Orchestrating On-Chip Memory Resources for Throughput-Oriented Compilation

Abstract

A key factor in GPU performance efficiency is the number of **active** *threads that can run simultaneously on each streaming multi-processor. The active threads have their states saved on fast memory devices and can quickly be scheduled to run if the set of running threads stalls due to memory latency. The greater number of* **active** *threads we have, the higher utilization we can obtain from many-core processor pipelines. To achieve optimal utilization, we typically need many more* **active** *threads than the number of physical cores. Due to limited on-chip memory resources including registers and scratch-pad memory, and the fact that every thread gets a equal partition of on-chip memory resource, the number of* **active** *threads depends on the characteristics of a given program and the back-end compilation efficiency in resource allocation. When a large and complicated program requires more registers per thread, the program performance may degrade significantly due to the decrease in the total number of* **active** *threads. In this paper, we propose a novel resource allocation approach for back-end compilation of throughput GPU processors. This approach leverages on-chip scratch-pad memory to reduce register pressure and increase GPU processor occupancy for maximum throughput. The scratch-pad memory serves as middle layer between register and long-latency off-chip memory. On one hand, it reduces register usage per-thread. On the other hand, it can serve as a caching layer for variables that need to be staged into registers from global memory. We have formulated the resource allocation problem for optimal utilization and throughput of many-core processors, and proposed efficient models and techniques. We implemented these techniques in a binary optimizer, and evaluated it on a set of realistic benchmarks on real GPUs. We demonstrated the effectiveness of our techniques by achieving up to 1.65 times speedup compared to the programs compiled by* nvcc *with highest optimization flag.*

1. Introduction

General Purpose Graphic Processing Unit (GPGPU) has become a prevailing computing platform due to its massive computing horsepower, cost-efficiency, and power-efficiency, and ease of programmability with C/C++ like programming models such as CUDA and OpenCL. The key factor of GPU application efficiency is the number of *active* threads that can run simultaneously on every streaming multi-processor. Conventionally, the states of these threads are saved in registers to enable fast threads scheduling for memory latency hiding purpose. These *active* threads reside on every streaming multiprocessor until they have finished their tasks and another batch of *active* threads are ready to switch in. The size of a batch of *active* threads is typically smaller than the total number of threads that are dispatched to run one GPU function. Once a batch of *active* threads have finished their execution, another batch of active threads will have their states loaded into registers and start executing. Different batches of active threads run sequentially. This process repeats until all dispatched threads complete their tasks. The number of active threads within a batch for a given program depends on hardware constraints such as schedule resources. It first cannot exceed the maximum number of active threads the scheduler can handle at one time. On the other hand, it depends on the on-chip memory resource requirement by the program itself. If every thread requires many registers, the number of *active* threads within the same batch will be very limited. Consequently, there may not be enough concurrency to help threads hide memory latency for each other and the utilization of many-core GPU processor may be degraded significantly.

In fact, even a small change in the number of registers per thread, may lead to large variation in performance. We show an example in Fig. 1. We conducted experiments by varying the number of registers every thread really needs and can use for the matrix multiplication program in NVIDIA GPU Computing SDK. We perform various levels of loop unrolling to vary the number of registers needed. We control the number of registers every thread can use by adding "-maxrregcount" flag to *nvcc* - the NVIDIA GPU compiler. We show two compiled versions of the program execution at every loop unrolling level. One is the default version with the highest optimization level. It is referred to as the version "without register limit". We mark every point with the actual number of registers used by compilation with the highest optimization level. In the other version, we control the number of registers per-thread to be no more than 32. It is referred to as the version "with register limit". The y-axis shows the speedup of both versions against the original program without loop unrolling. We can see clear patterns in Fig. 1 no matter which loop unrolling level we use. Whe can see that the version with register number controlled under 32 is always better than the version without any register number control even if there are only two registers more at unrolling levels of 8, 16, 20 and 28. At unrolling level 16, the speedup number is 1, which means the benefits of loop unrolling is completely offset by the performance degradation brought by only two extra registers

Figure 1: Motivation Example: Matrix Multiplication

used per-thread. When the number of registers is below 32, the changes do not make a big difference. The reason is that 32 registers per-thread makes the total number of *active* threads hit the maximum number of threads the hardware scheduler can handle.

Intuitively, to ensure a large number of *active* threads that can occupy GPU processor pipelines, we use register spilling approach to control the number of registers per-thread to be a reasonable amount. We off-load these live variables into large-latency off-chip memories and load them back to registers before their immediate usage in the corresponding instructions. However, by doing this, we fail to use the most appealing feature of GPU processors, which is fast *"context switches"* [1] among different groups of threads. When every group of threads within the same batch have their states saved in registers, they can be switched into processors or out of processors quickly. If the states of these threads are not completely saved in registers, some of which saved into off-chip memory, the *switches* between these threads may become much slower and therefore they may not be help to help hide memory latency in the processor pipelines for each other. Therefore these processor cores may not have 100% utilization and every thread in every core may in effect run in a sequential manner. It is worth mentioning that the performance of single GPU core is not comparable to that of a single CPU core even with the same frequency. It is mainly because of the simplified design of throughput processors which emphasize on more computing units, rather than complicated control units such as out-of-order execution units, sophisticated branch predictors and multiple instruction issuing units.

Due to this challenge, when the complexity of a program grows and when it requries more registers to save live variables in the middle of execution, it may be divided into many small sub-tasks. And explicit barriers are inserted between theses subtasks. Hence every sub-task is small enough to enable maximum number of *active* threads to occupy GPU processors. However, this approach introduces non-trivial overhead because of explicit synchronizations. The synchronization time may be comparable to the execution time of every sub-task.

In this paper, we propose a novel approach that employs on-chip scratch-pad memory as a secondary layer of storage for live variables in a program. By doing this, we can off-load register pressure and meanwhile keep taking advantage of fast *switches* between threads since the speed of scratch-pad memory is comparable to the speed of L1 caches. The on-chip scratch-pad memory is manageable by software as much as register memories. However, it remains most of the time idle and even when it is used, it is used for a limited time period of the whole program execution phases. Programmers, if use the scratch-pad memory (typically for explicit data caching purpose), have to manage the data explicitly to keep data consistency across memory hierarchies. We purpose to use a combination of register memory and scratch-pad memory to store live variables when a program is running, and use automatic compilation approach to control scratch-memory usage.

Using this approach, we have several obstacles laid ahead and we need to answer some questions before we can apply the approach efficientlyl. First and foremost, how can we partition the live variables to be saved directly in registers all the time and to be off-loaded into scratch-pad memory when necessary? Secondly, does the occupancy always need to hit the maximum scheduling capacity of GPU hardware? For instance, some programs are computation intensive and instruction level parallelism is enough for memory latency hiding purpose. If we off-load live variables from registers into scratch-pad memory, we may add extra overhead while not able to get any potential improvement due to the fact that increase occupancy does not help. Thirdly, even we are able to determine the best partition and the best occupancy level, how can we implement them and enforce the new resource allocation scheme in real GPU benchmarks? The back-end compilation component for commodity GPUs are proprietary to every vendor. The ABIs are not disclosed and it is nearly impossible to modify the register allocation component in back-end compilation component need to mention add a new layer of on-chip memory storage for live variables. Moreover, there has been very limited research on the topic of optimizing on-chp memory usage for the maximum occupancy of massively parallel GPU processors with compilation support. Most work in the past decades has been on CPU register spilling for off-loading live variables from registers into off-chip memory when there are not enough physical registers to store all the live variables [14] [2] [4] [1]. These studies mainly focus on instruction level parallelism such as [14] and [9] Some study on GPU exception handling is relevant to the topic of GPU program register usage analysis but not on optimizing physical register usage [11]. Some other GPU architecture work proposed hybrid SRAM and DRAM

[1]The "context switch" here is not the same concept as the traditional context switch on CPUs, which requires kernel and user model switch in operating systems. It means scheduling ready batches of *activ* threads to run on a given many-core streaming processor cores by hardware.

support to increase the number of bits that can be saved in the same die area [15]. Another work on GPU architecture research proposed hierarchical register file for energy saving purposes.

In this paper, we propose new on-chip memory resource allocation model and techniques for maximizing the occupancy of many-core processor pipelines. With increased occupancy, we can efficiently improve the utlization of massively parallel GPU processors and harness the real power by minimizing the gap between actual program throughput and peak throughput, which is currently considerable even for regular parallizable applications. To the best of our knowledge, this is the first work that addresses register pressure problem on throughput processors. The contributions of this paper are summarized as follows:

- We propose a novel approach that combines register memory and scratch-pad memory for thread state storing purposes, which on one hand maintains fast thread *switch* feature and on the other hand optimizes the occupancy of every GPU processor for best throughput.
- We formulate the problem of how to choose best register and scratch-pad memory allocation for overall whole processor occupancy. We propose efficient models and techniques. We are able to choose optimal partition between register memory and scratch-pad for a given program under various scenarios.
- We have implemented these techniques and evaluated them on a set of realistic GPU applications on real GPUs. We overcame the black-box issue of non-disclosed details in GPU architecture instruction set architecture (ISA) and application binary interface (ABI). We developed a binary transformation framework and observed significant performance improvement.

The rest of the paper is organized as follows: Section 2 describes background for GPU programming model and traditional CPU register allocation techniques; Section 3 formulates the problem of reducing register pressure through an extra layer of scratch-pad memories, and provides a model with discussion on solutions under different possible scenarios. Section 4 discusses implementation of techniques for real GPU programs. In Section 5, we present the evaluation results. Section 6 describes related work and Section 7 summarizes the contributions of the paper.

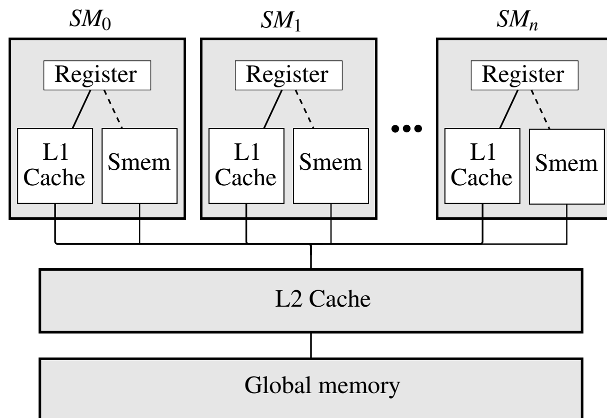
2. Background

2.1. GPU architecture and programming model

A GPU card consists of a set of vector processors, each of which is called Streaming Multi-Processors (SMP). [2] A group of threads that runs in one SMP in parallel is called a thread warp. A thread warp follows Single Instruction Multiple Data (SIMD) execution model. A warp is the minimal execution and scheduling unit for a SMP. When one warp is switched out of the SMP, if there are ready warps, another warp will be switched in to run. A number of warps are further organized into blocks. A number of blocks are further organized into a grid. A grid of threads execute a single function. A function in CUDA is called a *kernel* function.

[2] Without loss of generality, we will use NVIDIA Compute Unified Device Architecture (CUDA) terminology to describe GPU architecture throughout this paper.

Figure 2: A Typical GPU Memory Hierarchy



There are two types of memories on a GPU card. One is on-chip memory and the other is off-chip memory. On-chip memory includes registers, scratch-pad memory, and caches. Register is the fastest on-chip memory storage. Every SMP has a large register file and it is divided evenly into multiple partitions. Every partition maps to a single GPU thread. For thread warps within the same batch of *active threads*, their states are saved in their own register partition of the large register file and if they don't get enough registers, they may use off-chip memory. Only thread warps within the same batch of *active* threads can be scheduled to run to help memory latency of each other. The number of active threads is typically much larger than the number of physical cores for desirable utilization.

Programmers need to explicitly read from and write to scratch-pad memory. Scratch-pad memory is fast with the latency of only a few cycles. Its aggregated throughput can be up to 1+TB/s, comparable to L1 cache [10]. It is referred to as *shared memory* in CUDA and we will use the term throughout this paper. Every thread also has an equal shared of *shared memory* similar to registers. Off-chip memories include *global memory* and *local memory*, which takes around 400-800 cycles for a single transaction. There are also other types of off-chip memories such as *constant* and *texture* memory, which has the same latency as *global memory*, but for different purposes. *Local memory* is only used for register spilling purpose.

In Fig. 2, we show the memory hierarchy of a typical GPU. There are four levels of memory storages. The first level is the registers, which has smallest latency. The second level is L1 cache and shared memory, which is private to every SMP. The third level is a shared L2 level across different SMPs. For some GPUs, there is no shared L2 cache and therefore L1

cache is the last level cache. The next level is off-chip global memory. Conventionally, there are data paths between L1 and L2 caches, L1 cache and registers. The solid lines in Fig. 2 show the path of between off-chip memory and caches and registers. The dotted line show the path we created between registers and shared memory.

2.2. Register Allocation and Spilling on CPUs

Register allocation is a process for assigning different variables into a finite set of physical registers. At every given instruction, there are live variables that need to be used for current or future instructions. These variables are typically saved in physical registers. If there are not enough physical registers, we spill these variables into off-chip memory, and load them back into registers before their use. The purpose of register allocation is to minimize the number of spill loads and spill stores. There has been rich literature on register allocation and spilling in the past decades. In general, the problem of register allocation can be modeled as a graph coloring problem [5]. The general graph coloring problem is *NP*-complete [5] [8]. There have been several classical heuristics proposed in the past [3]. [12] and used in back-end compilation techniques.

3. Problem Formulation and Modeling

We define our problem as follows. Given a program, its total register demand and shared memory demand per thread, as well as hardware constraints on registers and shared memory, we would like to find a partition of the live variables such that part of them keep residing in registers and the rest reside in shared memory before being loaded into registers. To better formulate this problem, we would like to propose some definitions and terminology. First, we define the term of *variable*. Given a low level intermediate representation of a program such as three address code:

MOV R0, R10;

We define the operands in this instruction as *variables* and will use this term throughout this paper. Assume N_{rReq} is the minimal number of registers needed per thread if we do not spill any variables into local memory, N_{sReq} is the required number of shared memory variables (assume every shared memory variable have the same size as every register variable). N_{rTotal} is the total number of physical registers on every SMP and N_{sTotal} is the total number of variables that can reside in shared memory on every SMP. Assume the block size is *BlockSize*. With this information, we can determine how many threads can be active at one time on every streaming multiprocessor, which is also the total number of threads that can run simultaneously to help hide memory latency for each other. According to the GPU hardware thread scheduling policy, the number of active threads for this program that can run on one streaming multi-processor at one time is:

$$\lfloor \min(N_{rTotal}/N_{rReq}, N_{sTotal}/N_{sReq})/\text{BlockSize} \rfloor * \text{BlockSize}$$

The parameter *BlockSize* is introduced here since every thread block can run on only one streaming multi-processor at a time. Threads within a block can communicate with each other using shared memory (on-chip scratch-pad memory), which is private to every streaming multi-processor. One streaming processor can't access the scratch-pad memory of another. One thread block cannot be split to run on different streaming multi-processors.

With the formula above, we can see the relationship between the number of registers, the amount of shared memory and the number of active threads. The fewer on-chip memory resources that are used, the larger the number of active threads is. However, for example, if we assign too few registers per thread, we may have to force variables to be spilled into local memory, which adds more expensive memory operations. On the hand, the more on-chip memory resources a program assigns to every thread, the smaller the number of active threads is. Let's define occupancy as the ratio of the number of active threads and the maximum number of active threads supported by the hardware:

$$Occupancy = \frac{Num.Active.Threads}{Maximum.Num.Active.Threads}$$

If the number of active threads is small, the occupancy is small. Every single thread may have better performance as the number of registers and shared memory per thread increases. However, we may not have enough threads to hide memory latency. Conventionally, variables are spilled into local memory for better occupancy results and for smaller per-thread register usage, while shared memory is idle or only used a very small portion of time. However, unlike caches, shared memory can be explicitly managed by software just as registers can. The problem we want to address is how to assign variables to registers and shared memory, originally only residing in registers or local memory for spilling purpose, so that we can achieve maximum occupancy and single thread performance is affected minimally. We define the problem as follows:

Assume we assign N_s variables out of the total maximum number of N_{rReq} requested register variables to shared memory. Therefore the maximum number of active threads we can achieve now is:

$$Num.Active.Threads = \lfloor \min(N_{rTotal}/(N_{rReq} - N_s), N_{sTotal}/(N_{sReq} + N_s)) \,/\, \text{BlockSize} \rfloor * \text{BlockSize}$$

We want to maximize $Num.Active.Threads$ with the constraints of: $N_s < N_{sTotal}$ and $Ns < NrReq$.

In the above formula, the invariants are N_{rTotal}, N_{sTotal}, N_{rReq}, N_{sReq}, and the only variant is N_s. We need to find the value of N_s such that the number of active threads is maximum. The first expression in the *min* function is a monotonically increasing function of N_s – the larger N_s is, the larger the expression is. The second expression in the *min* function is a monotonically decreasing function of N_s. Since we want to find the minimum of these two expressions, and these two

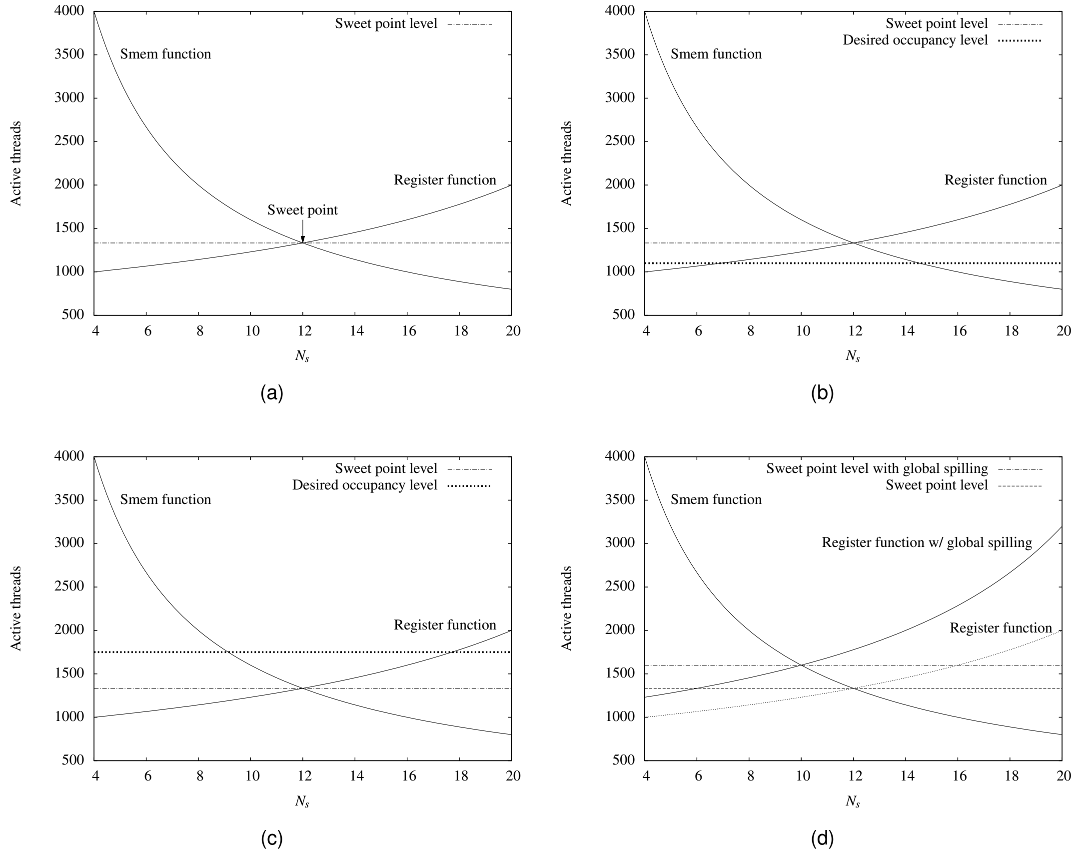


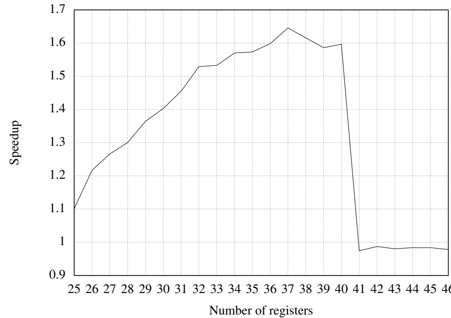
Figure 3: Objective Function for the Number of of Active Threads

expressions respectively decrease or grow with the variant N_s, simply increasing either of these two expressions is unlikely to result in the optimal value. We use Figure 3 as an example to illustrate this case. In Figure 3 (a), we assume $N_{rTotal} = 32k$, $N_{sTotal} = 16k$, $N_{rReq} = 36$ and $BlockSize = 1$. There are two curves corresponding to the number of active threads obtained by exclusively analyzing per-thread register usage or shared memory usage. We would like to take the minimum of these two functions and thus the part we are interested in is the bottom half of the figure below the *sweet point* level indicated in Fig. 3 (a). The two curves intercept at a point which means at a specific N_s value, we can get the same number of active threads from both functions. This point indicates a maximum number of active threads for the bottom half of the figure. We call this a *sweet point*. To solve for this *sweet point*, we need to equate the two functions indicated by per-thread register usage and shared memory usage.

Next, we discuss in different scenarios how we can get the best splitting of live variables into register and shared memory variable number given the above function and the desired occupancy for every specific program.

- *Case 1*: This case is when the number of active threads, indicated by the *sweet point* is greater than the desired number of active threads needed by a program, which is illustrated in Figure 3 (b). In this case, the desired occupancy line intercepts two possible points of the function and therefore we can have two possible combinations of register and shared memory numbers for storing live variables. We prefer the one that uses larger number of registers since registers are faster than shared memory. There are many scenarios in which this case can happen. For instance, when the maximum number of hardware-supported active threads is below this peak *sweet* point, when the total number of threads specified by the program is very small, or when the program is computation intensive and only a number of active threads greater than the number of cores are enough to hide the memory latency.
- *Case 2*: This case occurs when the desired occupancy is larger than the number of active threads indicated by the *sweet point*. We show this case in Figure 3 (c). In this case, we can't achieve the desired occupancy with the given hardware register and shared memory constraints. Therefore, we can only can try to get as much occupancy as possible. The peak point of the bottom half of the figure is indicative of the largest occupancy we can achieve. Therefore, the corresponding number of variables that need to be saved to shared memory before immediate use is given by the *sweet point* value on the x-axis of the figure.

Figure 4: The GPU Program Occupancy Pattern – Example: Particles



- *Case 3*: In the third case, some of these register variables can be spilled into local memory, if local memory spilling latency can be hidden by the overall execution of all active threads. We show this case in Figure 3. The curve marked as "Register function w/ global spilling" describes the function when some registers can be spilled into local memory in addition the ones that are spilled into shared memory, which can further reduce the register pressure. Compared with the original function derived from register usage, we can see that the curve is shifted upward and therefore the new *sweet point* hits a higher peak number of active threads. In this case, we have to be careful of the number of registers we want to spill into local memory since that may add more overhead that can't be offset by the benefits brought by the transformation.

Discussion: Determine the Best Occupancy

How many running threads will be enough to hide the memory latency? The number varies from program to program, and also depends on compilation efficiency (instruction assignment, instruction scheduling, etc.) For a computation-intensive application, as long as every processor core is occupied, the memory latency may be well-hidden at the overlapping execution of multiple processor pipeline stages. However, for a memory-intensive application, we may need many more threads than the total number of processor cores since instruction level parallelism is not enough to make memory overhead invisible. We might need to have millions of threads in total and tens of thousands of threads to be active at every batch of thread execution. Further, the number also depends on the back-end compilation efficiency. If the instructions are chosen and scheduled appropriately, combined with the appropriate GPU thread runtime scheduler, we can get better occupancy even for the same program using better compilation strategy.

How to increase the occupancy of a program from the aspect of programmers or from the aspect of compilers is beyond the scope of this paper. However, we can determine the desired occupancy for every program by profiling runs and the observing the characteristic of occupancy.

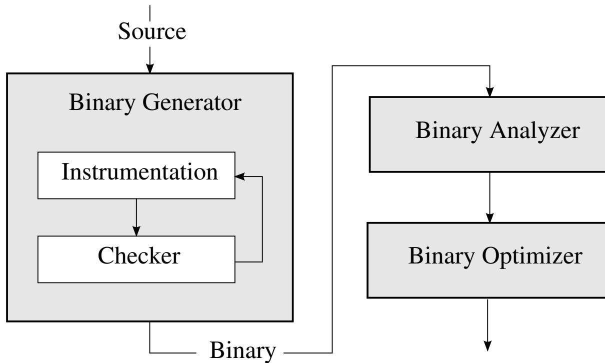
The influence of program occupancy on performance has a very distinctive pattern. Given exactly the same program and enough threads for every batch of active threads execution, if we keep increasing the occupancy by reducing available per-thread registers (assume no shared memory usage), without much register spilling to off-chip memories, at a certain point the performance will stop improving. It will remain the same no matter how much occupancy has been increased. And if we keep decreasing register usage until we have registers spilled to local memory, the performance may start degrading. This behavior resembles cache behavior when the working set size is smaller than the cache size. As long as working set size is smaller than the cache size, no matter how much large a cache we have, the performance will not change. We use Figure 4 to illustrate this pattern for the *particles* simulation program from NVIDIA CUDA SDK. We can see from Figure 4 that performance immediately drops when the number of registers goes above 41. This is because 41 registers is a cut-off point for a sudden change in the number of active threads for this program. When register count is below 41, when we increase register count, the performance gradually increases since the more registers we have per thread, the fewer variables we need to spill into local memory. From register counts of 35 to 40, the performance remains more or less the same. It means the occupancy has reached a saturation level, no matter how we reduce the number of registers to be spilled into local memory, the performance is little affected since occupancy has ability to tolerate memory latency even larger than that. Therefore, to determine how much occupancy a program needs and if the current optimization parameters are desired, we can control the occupancy by adjusting per-thread on-chip memory and getting the occupancy pattern. This helps us make optimization decisions about whether we want to make optimizations or whether we should keep optimizing once the performance can't be improved anymore. In Section 4, we discuss the detailed stress test techniques.

4. Binary Optimization Techniques

In Section 3, we model the relationship between on-chip memory resource allocation in back-end compilation and program performance at different occupancy levels. We are able to determine the best number of live variables to be placed in physical registers and the best number of variables to be placed in shared memory before they are loaded back to registers. In this Section, we describe the implementation techniques for realizing desired on-chip memory resource allocation schemes.

We choose to implement a binary optimizer which directly reads the executable file, analyzes it and transforms it to a new binary file. There are several reasons that we choose binary optimization over other relatively high level code transformations like assembly code transformations. One reason is that code transformations at a higher level such as assembly code may not be completely reflected in the final generated binary, especially when the optimization is about the allocation of

Figure 5: The Binary Optimization Framework



physical hardware memory resources. For instance, pairs of storing/loading of live variables into/from shared memory in PTX (assembly-like code for GPUs) may be optimized out by the assembler since they do not change the memory state of the program. Further, early decisions about how registers should be allocated and spilled, may not be optimal since the assembly code may be further optimized. For instance, different instruction assignment and scheduling schemes may lead to different live variable ranges and therefore the number of maximum co-living variables (also number of registers) may change. Directly performing transformation on binary code has the advantage that on-chip memory resource allocation scheme is final and any changes to the allocation policy can be directly reflected in the transformed executable.

One may wonder why we don't work on the back-end compiler for GPUs. The back-end compilation component for GPU programs are proprietery to different GPU vendors. It is impossible to modify the back-end compilation component directly. The instruction set architecture (ISA) and application binary interface (ABI) are not disclosed. The assembler works like a black box. A binary transformer itself is already very challenginng since it takes lots of efforts to reverse engineer the instruction encoding and decoding process. Not to mention the complexity of translation from relatively higher level intermediate representation code to executable code. Henceforce, we propose a hybrid approach that combines instrumentation of source code and binary transformation without completely hacking every detail of the GPU architecture ISA and ABI.

In Figure 5, we describe the overall workflow. There are three components in this binary optimization framework: binary generator (the hybrid component), binary analyzer and binary optimizer. We describe each of these components and we start with instruction encoding and decoding analysis.

Instruction decoding We first decoded a set of binary instructions we are particularly interested in such as shared/global memory load and store, arithmetic operations for memory address calculating purpose, and register movement for assigning new memory pointers. We followed the this approach when we tried to decode this set of instructions. In this approach, we first write source code or collect a set of sample programs, and use *nvcc* – the CUDA compiler to generate corresponding binary code. Then we use *cuobjdump* – the CUDA disassembler to generate annotated binary instructions. We compare same instructions with different operands or different instructions with same operands. Then we are able to distinguish which part of an instruction is opcode, and which part of the instruction is for register operands or immediate numbers. We discovered that every CUDA instruction is eight byte with Little Endian encoding. We mainly decoded instructions for Kepler GPUs with computing capability 3.0. But the approach follows for other GPU architectures.

Local memory layout and function calls When register variables are spilled to local memory, every thread is assigned a contiguous memory region. The address of the end of every contiguous region is saved into constant memory, which is automatically cached by hardware. The end of memory region instead of the beginning of every local memory partition is used. Because for nested function calls, the pointer to spilling local memory segment for every nested function is aligned on top of local memory segment for the caller function. When a function is called, the spilling region address pointer will be subtracted by a fixed amount. When a function returns, the spilling region address pointer will be incremented by the same fixed amount. If there are many levels of nested function calls, there will be as many local memory segment pointer decrement and increment. Every spilled register variable for the same thread is aligned one after another in local memory. However, in shared memory, in order to avoid bank conflicts, data items accessed at one instruction by adjacent threads are better put in an adjacent manner since accesses to same memory banks at the same time will cause bank conflict and lead to memory access serialization. We use an example to illustrate this idea. When we create shared memory access instructions for the variables to stay in shared memory, we let variables accessed at the same instruction by different threads be placed at adjacent locations. For instance if thread 1 access A at first instruction, B at second instruction, and thread 2 access C and D respectively at these two instructions, and thread 3 E and F, thread 4 G and H, in shared memory, the data items from A to H will be placed like this: "A,C,E,G,B,D,F,H". Using such alignment with we appropriate data item size that is the width of a shared memory bank, we will not have bank conflicts.

4.1. Binary Generator

The first component in this binary optimization framework is the binary generator. The binary generator generates binary code from instrumented source code for further processing. We use this binary generator to achieve several goals. The first one is to get appropriate amount of shared memory allocated. Since we need to use shared memory to help increase occupancy of this program. The second purpose is to generate dummy marker instructions such as volatile instructions so

that in the final generated binary we know where to do start transformation, and if we want to insert add new instructions, we can replace these dummy instructions with new instructions. By doing this, we don't need to change the meta data of the binary file which is also almost impossible since nothing but only code segment within a kernel function can be disassembled with the help of *cuobjdump*. The third purpose is to control the number of real physical registers used. Using *nvcc* and the flag *-maxrregcount*, we can control the number of registers used per-thread. If we reduce it to to a very low number, it will satisfy the requirement and start spilling necessary variables into local memory. After we generate binary from instrumented source, we use a binary checker to see how many spilling memory slots (every slot correspond to a variable) have been created and we may go back to the source instrument stage to modify the code and regenerate the binary if it is not satisfactory.

We use a hybrid approach to generate binary code for further processing purpose mainly because we focus on optimizations of resource allocation, which are completely perpendicular to other optimizations such like instruction scheduling operations. Therefore we can build a binary optimizer completely upon existing infrastructure without worrying about other low level details.

4.2. Binary Analyzer and Optimizer

After we get binary from binary generator, we start analyzing the binary and transforming it. In binary analyzer, we search for two types of instructions - (1) the local memory spilling instructions, part of or all of which will be transformed into shared memory spilling instructions; (2) the *marker* (dummy) instructions we inserted, which we will replace with shared memory address calculation instructions and/or register movement instructions. We also locate the section of the binary code we want to transform. The binary analyzer searches for first matching instruction that is the beginning of the kernel code segment and the last matching instruction that is the end of kernel code segment.

The binary optimizer performs real binary transformation. Since we already constrain the number of physical registers per-thread to be the desired number with *nvcc* compilation flag. Then for the variables that were originally spilled into local memory, we spill them into shared memory. We update all corresponding pairs of store and load instructions. Therefore, we manage to transform all necessary memory instructions without changing the whole structure of the program. Note that *nvcc* use one fixed register for address indexing of local memory, and we change the value of that register to be offset for shared memory locations. Therefore, we don't need to add extra registers for pointers to shared memory live variables.

4.3. Stress Test for Occupancy

In Section 3, we mentioned the occupancy pattern for a GPU program. To get occupancy pattern for a program and to find the parameters such as the occupancy range that we can further improve, we change register usage or shared memory usage. For a program with a given number of threads, we can keep reducing the number of registers per-thread by controlling the compilation flag, and therefore we can increase occupancy. We keep increasing the shared memory allocated for every thread (which we do not really use, just for occupancy control purpose), to reduce the occupancy. We run the same program with different occupancies. Therefore we can profile the original occupancy pattern and make optimization decisions. If a program has reached its occupancy steady level, no matter how we increase occupancy, the performance wouldn't be affected. And there is also no need to use shared memory to help increase its occupancy. If a program hasn't reached its occupancy steady level, then we can optimize programs with this pattern. We show more results in Section 5.

5. Evaluation

5.1. Experiment Setup and Methodology

To evaluate the performance of our method, we chose a representative set of benchmarks with different occupancy pattern from the Rodinia benchmark suite 2.2 [6] and CUDA SDK 5.0. These benchmarks all have considerable amount of usage for registers, which is over the per-thread register limit that leads to maximum number of active threads supported by the hardware scheduler. And we choose the input for all the benchmarks such that we have enough threads to run simultaneously and to occupy the processor cores, which means the number of threads is larger than the maximum number of threads the GPU hardware scheduler can schedule. Only with very large size computation tasks, the impacts of increased occupancy can exhibit in performance. We try to find out the appropriate number of registers per-thread to be staged into shared memory for every given benchmark based on two factors: the desired occupancy level by stress test techniques described in Section 4, and the performance function over number of spilled variables to shared memory described in Section 3.

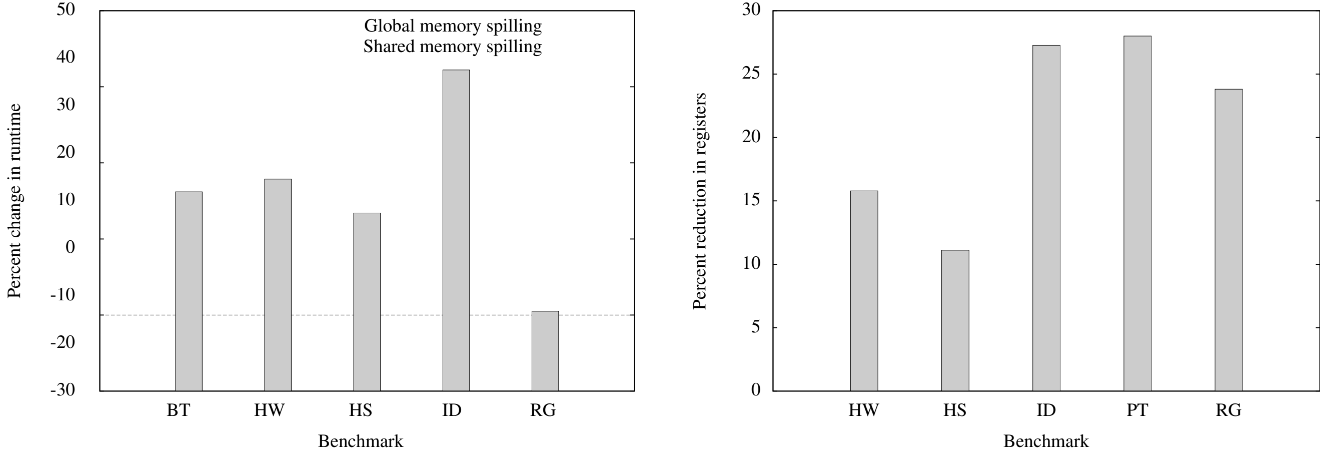
All experiments were run on a host machine with 64-bit Linux version 3.1.10. The CPU processor is an Intel Core i7-3770 running at 3.4 GHz. The GPU is a GeForce GTX 680 which is the latest Kepler generation card from NVIDIA. The compute capability of this Kepler card is 3.0, and it is equipped with 4 GB memory, 1536 CUDA cores, 65536 registers and 49152 bytes shared memory per streaming multiple processor. We used the CUDA compiler *nvcc* and disassembler *cuobjdump* (both at version 5.0). The baseline for performance comparison is the original program compiled with highest optimization flag by *nvcc* and we let *nvcc* choose the number of registers used per-thread according to its optimization strategies. Then we transform the code into binaries using the binary optimizer described in Section 4.

We discuss the features of all these benchmarks. There are five benchmarks in total. The benchmarks *hotspot* and

Table 1: Reduction in Registers and Increase in Occupancy for Every Benchmark

Characteristic	HW	HS	ID	PT	RG
Percent register reduction (%)	15.79	11.11	27.27	28	23.81
Percent increase occupancy	100	100	97.37	35	58.73
Increased shared memory usage (bytes)	8192	8192	24576	0	2048
Block dimension	1024	1024	768	768	128

Figure 6: Benchmark performance improvement (shared memory spill vs. original)



heartwall are from Rodinia. The benchmarks *image denoising*, *particles*, and *recursive Gaussian* are from CUDA SDK. We give a short description of every benchmark as follows:

- Heart Wall (HW) is designed to examine ultrasound images of a mouse heart and track its movement over a sequence of frames. It detects the shape of the inner and outer heart walls and reconstructs an approximate full heart wall shape. It uses operations such as edge detection, SRAD despeckling, and morphological transformation and dilation. Originally it uses 38 registers per-thread and 11872 bytes shared memory per block with the block size being 1024.
- Hotspot (HS) is a transient thermal differential equation solver for estimating processor temperatures on a chip based on the layout and estimated power measurements. It uses 36 registers per-thread and 12288 bytes of shared memory per block. The block size is 1024.
- Image Denoising (ID) uses two adaptive image denoising techniques (KNN and NLM), which are based on computation of both geometric and color distance between texels. We optimize the KNN kernel which uses more registers (44) per-thread and there is no shared memory used in original case. The block size is 768.
- Particles (PT) constructs and simulates a large set of particles and their physical interaction. There are 50 registers used per-thread and the block size is 768. There is no shared memory used in the original program.
- Recursive Gaussian (RG) implements a Gaussian blur using Deriche's recursive method. It uses 42 registers per-thread and the block size is 128.

Experiment Results Summary and Discussion We summarize the experiment results in Table 1 and Figure 6. In Table 1, we show the changes in registers usage, occupancy, and shared memory. All benchmarks have a reduction in register usage from 11.11% to 27.27%. Note that we didn't reduce the register usage for every benchmark to the level of the minimal possible number for best occupancy determined by hardware scheduler. This is due to the constraint on available shared memory size and also the best desired occupancy for every program. For occupancy, we almost doubled the occupancy for *hearwall* and *hot spot*. For other benchmarks, we got the occupancy increase between 35% and 97.37%. For shared memory, we can see from Table 1 that *particles*(PT) has 0 bytes increase in shared memory because we determined from its occupancy pattern in Fig. 4, the occupancy has already reached its saturation point and therefore no further variable spilling into shared memory can help improve performance. Instead, only reducing register count per-thread and having some registers variables spilled into local memory is enough corresponding to the Case 3 in Fig. 3 (d). In Fig. 6, we show the overall speedup for all benchmarks. The benchmark *recursive gaussian* (RG) has almost no performance improvement because the occupancy curve for this benchmark has also almost reached a steady state and increasing occupancy has a little advantage. In general, using the model we designed in Section 3, we can guarantee almost no performance loss since shared memory spilling is always no worse than local memory spilling. Further, we can improve performance for the benchmarks that have more potential by up to 1.65 times.

6. Related Work

Many studies in the past few years have been proposed on register spilling between physical registers and off-chip global memory. Some of them are based on graph coloring models [4]. [2] proposed integer linear program modeling of register allocation for CISC machines. In [14] and [9], the authors particularly tackled the register spilling problem for software pipelining in loops, which mainly focuses on minimize the stretchable liveness range of variables used for pipelining instructions at different iterations of loops. All these previous studies are for sequential programs, which didn't take parallel architecture features into account. In [1], the authors studied the register allocation schemes for vector machines. The vector registers, which is a set of registers for operands of one vector instruction, need to be ready before this vector instruction can be executed. However, this scenario is different from Simultaneous Multi-Threading (SMT) scenarios on modern GPUs.

GPU architecture is a combination of vector processing and simultaneous multi-threading. [13] points out that the ability for memory latency hiding among different vector thread groups has been critical. The authors present a model for gpu programs that predicts the performance by calculating *memory warp parallelism (MWP)* and *computation warp parallelism (CWP)*. But this work is not related to register usage optimization on GPUs. More recent literature on GPU architecture that is relevant to this work has been on topics such as GPU exception handling [11], where register states need to be stored and reloaded for resuming execution after an exception is detected, and energy saving [7], where the location of registers is important because the distance between the registers and processors determines the amount of energy consumed during data movement process, and register space saving [15], which combines SRAM and DRAM to store more bits into the die area and therefore increase the size of register file. Most of these studies propose architecture extensions and are evaluated based on hardware simulators.

Overall, these previous proposed work mainly targets register spilling for sequential programs and concentrates only spilling into off-chip memory. Very few studies are for GPU register allocation and spilling and the literature that are relevant tackles other problems in GPU computing rather than investigating the register pressure problem itself.

7. Conclusion

In this paper, we proposed a novel technique to combine register memory and scratch-pad memory for thread states storing purposes in order to mitigate the many-core processor utilization problem brought by register pressure. The current GPU compilation techniques failed to take the factor of on-chip memory resource contention into consideration and commodity GPU compilers use almost the same approaches for register pressure problem in CPUs. We greatly improved the performance of a set of real GPU application on real GPUs by simple binary translations using the models and techniques we proposed in this paper.

References

[1] R. Allen and K. Kennedy, "Vector register allocation," *Computers, IEEE Transactions on*, vol. 41, no. 10, pp. 1290 –1317, oct 1992.

[2] A. W. Appel and L. George, "Optimal spilling for cisc machines with few registers," in *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, ser. PLDI '01. New York, NY, USA: ACM, 2001, pp. 243–253. [Online]. Available: http://doi.acm.org/10.1145/378795.378854

[3] P. Briggs, K. D. Cooper, and L. Torczon, "Improvements to graph coloring register allocation," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 428–455, May 1994. [Online]. Available: http://doi.acm.org/10.1145/177492.177575

[4] G. J. Chaitin, "Register allocation & spilling via graph coloring," in *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, ser. SIGPLAN '82. New York, NY, USA: ACM, 1982, pp. 98–105. [Online]. Available: http://doi.acm.org/10.1145/800230.806984

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

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 44–54. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2009.5306797

[7] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "A hierarchical thread scheduler and register file for energy-efficient throughput processors," *ACM Trans. Comput. Syst.*, vol. 30, no. 2, pp. 8:1–8:38, Apr. 2012. [Online]. Available: http://doi.acm.org/10.1145/2166879.2166882

[8] S. Hack, D. Grund, and G. Goos, "Register allocation for programs in ssa-form," in *In Compiler Construction 2006, volume 3923 of LNCS*. Springer Verlag, 2006.

[9] J. Llosa, M. Valero, E. Ayguadé, and A. González, "Hypernode reduction modulo scheduling," in *Proceedings of the 28th annual international symposium on Microarchitecture*, ser. MICRO 28. Los Alamitos, CA, USA: IEEE Computer Society Press, 1995, pp. 350–360. [Online]. Available: http://dl.acm.org/citation.cfm?id=225160.225211

[10] J. Luitjens, "Cuda global memory usage & strategy." NVIDIA, 2011. [Online]. Available: http://developer.download.nvidia.com/CUDA/training/cuda_webinars_GlobalMemory.pdf

[11] J. Menon, M. De Kruijf, and K. Sankaralingam, "igpu: exception support and speculative execution on gpus," in *Proceedings of the 39th International Symposium on Computer Architecture*, ser. ISCA '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 72–83. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337159.2337168

[12] M. Poletto and V. Sarkar, "Linear scan register allocation," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 5, pp. 895–913, Sep. 1999. [Online]. Available: http://doi.acm.org/10.1145/330249.330250

[13] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "A performance analysis framework for identifying potential benefits in gpgpu applications," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 11–22. [Online]. Available: http://doi.acm.org/10.1145/2145816.2145819

[14] J. Wang, A. Krall, M. A. Ertl, and C. Eisenbeis, "Software pipelining with register allocation and spilling," in *Proceedings of the 27th annual international symposium on Microarchitecture*, ser. MICRO 27. New York, NY, USA: ACM, 1994, pp. 95–99. [Online]. Available: http://doi.acm.org/10.1145/192724.192734

[15] W.-k. S. Yu, R. Huang, S. Q. Xu, S.-E. Wang, E. Kan, and G. E. Suh, "Sram-dram hybrid memory with applications to efficient register files in fine-grained multi-threading," in *Proceedings of the 38th annual international symposium on Computer architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 247–258. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000094