allocation. Modern GPUs are equipped with a large register file. However, often the number of registers is not sufficient and we have to use register spillings to move the variables between registers and local memory. Register per-thread is a very important factor for performance as it not just impacts the single-thread performance, but indirectly affects the TLP. For example, given 2048 threads, each thread is allocated 32 registers at most on GTX680. If the GPU application demands more than 32 registers per-thread, some of the variables have to be spilled to local memory. Figure 6 illustrates this in details using application *CFD* as an example. On one hand, when register becomes the limiting resource, allocating more registers per-thread will limit the TLP as shown by Figure 6 (a). On the other hand, allocating

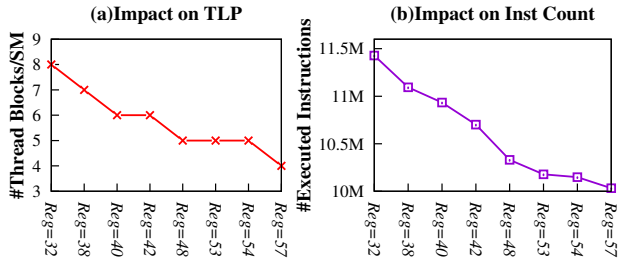


Figure 6: Performance impact of register per-thread on TLP and instruction count for application *CFD*.

less registers per-thread will incur more spilling cost, leading to instruction count increase as shown by Figure 6 (b). Furthermore, the spilled variables are stored in local memory, access to which takes longer latency compared to the access to registers.

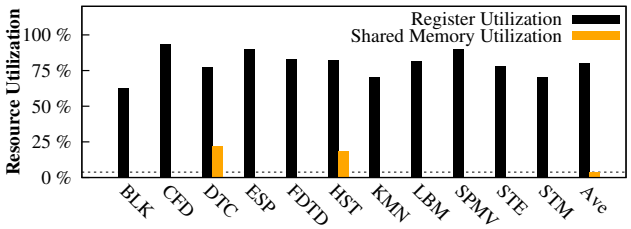


Figure 7: The comparison of register and shared memory utilization.

Shared Memory. On GPUs, shared memory, which is managed by software, is mainly used as a communication channel among threads in the same thread block to reduce the number of accesses to off-chip memory. However, we notice that not all the applications fully utilize the shared memory space. Figure 7 shows that shared memory has a much lower utilization compared to registers (3.8% vs. 65.5%). Shared memory resides on-chip, which is much faster than local memory. Hence, when register pressure happens, instead of spilling the variables to long-latency local memory, we can spill them to the shared memory.

Figure 8 presents the benefits of register spilling using the shared memory for application *FDTD*. First of all, we notice that *FDTD* prefers less registers per-thread. By limiting the registers from 48 to 32, we can increase the performance by 1.35X. We consider two variables var_1 and var_2 as spilling candidates. Both of them have long live intervals and they conflict the most with other variables. Spilling var_2 further improves the performance to 1.64X, while spilling var_1 only improves the performance to 1.41X as shown by Figure 8(b). This is because var_1 has higher access frequency. By spilling var_2 to shared memory, we can reduce the register pressure by 1 and store var_1 in the register. Hence, different variables have different spilling cost and benefit. In Section 5, we design a spilling optimization that minimizes the spilling cost using the available shared memory space.

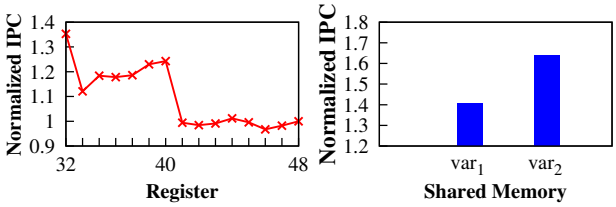


Figure 8: Performance impacts of register and shared memory exploration for application *FDTD*.

3. CRAT COMPILER FRAMEWORK

The experiments of Figure 2 and 3 rely on manual insertion of shared memory and inflexible max regcount interface as the ISA of NVIDIA GPU is not disclosed, which is not a practical solution. Hence, we implement our *CRAT* compiler framework based on GPGPU-Sim [6] compilation system by extending it with register allocation capability.

Figure 9 presents the overview of *CRAT* framework. It employs a source-to-source transformation to enable the coordinated register allocation and TLP optimization. The input and output of *CRAT* are in PTX format [9], which is the intermediate representation of CUDA code. *CRAT* involves three components: design space pruning, register allocation, and optimization as shown by Figure 9. Initially, design space pruning component explores the huge register per-thread and TLP design space and discards the design points with serious cache contention and poor register utilization (Section 4). The output of the design space pruning is a few good candidate solutions. Each solution is a 2-tuple (*reg*, *TLP*), where *reg* denotes the register per-thread and *TLP* represents the number of thread blocks that concurrently execute on the same SM. Then, for each candidate solution, *CRAT* performs register allocation. We extend the GPGPU-Sim compilation system with the register allocation capability. The details of our register allocation implementation is in Section 5. The register allocation contains a spilling optimization algorithm to minimize the register spilling cost using shared memory. Then, the optimization component evaluates different candidate solutions and compares them using a performance model described in Section 6.

4. DESIGN SPACE PRUNING

Design space pruning component collects the resource usage parameters and prunes the design space by discarding the points with serious cache contention and low register utilization.

4.1 Resource Usage Analysis

We first build the control- and data-flow graph based on the intermediate PTX representation. Then, *CRAT* analyzes the kernel and collects the parameters shown in Table 1. The parameters are categorized into three categories.

Register. We collect two parameters about registers: *MaxReg* and *MinReg*. *MaxReg* denotes the number of registers per-thread required to hold all the variables. It depends on applications and we obtain *MaxReg* through data flow analysis as described in [10]. *MinReg* denotes the minimum

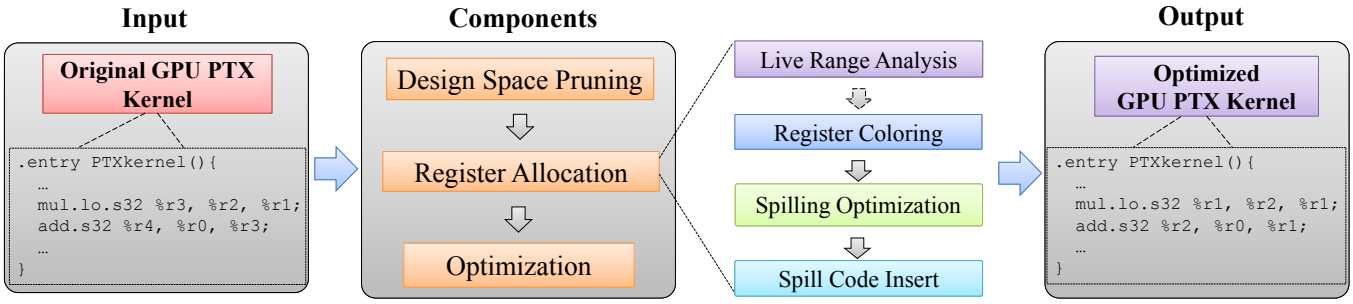


Figure 9: Overview of *CRAT* compiler framework.

number of register per-thread, it is architecture-dependent and we define it as

$$MinReg = \frac{NumRegister}{MaxThreads}$$

where *NumRegister* is the number of registers per SM and *MaxThreads* is the maximum allowed number of concurrent threads per SM. Allocating registers less than *MinReg* per-thread would not limit the TLP and allocating registers more than *MaxReg* per-thread would not increase the single-thread performance. [*MinReg*, *MaxReg*] represents the range for the register per-thread usage.

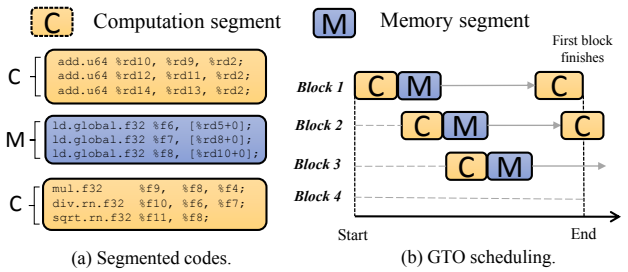
Table 1: Collected Resource Usage Parameters.

Parameter	Category
MaxReg/MinReg	Register usage.
MaxTLP/OptTLP/BlockSize	Thread-level parallelism.
ShmSize	Shared memory per thread block.

TLP. We collect three parameters about TLP: *BlockSize*, *MaxTLP*, and *OptTLP*. *BlockSize* denotes the number of threads per thread block. [*1*, *MaxTLP*] defines the range for TLP. In *CRAT* framework, *OptTLP* can be obtained through profiling, which requires to run the application for a few times (e.g. $\leq$ *MaxTLP*) and finds the optimal one.

As an alternative to profiling, we also propose to estimate the optimal TLP through static code analysis. Prior analytical models [11] have demonstrated that GPU application performance can be accurately predicted by dividing the thread lifetime into computation and memory period and modeling their overlapping through warp scheduling. Thus, our static analysis first analyzes the PTX code and divides the kernels into computation and memory segments as shown by Figure 10 (a). For each segment, we compute its latency by summing the latency of all its instructions. For the memory instructions, we empirically measure the cache hit ratio for all the applications and then compute an average memory access latency using the cache hit ratio.

Recent study [5] has shown that the *OptTLP* can be estimated by using a greedy-warp scheduler (greedy-then-oldest, GTO). The behind intuition is if when the first thread block finishes execution, only $n (n \leq MaxTLP)$ thread blocks are involved in the GTO scheduling, then n thread blocks will be sufficient for this application and is likely to be the $OptTLP$. Based on this observation, we mimic a GTO scheduling as shown by Figure 10 (b) using the computed latency for each segment until the first thread block finishes. We extend it by modeling the memory bandwidth [11] and cache contention. In Figure 10 (b), when the first thread block finishes, only 3 thread blocks are involved suggesting that 3 is the *OptTLP* (*MaxTLP* is 4). In the experiments, we will demonstrate that static code analysis gives accurate *OptTLP* estimation with low overhead.



(a) Segmented codes. (b) GTO scheduling.

Figure 10: Illustration of static code analysis.

Shared memory. *ShmSize* denotes the size of the shared memory requested per thread block. We will use it in the spilling optimization algorithm.

4.2 Pruning

Let $\mathcal{C}$ be the design space of the coordinated register allocation and TLP optimization. Each point $c \in \mathcal{C}$ is represented by a 2-tuple: (*reg*, *TLP*), where *reg* denotes the allocated register per-thread and *TLP* denotes the number of thread blocks that concurrently execute. We have,

$$\mathcal{C} = \{(reg, TLP) | MinReg \leq reg \leq MaxReg \wedge 1 \leq TLP \leq MaxTLP\}$$

In this paper, we focus on the applications that have non-trivial register demand and the TLP is limited by the register. If we allocate more registers per-thread, this will lead to fewer TLP. Thus, the points in the design space form a staircase shape as shown by Figure 11. Each stair consists of a few design points with the same TLP but different register per-thread due to the discontinues and non-convex correlation between the TLP and register per-thread [1].

Each point in the design space exhibits tradeoff between the TLP and single-thread performance. More clearly, high register per-thread increases the single-thread performance but leads to low TLP. The design space is so huge that it is infeasible to explore it exhaustively. Hence, we employ two pruning strategies for efficiency. First, for two points a and b,

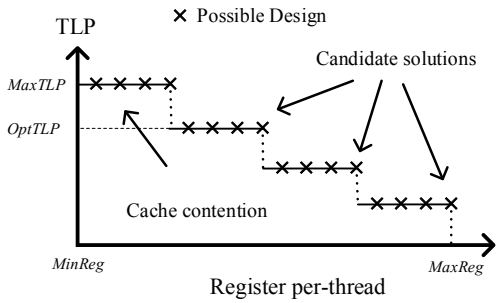


Figure 11: Design Space.

if a has higher register per-thread than b, but the same TLP with b, then a is guaranteed to be better than b. Thus, we can safely remove b without consideration. In other words, we only need to consider the rightmost point on each stair. Second, we discard the points whose TLP is higher than the *OptTLP* as they will cause serious cache contention. The remaining points are the candidate solutions, and they will be further optimize and evaluated after the register allocation.

5. REGISTER ALLOCATION

Register allocation is carried out for each candidate solution left by the design space pruning.

5.1 Implementation

CRAT is implemented based on GPGPU-Sim compilation system. GPGPU-Sim mimics the thread execution at the PTX level, which is designed to be close to the machine code [9]. On real hardware, a platform-specific just-in-time (JIT) compiler will be called to translate the PTX code into the machine code.

However, PTX uses static single assignment(SSA) style, which does not perform register allocation. It assumes an infinite register set, each time a new variable is generated, it is assigned to a new register. List 1 shows an example of CUDA kernel and List 2 shows the corresponding compiled PTX code. The original CUDA code in List 1 simply computes the global thread identifier. In the PTX code shown in List 2, it first reads data from built-in registers to registers $r0$, $r1$, and $r2$. Then, $r3$ is created to hold the result of the multiplication operation. Finally, $r4$ is created to store the value of tid. In total, five registers are required in this example. However, not all the variables are live at the same time. When a variable is dead, we can reuse the corresponding register. For the variables that are not live at the same time, they can be assigned to the same register. With register allocation, only three registers are needed as shown by Listing 3.

We extend GPGPU-Sim compilation system with the register allocation capability. The register allocation algorithms have been widely studied [12, 13]. In this paper, we implement a Chaitin-Briggs' register allocator [10]. It consists of three main steps as shown in Figure 9. Firstly, it analyzes the live range of each variable and constructs the interference graph. Secondly, the graph coloring algorithm is invoked to allocate the variables to registers. If a variable can not be assigned to a register due to register limit, it will be spilled to memory. Finally, spill codes are inserted if necessary. In our implementation, we spill the variables to local memory using load and store operations and optimize the spilling to shared memory when possible (Algorithm 1). If we assume only two 32-bit registers are available, the kernel in List 3 has to be spilled. List 4 shows the spilled kernel. The $SpillStack$, which stores the spilled variables, is declared as an array in the local memory. A 64-bit addressing register $d0$ is required to store the base address of $SpillStack$ as PTX ISA does not support displacement addressing mode. Spill codes are inserted when there exists register pressure. For example, when $r2$ is generated, $r0$ is spilled to $SpillStack$ using store instruction $st.local.u32$ to free a register. When $r2$ is dead and $r0$ is referenced again, $r0$ is loaded from local memory to register file using load instruction $ld.local.u32$.

```
1  __global__ void kernel(int*output){
2     //Thread identifier computation
3     int tid = threadIdx.x +
4     blockIdx.x*blockDim.x;
5  }
```

Listing 1: Original CUDA kernel.

```
1  .entry kernel(...)
2  {
3     mov.u32 %r0, %tid.x;
4     mov.u32 %r1, %ctaid.x;
5     mov.u32 %r2, %ntid.x;
6     mul.lo.u32 %r3, %r2, %r1;
7     add.u32 %r4, %r0, %r3;
8  }
```

Listing 2: Native PTX kernel.

```
1  .entry kernel(...)
2  {
3     mov.u32 %r0, %tid.x;
4     mov.u32 %r1, %ctaid.x;
5     mov.u32 %r2, %ntid.x;
6     //dead:%r1,%r2, generate:%r1
7     mul.lo.u32 %r1, %r1, %r2;
8     //dead:%r0,%r1
9     add.u32 %r0, %r0, %r1;
10 }
```

Listing 3: PTX kernel with register allocation.

```
1  .entry kernel(...)
2  {
3     //Addressing register
4     .reg .u64 %d<1>;
5     //Spill stack in local memory
6     .local .align 4 .b8 SpillStack[4];
7     mov.u32 %r0, %tid.x;
8     mov.u32 %r1, %ctaid.x;
9     mov.u64 %d0, SpillStack;
10    st.local.u32 [%d0], %r0;
11    mov.u32 %r0, %ntid.x;
12    mul.lo.u32 %r1, %r1, %r0;
13    ld.local.u32 %r1, [%d0];
14    add.u32 %r0, %r0, %r1;
15 }
```

Listing 4: PTX kernel with spill codes.

5.2 Validation

We first verify that the executions with and without register allocation are consistent in GPGPU-Sim framework. Then, we validate our register allocation by comparing to the built-in PTX assembler provided by the *nvcc* tool-chain on commercial hardware. More clearly, we perform register allocation on the original PTX code using our *CRAT* framework and compare it with the binary code on GTX680, which is generated by *nvcc* compiler. Figure 12 compares the number of spill load/store bytes for application *CFD*. The number of spill load/store bytes can be collected through profiling on GTX680. In general, the number of spill bytes is very similar except when *Reg=32* and *Reg=35*. There are several reasons for this discrepancy. First, the register allocation algorithms might be different. Second, PTX ISA is type-sensitive. Each instruction has its type specification. The type of the instruction must match with the type of the registers. For example, a 32-bit instruction can only operate on 32-bit registers. When a variable dies, the corresponding register could not be assigned to a variable with different type. This may lead to register waste.

Note that, we do not attempt to implement a register allocator that perfectly matches with the commercial compilers. In fact, as part of the coordinated register allocation and thread-level parallelism optimization framework, we desire to implement an efficient register allocator that can allocate registers to variables given a per-thread register limit.

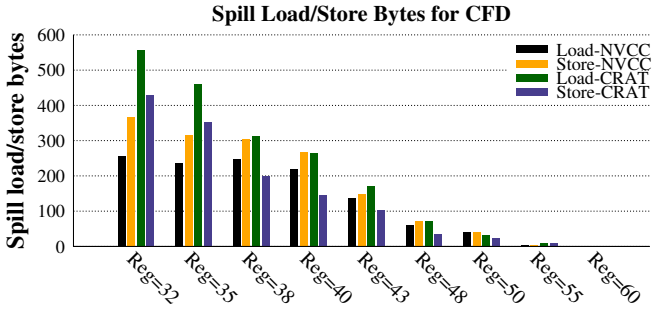


Figure 12: The comparison of *nvcc* 4.2 and *CRAT* for spill load/store bytes.

5.3 Spilling Optimization

By default, register allocation spills variables to local memory after live range analysis and register coloring as shown in Figure 9. However, off-chip local memory is much slower than on-chip shared memory [14]. More importantly, as shown in Section 2, most of the applications fail to utilize all the shared memory space. This opens up the opportunities for spilling using shared memory instead of local memory. *CRAT* employs a spilling optimization that attempts to spill variables to shared memory(Algorithm 1). Excessive use of shared memory may hurt the TLP, our spilling optimization ensures that the TLP is not changed and only utilizes the spare shared memory for spilling.

Algorithm 1 presents the details of our spilling optimization. The input to Algorithm 1 are the spill stack ($Stack[]$) and the size of spare shared memory ($SpareShmSize$). We compute the $SpareShmSize$ using those parameters collected in Section 4. Spill stack is the array used to store the spilled variables. Originally, spill stack is allocated in local memory. In Algorithm 1, we split the spill stack into N smaller sub-stacks and try to allocate them into shared memory. Smaller sub-stacks are easier to fit in shared memory than the entire stack. The value N is determined by the splitting method. Here, we split the spill stack according to the data type and the width of the spilled variables. For example, all the integer variables with 32-bit width are spilled to the same sub-stack. Alternative split methods may lead to different result, we leave it as future work.

Algorithm 1 Spilling Optimization Algorithm.

```
Input: Stack[], SpareShmSize;
1:  ▷ Spill stack split.
2:  subStack[N], subStackSize[N] ← split(Stack[]);
3:
4:  ▷ gain estimation.
5:  gain[N] ← 0;
6:  for i ← 1 to N do
7:      for each spill inst do
8:          if inst accesses subStack[i] then
9:              gain[i] + +;
10:         end if
11:     end for
12: end for
13:
14: ▷ knapsack problem formulation.
15: S[N, SpareShmSize] ← 0, Mask[N, SpareShmSize] ← 0
16: for i ← 1 to N do
17:     for v ← 0 to subStackSize[i] do
18:         S[i, v] ← S[i - 1, v]
19:     end for
20:     for v ← 0 → SpareShmSize do
21:         S[i,v], Mask[i,v] ← Max(S[i - 1,v],S[i - 1,v - subStackSize[i]] + gain[i])
22:     end for
23: end for
Output: ← Mask[N - 1,SpareShmSize]
```

For each sub-stack, we scan the kernel and record the number of accesses to it using array $gain[]$ (Line 4-12). Then, we formulate a 0-1 Knapsack problem (Line 14-23). Each sub-stack can either be spilled to shared memory or not. The goal of the optimization is to minimize the number of accesses to local memory. If we place sub-stack[i] into shared memory, we estimate its benefit as $gain[i]$ (the number of access to local memory). Algorithm 1 maximizes the gain under the limitation of shared memory size. We solve this problem using dynamic programming. $S[i, v]$ denotes the maximum gain for the sub-stacks from 1 to i given shared memory size v. The maximum gain is returned by $S[N, SpareShmSize]$ and the spilled sub-stacks are returned by $Mask[N, SpareShmSize]$. Finally, we insert spill codes to the kernel using the sub-stacks.

6. OPTIMIZATION

After the design space pruning and register allocation, multiple design points may remain. We employ a performance metric to compare them and select the optimal one. The performance of GPU applications depends on the single-thread performance and the TLP. Obviously, every design point (*reg*, *TLP*) impacts the single-thread performance and TLP differently. We design the metric Thread-level Parallelism and Spill Cost (TPSC) to model both of them to capture the performance tradeoff.

$$TPSC = TLP_{gain} \cdot Spill_{cost}$$

TLP_{gain} models the TLP and $Spill_{cost}$ represents the spilling cost, which indirectly models the single-thread performance. TLP_{gain} is defined as

$$TLP_{gain} = 1 - \frac{TLP \cdot BlockSize}{TLP \cdot BlocSize + MaxThread}$$

where $MaxThread$ represents the maximum allowed number of threads per SM. When TLP is sufficiently high, increasing TLP leads to diminishing effect on performance as demonstrated by prior studies [15]. TLP_{gain} reflects such trend. The $Spill_{cost}$ estimates the overhead of the inserted spill instructions as follows,

$$Spill_{cost} = Num_{local} \cdot Cost_{local} + Num_{shm} \cdot Cost_{shm} + Num_{others}$$

where Num_{local}, Num_{shm}, and Num_{others} represent the number of inserted local memory instructions, shared memory instructions, and other instructions, respectively. Other instructions refer to those extra instructions used for address computation for spilling. $Cost_{local}$ and $Cost_{shm}$ represents the delay of per access to the local memory and shared memory, respectively. $Cost_{local}$ and $Cost_{shm}$ are measured on the target architecture through micro benchmarks.

TPSC does not model the cache effect as the design points with serious cache contention are pruned(Section 4). We compare *TPSC* of different design points and select the one that yields the smallest value. Evaluations in Section 7 show that *TPSC* metric can accurately capture the tradeoff between single-thread performance and TLP.

Table 2: Simulated GPGPU-Sim configuration.

SM	15 SMs, 32 cores/SM, 700 MHz
Register	128KB, 16 banks
Shared Memory	48KB, 32 banks
TLP Limitation	1536 threads, 8 thread blocks
Scheduler	2 warp schedulers per SM, GTO
L1 Data Cache	32KB, 4-way, 128B block, LRU, 32 MSHR entries
L2 Unified Cache	768KB size, 6 banks

7. EXPERIMENTAL EVALUATION

7.1 Experiment Methodology

We implement our *CRAT* compiler framework based on GPGPU-Sim [6] compilation and runtime system (version 3.2.3+), using the Fermi-like architecture parameters shown in Table 2. We measure the power consumption using GPUWattch [16]. We evaluate *CRAT* using all the applications from Rodinia [7] and Parboil [8] suites, and some applications from NVIDIA SDK [1] suite. For each application, its tested inputs are from the original benchmark suites. For the applications that consist of more than one kernel, we only focus on the most time-consuming kernel. For all of the applications, we simulate them until all the instructions finish.

CRAT mainly targets for the resource (cache, register file) sensitive applications. For cache sensitive applications, thread throttling improves the performance by reducing the cache contention, which may lead to less register utilization. Register sensitive applications mainly are complex applications, which require large register file. For these applications, different register allocation and TLP gives different tradeoff in single-thread performance and thread parallelism. Thus, we classify the applications into two categories: resource sensitive and resource insensitive applications. If an application is sensitive to cache or register, it is classified as resource sensitive, otherwise it is classified as resource insensitive.

Table 3: Evaluated GPU Applications.

Resource Sensitive Applications			
Application	Kernel	*abbr.*	Suite
BlackScholes	BlackScholesGPU	BLK	SDK [1]
cfd	cuda_compute_flux	CFD	Rodinia [7]
dxtc	compress	DTC	SDK [1]
EstimatePi	initRNG	ESP	SDK [1]
FDTD3d	FiniteDifferences	FDTD	SDK [1]
hotspot	calculate_temp	HST	Rodinia [7]
kmeans	invert_mapping	KMN	Rodinia [7]
lbm	StreamCollide	LBM	Parboil [8]
spmv	spmv_jds	SPMV	Parboil [8]
stencil	block2D	STE	Parboil [8]
streamcluster	compute_cost	STM	Rodinia [7]
Resource Insensitive Applications			
backprop	layerforward	BAK	Rodinia [7]
bfs	kernel	BFS	Rodinia [7]
b+tree	findK	B+T	Rodinia [7]
gaussian	Fan1	GAU	Rodinia [7]
lud	diagonal	LUD	Rodinia [7]
mummergpu	mummergpuKernel	MUM	Rodinia [7]
nw	cuda_shared_1	NEED	Rodinia [7]
particlefilter	kernel	PTF	Rodinia [7]
pathfinder	dynproc	PATH	Rodinia [7]
sgemm	mysgemmNT	SGM	Parboil [8]
srad	srad_cuda	SRAD	Rodinia [7]

In Section 4, we propose to obtain the *OptTLP* through profiling or static code analysis. We first use the *OptTLP* obtained through profiling for *CRAT* implementation. In the following, we perform six sets of experiments to evaluate *CRAT*. First, we present the performance benefit of *CRAT* using the configurations in Table 2 for resource sensitive applications in Table 3. The platform is a Fermi-like architecture. GPU architectures are rapidly evolving; each new generation of GPUs are equipped with more memory resources (e.g. registers) and computing capability (e.g. threads). Second, we evaluate *CRAT* using a Kepler-like architecture to demonstrate its scalability to new architecture. Third, we present the input sensitivity study. Fourth, we present the performance results for resource insensitive applications. Fifth, we evaluate *CRAT* using the estimated optimal TLP. Finally, we report the overhead of *CRAT* framework.

7.2 Performance Results

In order to demonstrate the benefit of *CRAT*, we compare it with the following techniques.

MaxTLP. *MaxTLP* uses the default register allocation and does not employ thread throttling. The default register allocation is oblivious to thread throttling. *MaxTLP* will run as many thread blocks as possible until one or multiple resources are exhausted.

OptTLP. *OptTLP* implements the thread throttling technique at thread block level [3]. For each application, its optimal TLP is determined offline by exhaustively testing all the possible TLPs. The design space of TLP is small (e.g. 8).

CRAT-local. *CRAT-local* is exactly the same with *CRAT*, but it disables the spilling optimization. In other words, it

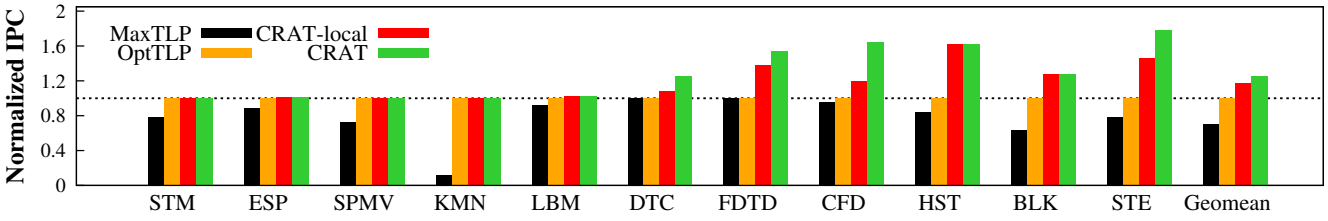


Figure 13: Performance results of the *MaxTLP*, *OptTLP*, *CRAT-local*, and *CRAT*(normalized to the *OptTLP*).

does not spill any variables into shared memory.

Figure 13 shows the performance results normalized to the *OptTLP*. Compared with *OptTLP*, the performance improvement(geometric mean) of *CRAT-local* and *CRAT* are 1.17X and 1.25X, respectively. Different applications show different performance improvements. Overall, *CRAT* performs consistently well across all the applications.

The performance benefit of *CRAT* can be attributed to three reasons: (a) explore *TLP* to reduce cache contention; (b) increase the register utilization and find the best tradeoff between register per-thread and TLP; (c) employ shared memory to reduce the register spilling cost. Next, we support our argument with quantitative experiments.

Figure 14 compares the TLP of *MaxTLP* and *CRAT*. On average, *CRAT* executes 2.6 thread blocks per SM, while *MaxTLP* executes 5.1 thread blocks per SM. For application *KMN*, *CRAT* chooses to run only 1 thread block due to serious cache contention, compared to the 6 thread blocks in *MaxTLP*. This leads to substantial performance improvement in cache. More clearly, the L1 cache hit rate is improved by 82.1% and the pipeline stall caused by resource congestion is reduced by 97.2% for *KMN*.

Compared with the thread throttling techniques (e.g. *OptTLP*), *CRAT* further increases the performance by improving the register utilization and exploring the tradeoff between single-thread performance and TLP. Figure 15 compares the register utilization of *OptTLP* and *CRAT*. For applications *STM*, *SPMV*, *KMN*, and *LBM*, the default register per-thread happens to be the optimal register allocation. Therefore, the register utilization is not improved. Consequently, for these applications, *CRAT* achieves the same performance with the *OptTLP* as shown in Figure 13. For the rest of the applications, the register utilization are all improved. On average, the register utilization is improved by 15% – 27%. For some applications, the performance improvement also comes from the balance between the single-thread performance and TLP. For example, for application *FDTD*, *OptTLP* uses 42 registers per-thread and executes only 1 thread block concurrently. This leads to 73% register utilization. *CRAT* compares two candidate solutions. One uses 52 registers per-thread to maximize the single-thread performance, and the other one uses 32 registers per-thread to maximize the TLP. Both solutions have higher register utilization, compared with *MaxTLP*. *CRAT* finally chooses to allocate 32 registers per-thread and 2 thread blocks per SM. This leads to 100% register utilization. For this case, *CRAT* improves the register utilization and finds a balance between the single-thread performance and TLP.

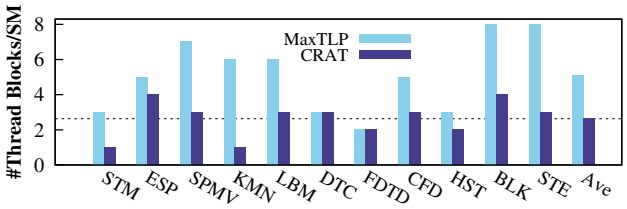


Figure 14: The selected TLP for *MaxTLP* and *CRAT*.

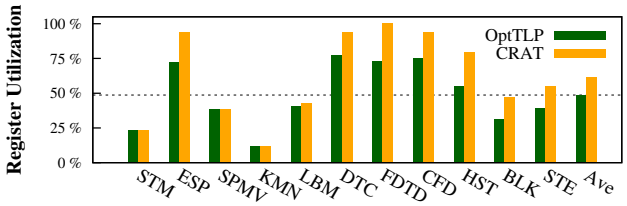


Figure 15: The register utilization of *OptTLP* and *CRAT*.

In the *MaxTLP* and *OptTLP*, applications such as *HST*, and *BLK* employ spills to local memory as they lack sufficient registers. As the register utilization is improved by *CRAT*, each thread has sufficient amount registers to hold the variables and do not incur any register spilling any more. Hence, for applications *HST* and *BLK*, *CRAT* does not improve *CRAT-local* as shown by Figure 13. However, for applications *DTC*, *FDTD*, *CFD*, and *STE*, register spilling can not be completely eliminated. Spilling optimization is an effective technique for them and *CRAT* improves *CRAT-local* as shown by Figure 13. Figure 16 shows the reduction of local memory accesses for these applications. The number of local memory accesses are reduced by 42% on average.

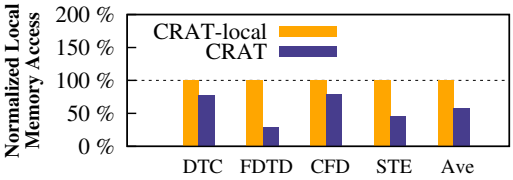


Figure 16: The normalized local memory accesses of *CRAT-local* and *CRAT*.

Energy Results. Due to the performance gain, experiments show that *CRAT* achieves on average 16.5% energy savings compared with *OptTLP*.

7.3 Architecture Scalability

In the previous subsection, we evaluate *CRAT* on a Fermi-like architecture [17] using the configuration in Table 2. However, GPU architecture rapidly evolves, new generation of GPUs will be equipped with more resource. For example, compared to Fermi, Kepler architecture increases the size of register file from 128KB to 256KB per SM and the maximum number of concurrently executing threads from 1536 to 2048 per SM. We update the parameters in Table 2 and evaluate *CRAT* using the Kepler configuration. Figure 17 gives the normalized performance. On average, we achieve 1.32X performance speedup compared with the *OptTLP*.

Some of applications such as *LBM*, *FDTD*, and *CFD* show smaller improvement on Kepler than Fermi. This is because the larger register file on Kepler alleviates the register pressure. We also notice that applications *SPMV*, *HST*, *BLK*, and *STE* show better performance speedup. The reason behind this is two-folds. First, as more threads are allowed to execute concurrently, cache contention become even worse. As a result, the gap between the optimal and maximum TLP becomes larger, leaving more registers under-utilized by pure thread throttling. Second, the design space of register per-thread and TLP continues to grow with the resource limits. This leads to large room for our coordinated approach.

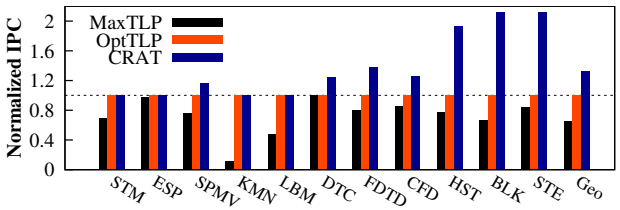


Figure 17: The performance speedup of *CRAT* on a Kepler-like GPU.

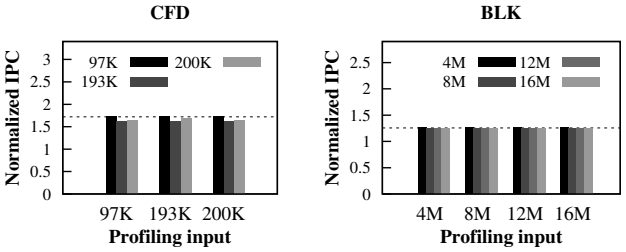


Figure 18: Achieved performance speedup of *CRAT* across different inputs.

7.4 Input Sensitivity

In the above experiments, we use the *OptTLP* obtained through profiling for *CRAT* implementation and use the same input for profiling and evaluation. Here, we evaluate the performance against different inputs. All the inputs are chosen from the original benchmark suites. For each input, we use it as profiling input and test it across all the inputs.

We use applications *CFD* and *BLK* for this study. For each application, we use 3-4 different inputs. First of all, for every application, different profiling inputs lead to the same *OptTLP*. Though different inputs vary with threads/input size, the behaviors of different thread blocks in one application tend to be stable. Therefore, the *OptTLP* is likely to be the same across different inputs. Figure 18 shows the performance speedup for all the inputs. *CRAT* achieves consistently good results. The speedup of different inputs could be different due to different workload.

7.5 Insensitive Applications

Figure 19 shows the performance results for resource insensitive applications. The performance is normalized to the *OptTLP*. In general, these applications do not face the cache contention and register pressure problem. Hence, the default *MaxTLP* and register allocation is a good solution for them. As a result, neither *OptTLP* nor *CRAT* has remarkable improvement.

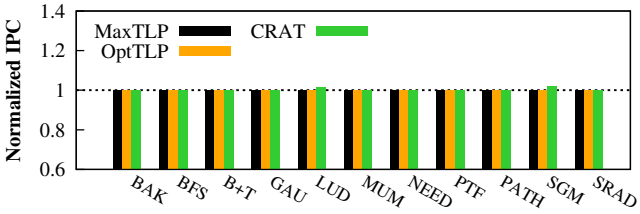


Figure 19: Performance results for resource insensitive applications.

7.6 Evaluation of Estimated OptTLP.

For the above experiments, *CRAT* uses the truly *OptTLP* obtained through profiling. In Section 4, we also propose a code analysis technique that estimates the *OptTLP* statically. *CRAT* implemented using static analysis (profiling) is termed as *CRAT-static* (*CRAT-profile*). Figure 20 show the performance comparison. As shown, *CRAT-static* achieves comparable performance to *CRAT-profile* (1.22X vs 1.25X).

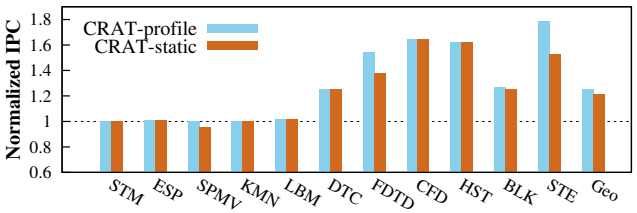


Figure 20: Performance Comparison.

7.7 Overhead

The overhead of *CRAT* consists of two parts, the computation of *OptTLP* and the design space exploration. The overhead of design space exploration is so small that can be ignored. If we use profiling to obtain *OptTLP*, this requires to run each application a few times (average 5, maximal 8). For all the applications we study, the average profiling overhead is about 1.8 hours using GPGPU-Sim and 1.94 millisecond on real GPUs. If we use static code analysis to obtain *OptTLP*, the average overhead is only 1 millisecond.

8. RELATED WORK

To guide the optimization of GPU applications, analytical models [11, 15, 18, 19] are proposed. Optimization techniques, such as warp schedulers [20, 21], concurrency polices [22, 23], divergence optimization [24, 25], data prefetching [26], loop optimization [27], hardware customization [28, 29], and memory managements [30, 31, 32, 33] are developed, too.

Thread throttling techniques are proposed to mitigate the cache contention problem brought by massive threading. Rogers et al. develop a cache-concious wavefront scheduling (*CCWS*) to capture the intra-warp data locality [2]. *CCWS* monitors the early evictions caused by cache contention and limit the number of active warps to mitigate the contention problem. They also propose a divergence-aware warp scheduling(*DAWS*) [34]. Kayıran et al. propose to pause some of the thread blocks to alleviate the cache pressure dynamically [3]. Lee et al. employ thread throttling in the thread block scheduling and concurrent kernel execution [5]. However, all these thread throttling techniques may lead to under utilization of register file. Cache bypassing techniques that selectively filter data request can also improve the cache performance. Both static [35, 36] and dynamic [37, 38] approaches are proposed. Recently, to further improve the performance, Chen et al. [4] and Li et al. [39] coordiate cache bypassing with thread throttling. Our *CRAT* framework can be used together with cache bypassing techniques to further improve the cache performance.

The register allocation on CPUs have been widely studied [12, 10, 13]. However, those techniques do not necessarily work well for GPUs. Gebhart et al. observe that the register file consumes a large part of the power budget of the whole GPU [40]. They propose a multi-level register file to minimize the energy consumption. Registers that are frequently referenced are allocated to a energy-efficient register file cache. They also develop unified on-chip memories, the partition of which can be reconfigured according to the application requirements [41]. Register file energy optimizations are also discussed in [42]. They propose to power gate inactive registers to reduce both static and dynamic power leakage. Lakshminarayana et al. propose to utilize the spare registers to prefetch data for graph algorithms [43]. Gilani et al. develop a new register file structure to optimize both the performance and energy consumption [44]. *CRAT* can work with these techniques through coordinately optimizing the register allocation and TLP together. Hayes et al. develop a unified on-chip memory allocation technique, however, they do not explore register allocation together with thread throttling [45].

Data placement techniques aim to swap the variables among different memories such as shared memory and register file. Chen et al. introduce a memory specification language to guide the data placement [46]. Li et al. develop an automatic data placement algorithm which enables cross-platform data management [47].

9. CONCLUSION

GPUs are becoming popular to accelerate applications in high-performance heterogeneous systems. However, tuning GPU applications to fully exploit the hardware resource is never a trivial task. Comprehensive architecture understanding and scalable tuning techniques are urgently needed to keep pace with the rapid evolvement of applications and architectures.

In this paper, we develop the *CRAT* compiler framework to coordinatedly optimize the register allocation and thread-level parallelism(TLP) for GPUs. *CRAT* maintains the TLP that does not cause cache contention and efficiently utilize the register file saved by thread throttling to improve single-thread performance. *CRAT* also balances the tradeoff between the single-thread performance and TLP. We present detailed characterizations of GPU workloads to demonstrate the considerable performance potential using the coordinated solution. We conduct systematic evaluations of *CRAT* using a variety of GPU applications. *CRAT* improves the performance speedup up to 1.79X(1.25X geometric mean) compared to thread throttling techniques.

Acknowledgement. This work was partially supported by the National Science Foundation China (No. 61300005) and National High-Tech Research & Development Program of China (2015AA01A301). We thank all the anonymous reviewers for their feedback.

10. REFERENCES

[1] "NVIDIA CUDA programming guide.." `http://docs.nvidia.com/cuda/cuda-c-programming-guide`.

[2] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pp. 72–83, 2012.

[3] O. Kayıran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: Optimizing thread-level parallelism for GPGPUs," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pp. 157–166, 2013.

[4] X. Chen, L.-W. Chang, C. Rodrigues, J. Lv, Z. Wang, and W. mei Hwu, "Adaptive cache management for energy-efficient GPU computing," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pp. 343–355, 2014.

[5] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Improving GPGPU resource utilization through alternative thread block scheduling," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture*, HPCA'14, pp. 260–271, 2014.

[6] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS'09, pp. 163–174, 2009.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization*, IISWC'09, 2009.

[8] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei Hwu, "The IMPACT Research Group, Parboil Benchmark Suite." `http://impact.crhc.illinois.edu/parboil/parboil.aspx`.

[9] "NVIDIA PTX ISA document.." `http://docs.nvidia.com/cuda/parallel-thread-execution`.

[10] P. Briggs, "Register allocation via graph coloring," *Ph.D. thesis, Department of Computer Science, Rice University*, 1992.

[11] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *2009 36th Annual International Symposium on Computer Architecture*, ISCA'09, pp. 152–163, 2009.

[12] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Computing Languages*, vol. 6, no. 1, pp. 47–57, 1981.

[13] M. Poletto and V. Sarkar, "Linear scan register allocation," *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 5, pp. 895–913, 1999.

[14] C. Li, Y. Yang, H. Dai, S. Yan, F. Mueller, and H. Zhou, "Understanding the tradeoffs between software-managed vs. hardware-managed caches in GPUs," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS'14, pp. 231–242, 2014.

[15] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for GPU architectures," in *2010 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'10, pp. 105–114, 2010.

[16] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling energy optimizations in GPGPUs," in *2013 40th Annual International Symposium on Computer Architecture*, ISCA'13, pp. 487–498, 2013.

[17] "NVIDIA Fermi architecture.." `www.nvidia.com/object/fermi-architecture.html`.

[18] Z. Cui, Y. Liang, K. Rupnow, and D. Chen, "An accurate GPU performance model for effective control flow divergence optimization," in *2012 IEEE 26th International Parallel Distributed Processing Symposium*, IPDPS'12, pp. 83–94, 2012.

[19] X. Chen and T. Aamodt, "A first-order fine-grained multithreaded throughput model," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, HPCA'09, pp. 329–340, 2009.

[20] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU performance via large warps and two-level warp scheduling," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pp. 308–317, 2011.

[21] A. Jog, O. Kayıran, C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "OWL: Cooperative thread array aware scheduling techniques for improving GPGPU performance," in *2013 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'13, pp. 395–406, 2013.

[22] Y. Liang, H. Huynh, K. Rupnow, R. Goh, and D. Chen, "Efficient GPU spatial-temporal multitasking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, pp. 748–760, March 2015.

[23] O. Kayıran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, "Managing GPU concurrency in heterogeneous architectures," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pp. 114–126, 2014.

[24] W. Fung, I. Sham, G. Yuan, and T. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *2007 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-40, pp. 407–420, 2007.

[25] A. ElTantawy, J. Ma, M. O'Connor, and T. Aamodt, "A scalable multi-path microarchitecture for efficient GPU control flow," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture*, HPCA'14, pp. 248–259, 2014.

[26] A. Jog, O. Kayıran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated scheduling and prefetching for GPGPUs," in *2013 40th Annual International Symposium on Computer Architecture*, ISCA '13, pp. 332–343, 2013.

[27] Y. Yang and H. Zhou, "CUDA-NP: Realizing nested thread-level parallelism in GPGPU applications," in *2014 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pp. 93–106, 2014.

[28] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović, "Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators," in *2011 38th Annual International Symposium on Computer Architecture*, ISCA '11, pp. 129–140, 2011.

[29] P. Xiang, Y. Yang, and H. Zhou, "Warp-level divergence in GPUs: Characterization, impact, and mitigation," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture*, HPCA'14, pp. 284–295, 2014.

[30] H. Lee, K. Brown, A. Sujeeth, T. Rompf, and K. Olukotun, "Locality-aware mapping of nested parallel patterns on GPUs," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pp. 63–74, 2014.

[31] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU compiler for memory optimization and parallelism management," in *2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pp. 86–97, 2010.

[32] B. Jang, D. Schaa, P. Mistry, and D. Kaeli, "Exploiting memory access patterns to improve memory performance in data-parallel architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, pp. 105–118, Jan 2011.

[33] G. Yuan, A. Bakhoda, and T. Aamodt, "Complexity effective memory access scheduling for many-core accelerator architectures," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-42, pp. 34–44, Dec 2009.

[34] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-aware warp scheduling," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pp. 99–110, 2013.

[35] X. Xie, Y. Liang, G. Sun, and D. Chen, "An efficient compiler framework for cache bypassing on GPUs," in *2013 International Conference on Computer-Aided Design*, ICCAD '13, pp. 516–523, 2013.

[36] Y. Liang, X. Xie, G. Sun, and D. Chen, "An efficient compiler framework for cache bypassing on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, pp. 1677–1690, Oct 2015.

[37] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang, "Coordinated static and dynamic cache bypassing for GPUs," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture*, HPCA'15, pp. 76–88, 2015.

[38] W. Jia, K. Shaw, and M. Martonosi, "MRPB: Memory request prioritization for massively parallel processors," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture*, HPCA'14, pp. 272–283, 2014.

[39] D. Li, M. Rhu, D. Johnson, M. O'Connor, M. Erez, D. Burger, D. Fussell, and S. Redder, "Priority-based cache allocation in throughput processors," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture*, HPCA'15, pp. 89–100, Feb 2015.

[40] M. Gebhart, S. W. Keckler, and W. J. Dally, "A compile-time managed multi-level register file hierarchy," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pp. 465–476, 2011.

[41] M. Gebhart, S. Keckler, B. Khailany, R. Krashinsky, and W. Dally, "Unifying primary cache, scratch, and register file memories in a throughput processor," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pp. 96–106, 2012.

[42] M. Abdel-Majeed and M. Annavaram, "Warped register file: A power efficient register file for GPGPUs," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture*, HPCA '13, pp. 412–423, 2013.

[43] N. Lakshminarayana and H. Kim, "Spare register aware prefetching for graph algorithms on GPUs," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture*, HPCA'14, pp. 614–625, Feb 2014.

[44] S. Z. Gilani, N. S. Kim, and M. J. Schulte, "Exploiting GPU peak-power and performance tradeoffs through reduced effective pipeline latency," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pp. 74–85, 2013.

[45] A. B. Hayes and E. Z. Zhang, "Unified on-chip memory allocation for simt architecture," in *2014 ACM 28th International Conference on Supercomputing*, ICS '14, pp. 293–302, 2014.

[46] G. Chen, B. Wu, D. Li, and X. Shen, "PORPLE: An extensible optimizer for portable data placement on GPU," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pp. 88–100, 2014.

[47] C. Li, Y. Yang, Z. Lin, and H. Zhou, "Automatic data placement into GPU on-chip memory resources," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, 2015.