Compiler-Assisted Resource Management for CUDA Programs

Yi-Ping You and Yu-Shiuan Tsai

Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan
ypyou@cs.nctu.edu.tw,ystsai@sslab.cs.nctu.edu.tw

Abstract. CUDA is a C-extended programming model that allows programmers to write code for both central processing units and graphics processing units (GPUs). In general, GPUs require high thread-level parallelism (TLP) to reach their maximal performance, but the TLP of a CUDA program is deeply affected by the resource allocation of GPUs, including allocation of shared memory and registers since these allocation results directly determine the number of active threads on GPUs. There were some research work focusing on the management of memory allocation for performance enhancement, but none proposed an effective approach to speed up programs in which TLP is limited by insufficient registers. In this paper, we propose a TLP-aware register-pressure reduction framework to reduce the register requirement of a CUDA kernel to a desired degree so as to allow more threads active and thereby to hide the long-latency global memory accesses among these threads. The framework includes two schemes: register rematerialization and register spilling to shared memory. The experimental results demonstrate that the framework is effective in performance improvement of CUDA kernels with a geometric average of 14.8%, while the geometric average performance improvement for CUDA programs is 5.5%.

Keywords: CUDA, compilers, register-pressure reduction, thread-level parallelism

1 Introduction

General purpose application development for graphics processing units (GPUs) has recently gained considerable attention in various domains for accelerating data-intensive applications. NVIDIA's CUDA (Compute Unified Device Architecture) [9] is one of the dominating C-based programming models, which enables massively parallel high-performance computing on NVIDIA's powerful GPUs.

In the CUDA execution model, all input data are initially transferred from host memory to the global memory of GPUs so that all streaming multiprocessors (SMs), which are analogous to CPU cores and are each designed to execute hundreds of simultaneous threads, can access them. However, a global memory access takes a very long latency (hundreds of cycles) in comparison with a common instruction [9]. The execution of a CUDA program requires massively-parallel computation to hide global memory latencies. In this regard, CUDA programmers are encouraged to use the shared memory, a fast, small on-chip memory within each SM, in order to enable global memory coalescing, but to keep the shared memory requirement low enough to allow more threads to be active at a time within an SM, giving additional scope for hiding global memory latencies.

Table 1. The resource usage of a simple kernel.

Resource	Usage
# of threads per block	256
# of registers per thread	17
shared memory usage per block	4KB

Table 2. Occupancy calculation of the kernel whose resource usage is as shown in Table 1.

Resource limits per SM (Compute Capability 1.3)		# of active warps per SM
Registers	32 allocation units (512 per unit)	$\lfloor 32/\lceil 256 \times 17/512 \rceil \rfloor \times (256/32) = 24$
Shared memory	32 allocation units (0.5KB per unit)	$\lfloor 32/\lceil 4/0.5 \rceil \rfloor \times (256/32) = 32$
Maximum # of active blocks	8	$8 \times (256/32) = 64$
Occupancy	$min(24, 32, 64)/32 \times 100\% = 75\%$	

More specifically, the performance of a CUDA kernel is deeply influenced by the resource requirements of GPUs within the kernel, especially registers and shared memory because the register and shared memory usages directly impact the number of active thread blocks which determines the occupancy of a program. The higher occupancy, the more threads are likely to be queued or executed on an SM and thus the more thread-level parallelism (TLP) will be achieved for hiding long-latency global memory operations. In general, occupancy is the ratio of the number of active warps (32 parallel threads) to the maximum number of warps—a warp is the scheduling unit on each SM—supported on an SM of the GPU in terms of limited hardware resources. For example, given the resource usage of a CUDA kernel in Table 1, suppose the kernel is executed on a NVIDA GPU whose Compute Capability is 1.3, implying that each SM has 32 shared memory units and 32 register units and possesses a maximum of 32 active warps and 8 active thread blocks where register allocation unit size is 512 and shared memory allocation unit size is 0.5KB. The number of active warps per SM is determined by the minimum of three factors: register, shared memory, and thread availabilities, giving that the occupancy of the kernel is 75%, as shown in Table 2, where register availability is the limiting factor in this case. However, if we reduce the register usage of the kernel thread by one register (i.e., 16 registers per thread), the number of active warps will increased from 24 (three active blocks) to 32 (four active blocks), making the occupancy 100% and thus increasing the TLP for hiding global memory latencies.

We examined the kernels in sample code of CUDA SDK [7] and found that 54.3% of kernels (44 of 81 kernels) may benefit from reducing their register usage with regard to increasing one extra active thread block when considering their occupancies and shared memory usage. Figure 1 shows that 56.8% of kernels (25 of 44 kernels) with the occupancy of 62.6% on average can have an extra active thread block at run time if we could reduce their register usage by economizing the use of one or two registers. In other words, they have high potential to be optimized for increasing their TLP without sacrificing too much instruction-level parallelism (ILP).

Therefore, in this paper we attempt to reduce the register usage of a CUDA kernel to a specific degree which is lower enough so as to allow more threads to be active for hiding global memory latencies and thus speeding up the CUDA execution. We propose two methodologies to decrease the register usage of a CUDA kernel: the first

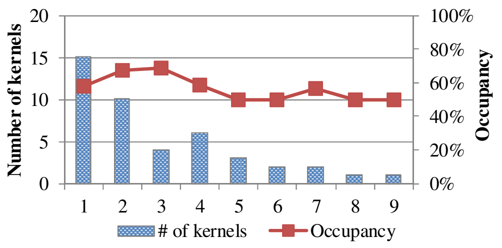


Fig. 1. The number of registers to reduce in order to increase an active thread block.

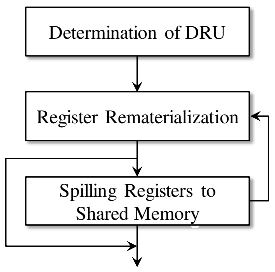

Determination of DRU
Register Rematerialization
Spilling Registers to Shared Memory

Fig. 2. TLP-aware register-pressure reduction framework.

one is register rematerialization which reduces the register pressure by recomputing the variable to cut the live range, and the second one is register spilling which spills registers to shared memory programmers did not allocate.

2 TLP-Aware Register-Pressure Reduction Framework

We propose a TLP-aware register-pressure reduction framework to reduce register usage in order to increase the number of active thread blocks for more TLP. More specifically, we attempt to use techniques of rematerialization and spilling to reduce the register usage of a CUDA kernel to a desired degree so that the register usage of a thread decreases and thus, given a limited number of registers on an SM, more threads turn into active. Such increment of active threads might result in more latencies of global memory accesses being hidden and consequently exploit more TLP. Figure 2 shows the flow chart of the proposed framework, whose input is a low-level intermediate representation (IR) and whose output is the low-level IR with reduced register usage. The framework first sets a goal of reducing the register usage to a desired degree (DRU) according to a simple cost model and then achieves it by performing the rematerialization and spilling schemes. The processes of rematerialization and spilling are repeated and interleaved until the goal is achieved or no changes could be made. Generally, the order of the two processes is free since each one would have chance to reduce more register usage after the other is performed. However, in some cases applying only one scheme is sufficient for the purpose, so the rermaterialization scheme is considered to be carried out first due to its less overhead in terms of the number of instructions being added in comparison with the spilling scheme.

2.1 Determination of Desired Degree of Register Usage

In CUDA architecture, threads are organized into thread blocks, which are sets of concurrent threads that can cooperate among themselves. The decrease in register usage per thread may not lead to the increase in the number of active thread blocks. The goal of the framework is thus to increase the number of active thread blocks by reducing the register usage of each thread to a specific degree, namely the desired register usage (DRU), while considering the relation between the size of a thread block and the number of registers available on an SM:

$$DRU = \left\lfloor \frac{N_r^{sm}}{N_t^{tb} \times N_{db}^{sm}} \right\rfloor , \tag{1}$$

where N_r^{sm} is the number of registers per SM, N_t^{tb} is the number of threads per thread block, and N_{db}^{sm} is the desired number of active thread blocks. The DRU is actually the number of registers per thread when all registers available on an SM is averagely allocated to all of the active threads.

However, the registers for each thread block are allocated in multiples of register-allocation unit, so Equation 1 must be rewritten accordingly. The number of register-allocation units required by a thread block is $\lceil N_t^{tb} \times DRU/N_r^{ru} \rceil$, where N_r^{ru} is the number of registers per register-allocation unit, and the number of active thread blocks is therefore $\lfloor N_{ru}^{sm}/\lceil N_t^{tb} \times DRU/N_r^{ru} \rceil \rfloor$, where N_{ru}^{sm} is number of register-allocation units per SM. Accordingly, we can derive the following equation to formulate the DRU:

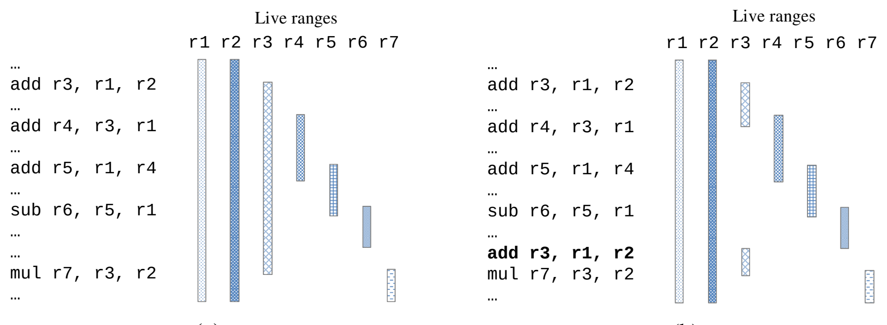
$$DRU = \left\lfloor \frac{\left\lfloor \frac{N_{ru}^{sm}}{N_{db}^{sm}} \right\rfloor \times N_r^{ru}}{N_t^{tb}} \right\rfloor . \tag{2}$$

For example, given a CUDA program which contains a kernel function whose register usage is 17 and the size of a thread block is configured as 256 (as shown in Table 1), it is known that the number of active thread blocks per SM of the kernel is three if the register file size causes a bottleneck of resource limitation. Suppose each SM has 32 register-allocation units (i.e., N_{ru}^{sm} is 32) and a register-allocation unit is 512 (i.e., N_r^{ru} is 512). If we reduce the register usage to 16 ($\lfloor(\lfloor 32/4 \rfloor \times 512)/256 \rfloor$), the number of active thread blocks per SM will become four, which induces more TLP and thereby increases the performance of the program.

2.2 Register Rematerialization

Rematerialization is a compiler optimization which generally has been used to save time by recomputing a value, rather than loading it from memory [2]. I n this paper, we propose rematerialization to reduce register usage of a thread by recomputing the value of a variable to split the live range of the variable and thus to increase the number of active thread blocks to exploit more TLP. For example, given a program segment with live range information in Figure 3(a), it is observed that the live range of variable `r3` overlaps with the period that sustains the register pressure. With rematerialization, we could split the live range of variable `r3` to decrease the register usage by adding a new instruction that recomputes the value of variable `r3` with two variables, `r1` and `r2`, right before the value is accessed as shown in Figure 3(b). The concept of rematerialization in the above example seems easy to implement, but many factors actually need to be considered. If we insert the recomputing instruction at any location in the code, the rematerialization may lengthen the live ranges of the source variables and increase the register usage instead. Therefore, we need to consider such effect and determine appropriate locations for placing recomputing instructions. In general, a recomputation instruction should not appear after the last overlap region of the live ranges of its source variables in order to avoid extending the live ranges of the source variables.

Algorithm 1 presents the details for using rematerialization to reduce the register usage of a program to a given degree in a compilation phase. Conceptually, it determines which values are candidates to be rematerialized with several condition checks which are discussed later. At the beginning, Algorithm 1 examines whether the register usage of the current IR reaches the desired degree of register usage (DRU for short). The following procedures will be iterated until the register usage is equal to or less than



```
…
add r3, r1, r2
…
add r4, r3, r1
…
add r5, r1, r4
…
sub r6, r5, r1
…
…
mul r7, r3, r2
…
```

```
…
add r3, r1, r2
…
add r4, r3, r1
…
add r5, r1, r4
…
sub r6, r5, r1
…
add r3, r1, r2
mul r7, r3, r2
…
```

Fig. 3. An example of rematerialization: (a) a code segment which requires five physical registers, and (b) the code segment with rematerilzation which requires only four physical registers.

Algorithm 1: Register Rematerialization

```
Input: a low-level IR and the desired degree of the register usage (DRU).
Output: the low-level IR with reduced register usage.
begin
    while register usage $>$ DRU do
        Locate a region RP that has the highest register pressure. Suppose the region begins at one instruction
        whose ID is RPbegin and ends at another instruction whose ID is RPend, denoted as [RPbegin, RPend].
        for each variable $\nu$ whose live range is [Def$\nu$, Use$\nu$], where Def$\nu$ and Use$\nu$ are instruction IDs do
            if Def$\nu$ is not a long instruction (Condition 1) then
                if [Def$\nu$, Use$\nu$] $\supseteq$ [RPbegin, RPend] and $\nu$ is not accessed within [RPbegin, RPend] (Condition
                2) then
                    Locate the last overlap location, LastOverlapLoc, of the live ranges of the source variables of
                    instruction Def$\nu$.
                    if LastOverlapLoc $>$ RPend (Condition 3) then
                        if $\nu$ has an explicit definition (Condition 4) then
                            Add $\nu$ to CandidateSet.
        Locate other regions that has the same register usage as RP does where applicable.
        Select a variable $\alpha$ whose live range covers the most register pressure regions or whose live range is
        longest from CandidateSet.
        Insert the recomputing instruction of $\alpha$ prior to one of the following locations which appears first: $\alpha$'s
        $LastOverlapLoc$ or the first instruction which accesses $\alpha$ after the RP region.
        Break if there is no change could be made.
```

the DRU or no further changes could be made. A region which sustains the highest register pressure (RP for short) is located firstly. Suppose the RP begins at an instruction whose ID is *RPbegin* and ends at another instruction whose ID is *RPend*, denoted as [*RPbegin*, *RPend*]. Then four condition checks are used for examining each variable ν within the given program to determine which are candidates for rematerialization. More specifically, if ν satisfies all the conditions, it is eligible for rematerialization. Suppose the live range of variable ν is [*Def*ν, *Use*ν], where *Def*ν and *Use*ν are instruction IDs. We introduce the conditions as follows.

1. The first condition is that ν is out of consideration for rematerialization when the *Def*ν is a long instruction, such as a global memory operation or a division instruction, which takes hundreds of cycles for execution, so it is unworthy to rematerialize the value of ν.
2. The second condition is that the live range of ν must cover the RP region and ν is not accessed within the region so that the register usage within the region can be reduced by one by splitting the live range of ν; otherwise, the register pressure remains the same, only that its period is shorten. For example, given a code segment

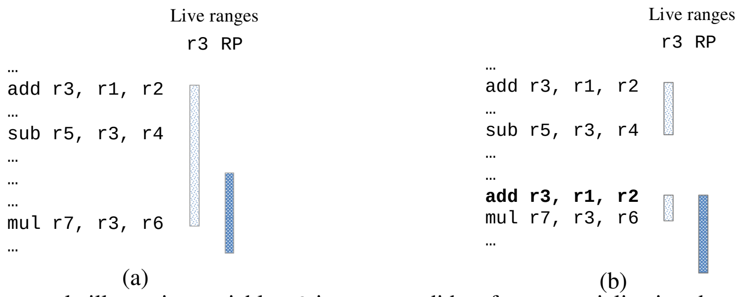


Fig. 4. An example illustrating variable `r3` is not a candidate for rematerialization due to its live range not fully covering the period of register pressure (`RP`): (a) a program with the live rang and register pressure information, and (b) the register pressure remains even if the live range of `r3` is split by remeterializing `r3`'s value.

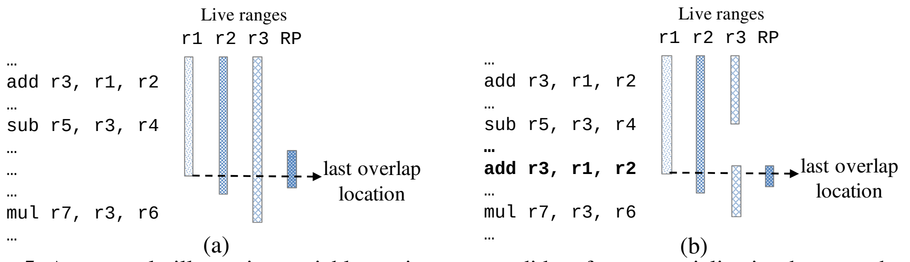


Fig. 5. An example illustrating variable `r3` is not a candidate for rematerialization because the last overlap location is located within period of register pressure (`RP`): (a) a program with the live range and register pressure information, and (b) the register pressure remains even if the live range of `r3` is split by remeterializing `r3`'s value.

in Figure 4(a) in which the live range of variable `r3` partially overlaps a period which sustains the register pressure (`RP`), we will not rematerialize the value of variable `r3` since the register pressure will not be reduced after the rematerializatin as shown in Figure 4(b).

3. The third condition is that the last overlap location, *LastOverlapLoc*, of the live ranges of the source variables of instruction *Def*ν must appear after the RP region so that the recomputing instructions can be inserted behind the RP region to reduce one register usage by splitting the live range of ν; otherwise, the recomputing instructions will be inserted within or before the RP region to avoid lengthening the live ranges of the source variables of the recomputing instruction, and the degree of the register pressure will still remain the same. For example, given a code segment in Figure 5(a), it shows that the last overlap location of the live ranges of the source variables of `r3`'s definition instruction is located within a period which sustains register pressure (`RP`), and therefore variable `r3` is not a candidate for rematerialization because the register pressure will not be reduced after rematerialization as shown in Figure 5(b).
4. The rematerialization reduces register usage by recomputing the value of ν, so the sources of the recomputing instruction of ν must be explicit. For example, given the information of live range of variable `r5` and a period which sustains register pressure (`RP`) in Figure 6, we will not rematerialize the value of `r5` at block `B3` because the definition of `r5` comes from two branches (one from `B1` and the other

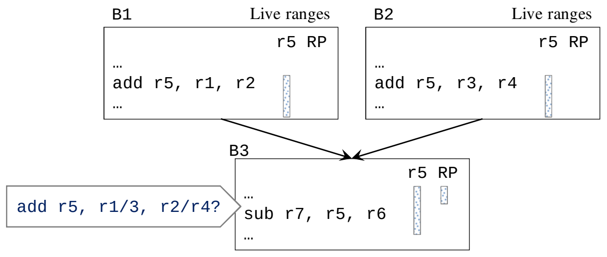


Fig. 6. An example illustrating variable `r5` is not a candidate for rematerialization because the definition of `r5` is not explicit.

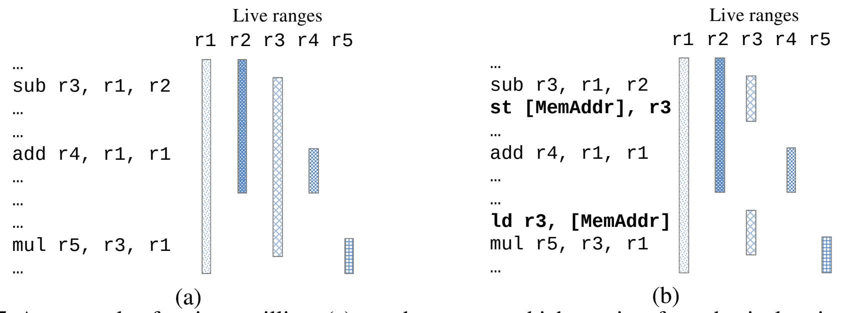
from `B2`). Rematerializing `r5` is not possible unless extra conditional checks are inserted before the rematerialization; however, such rematerialization associated with conditional branches is costly.

Variable ν is qualified as a candidate for rematerialization only if it satisfies the four conditions. However, there may be multiple candidates for rematerialization, so we need a fast and effective mechanism to pick a variable out for rematerialization. We select a variable α whose live range covers the most register pressure regions which has the 'same register usage as *RP* does or whose live range is longest from all candidates since rematerializing such variable has a high possibility to effectively reduce the most register usage without further rematerializations. Then the recomputing instruction of α is inserted prior to one of the following locations which appears first: α's $LastOverlapLoc$ or the first instruction which accesses α after the *RP* region.

2.3 Spilling Register to Shared Memory

In general, compiler often applies spilling techniques to reduce register usage. A spilling process involves demoting a live (but temporarily not used) variable from a register to memory so as to make one register available to other variables and promoting the variable from memory to a register before it is accessed. Spilling makes it possible to reduce the register usage of a program, although spilling doesn't come for free. It comes at the expense of performance due to the extra store/load instructions inserted [1], and the performance might be dramatically decreased if the store/load instructions take a large number of cycles. However, register spilling provides us an opportunity to reduce the register usage of the kernel function of CUDA programs for exploring more TLP. Figure 7 illustrates a simple example of register spilling to shared memory to reduce the register usage. Given a code segment and information of live ranges in Figure 7(a), it is observed that the live range of variable `r3` overlaps a period which sustains register pressure. With register spilling, we could split the live range of `r3` to decrease the register usage by adding a pair of store and load instructions as shown in Figure 7(b).

In CUDA programming, programmers are able to limit the register count of a kernel by using compiler option *-maxrregcount* for increasing the number of active thread blocks on an SM. In fact, the CUDA compiler does use spilling techniques to meet such demands. If the number of the registers required by a kernel function exceeds the specified register count, the compiler spills values from registers to local memory, which is private for each thread. Unfortunately, local memory is essentially allocated within the global memory which requires hundreds of cycles to access. As a matter



```
…
sub r3, r1, r2
…
…
add r4, r1, r1
…
…
…
mul r5, r3, r1
…
```

```
…
sub r3, r1, r2
st [MemAddr], r3
…
add r4, r1, r1
…
…
ld r3, [MemAddr]
mul r5, r3, r1
…
```

Fig. 7. An example of register spilling: (a) a code segment which requires four physical registers, and (b) the code segment with spilling which requires only three physical registers.

of fact, spilling registers to local memory significantly slows down the computation threads in GPUs and counteracts the benefit of the increased TLP. Consequently, we think of using shared memory, the fastest memory (up to 16KB per SM in Compute Capability 1.3) shared among threads in a thread block in CUDA GPUs, as the space for spilling registers as long as programmers do not fully utilize it. We examined the sample programs (34 programs in total) in CUDA SDK [7] and observed that the shared memory requirements of 85.29% the programs are less than 4 KB, and the phenomenon remains for other CUDA benchmark suites like Parboil [5] and benchmark collection from NVIDA's CUDA Developer Zone [6]. In this regard, we propose an approach of spilling registers to shared memory to reduce register usage of a thread with the aim of increasing more TLP. The proposed spilling approach has no negative effect in terms of memory access because the shared memory is as fast as registers. However, aside from the extra store/load instructions to shared memory, the approach requires an indexing mechanism for threads to privately use their own space within the shared space.

Algorithm 2 shows the process for using register spilling to reduce the register usage of a program to a desired degree in a compilation phase and it works almost the same as Algorithm 1, but with different condition checks. In brief, it determines which values are candidates to be spilled with a condition check. Algorithm 2 repeats the following procedures until the desired degree of register usage (DRU) is reached or there are no more spaces for register spilling, which is determined by examining whether the number of concurrently spilled registers exceeds the maximum spilling capacity (MSC) of the given IR. MSC indicates the maximum amount of concurrent spilling allowed which is essentially determined upon the shared memory usage and the thread block configuration of the given kernel, and it is regarded as an input of Algorithm 2. A region which sustains the highest register pressure (RP for short) is firstly located at $[$*RPbegin*, *RPend*$]$ which represents that region of RP begins at an instruction whose ID is *RPbegin* and ends at another instruction whose ID is *RPend*. Then each variable ν is examined by a condition check: the live range of ν must cover the RP region and ν is not accessed within the region. The condition check ensures that the register usage within the RP region is decreased by one after variable ν is spilled during the RP region.

If variable ν satisfies the condition check, variable ν is eligible for spilling. However, there may be many candidates for spilling and spilling one of them would be adequate to lighten the register pressure within the RP region, so we need to pick a variable out for spilling. A variable α whose live range covers the most register pres-

Algorithm 2: Register Spilling

```
Input: a low-level IR, the maximum spilling capacity (MSC) of the given IR and a desired degree of the register usage (DRU).
Output: the low-level IR with reduced register usage.
while register usage $>$ DRU and number of concurrently spilled registers $<$ MSC do
    Locate a region RP that has the highest register pressure. Suppose the region begins at one instruction whose ID is RPbegin and ends at another instruction whose ID is RPend, denoted as [RPbegin, RPend].
    for each variable $\nu$ whose live range is [Def$\nu$, Use$\nu$], where Def$\nu$ and Use$\nu$ are instruction IDs do
        if [Def$\nu$, Use$\nu$] $\supseteq$ [RPbegin, RPend] and $\nu$ is not accessed within [RPbegin, RPend] then
            Add the $\nu$ to CandidateSet.
    Locate other regions that has the same register usage as RP does where applicable.
    Select a variable $\alpha$ whose live range covers the most register pressure regions or whose live range is longest from CandidateSet.
    Insert the a store (st [$SharedMemAddr$], $\alpha$)/load (ld $\alpha$, [$SharedMemAddr$]) instruction of $\alpha$ after/before the first instruction which accesses $\alpha$ before/after the RP region.
if any spilling operation is performed then
    Collect the number of concurrently spilled registers, $N_{sr}$, of the IR.
    Allocate a space of size $N_t^{tb} \times N_{sr} \times 4$ at $MemAddr_{\text{base}}$ from the shared memory.
    Insert the address computation instructions of register spilling for each thread according the following method: $SharedMemAddr = MemAddr_{\text{base}} + (ID_t \times N_{sr} + Offset(\alpha)) \times 4$.
```

sure regions which has the same register usage as *RP* does or whose live range is longest is picked from all candidates because spilling such variable has high potential to reduce the most register usage without too many spilling operations. Then a store (`st [`$SharedMemAddr$`],` α)/load (`ld` α`, [`$SharedMemAddr$`]`) instruction is inserted after/before the first instruction which accesses α before/after the *RP* region, where `[`$SharedMemAddr$`]` is a memory address within the shared memory and will be set after all spilling instructions are inserted. The address computation instructions are inserted according to Equation 3, which uses thread ID for indexing the space allocated for the running thread, to avoid the spilled registers of each thread being stored in the same location of the shared memory because CUDA kernels are executed in an SIMT manner. In brief, each thread owns $N_{\text{sr}} \times 4$ bytes of memory space, where N_{sr} is the number of concurrently spilled registers and each register occupies four bytes. The memory address $SharedMemAddr$ that is used to store the spilled variable α is, therefore, defined as

$$SharedMemAddr = MemAddr_{\text{base}} + (ID_t \times N_{sr} + Offset(\alpha)) \times 4 \qquad (3)$$

where $MemAddr_{\text{base}}$ is a memory address which points to the base address of the space allocated for spilling registers in the shared memory, ID_{t} indicates the unique ID of the running thread, and $Offset(\alpha)$ returns the index of α in the thread's own space after a memory allocation, which is similar to a naive register allocation.

Continuing with the example of spilling variable `r3` to memory in order to reduce the register pressure of the given kernel in Figure 7, suppose that the number of registers required by the kernel is reduced by two by the first phase (while loop) of Algorithm 2. The computation of memory addresses for spilling are then inserted according to Equation 3. Figure 8 illustrates the shared-memory allocation of the kernel, in which the 1024-byte region between 0x0000 and 0x0399 are used by programmers, while its following space is allocated for spilling registers for all threads in active thread blocks. The spilling address for the thread whose ID is “1” is 0x0408 (0x0400 ($MemAddr_{\text{base}}$) + 1 (ID_t) $\times$ 2 (N_{sr}) $\times$ 4), and the second concurrently spilled register of the thread is located at 0x040C.

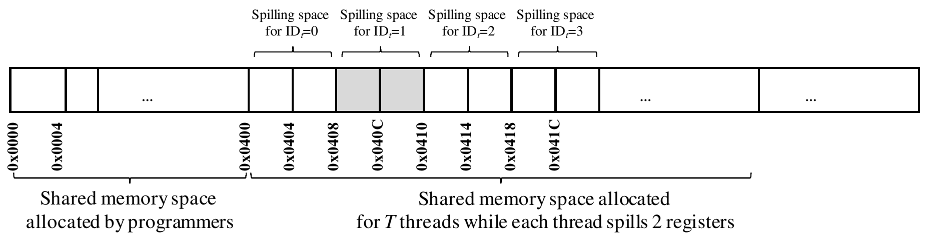

Spilling space for ID_i=0
Spilling space for ID_i=1
Spilling space for ID_i=2
Spilling space for ID_i=3
...
...
...
0x0000
0x0004
0x0400
0x0404
0x0408
0x040C
0x0410
0x0414
0x0418
0x041C
Shared memory space allocated by programmers
Shared memory space allocated for *T* threads while each thread spills 2 registers

Fig. 8. An example of shared memory allocation for threads.

2.4 Implementation Issues

In this paper, we propose two methodologies (rematerialization and spilling) that manipulate the live ranges of variables of a given kernel thread to reduce its register usage. Therefore, realization of our approach requires somewhat of a connection to the process of register allocation so that the register usage of the thread is reduced to a certain degree demanded by the proposed framework. In general, register allocation is a necessary component of most compilers, especially those for RISC machines. However, in the CUDA compilation system, the “real” register allocation is performed in the *ptxas* assembler, while the register allocator in the *nvopencc* compiler merely allocates virtual registers to each variable. Unfortunately, the implementation of the *ptxas* assembler is not available to the public domain. The only component which is open-sourced in the CUDA SDK is the BSD-licensed *nvopencc* compiler, which generates PTX assembly code. Therefore, to automate the proposed framework for each CUDA program, we incorporated the TLP-aware register-pressure reduction framework into the *nvopencc* compiler, although it is more appropriate if we could implement the framework in the *ptxas* assembler. As a matter of fact, implementing the proposed framework as a phase of the *nvopencc* compiler has two potential problems:

1. In addition to assembling PTX code into binary code, the *ptxas* assembler performs redundancy elimination optimizations within basic blocks for performance improvement. However, rematerialization can be considered as an opposite optimization to redundancy elimination; that is, a prior rematerialization will be discarded by the redundancy elimination if the recomputing instruction is located in the same basic block that the original computing instruction resides in since the recomputing instruction is treated as a redundancy. Moreover, we believe that the *ptxas* assembler also performs some other optimizations, such as instruction scheduling, that might destroy the efforts of rematerialization and spilling optimizations to reduce register pressure. Therefore, the effectiveness of the proposed framework might be decreased by the *ptxas* assembler because *ptxas* is not aware of these optimizations.
2. As described above, the *ptxas* assembler involves optimizations that might change the live ranges of variables before it performs the register allocation, so the actual register usage of a thread might not be the same as what we expect in the *nvopencc* compiler, where the proposed framework was implemented. We approach this problem in an adaptive manner by means of iterative compilation, in which a controller over the *nvopencc* compiler and the *ptxas* assembler is required. The controller initially determines a DRU with the method proposed in Section 2.1, invokes the

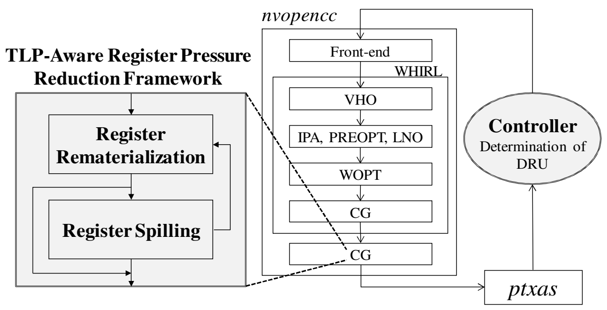


Fig. 9. The implementation details of the TLP-aware register-pressure reduction framework and the controller for iterative compilation.

compiler and assembler, and examines whether the register usage of the given kernel is less than or equal to the DRU. If not, the controller decreases the DRU and recompiles and reassembles the kernel until the goal of reducing the register usage to the DRU set in the first stage is achieved. Figure 9 shows the implementation details of our framework in the *nvopencc* compilation framework—the framework is actually implemented in the phase of code generator (CG)—and the proposed controller that allows iterative compilation.

3 Experimental Results

3.1 Evaluation Platform and Benchmarks

We used an AMD Athlon™ II quad-core processor, running at 2.8GHz with 4GB of main memory with two-level caches, and an NVIDIA Geforce GTX 285 with CUDA version 3.2 [8] as the target platform for our experiments. The Compute Capability of GTX 285 is 1.3, allowing at most 1024 active threads on an SM. The GTX 285 is comprised of 30 SMs and each SM has 8 streaming processors (SPs) for a total of 240 SPs, in which the SPs are clocked at 1.674GHz. Each group of 8 SPs shares one 16 KB of fast per-block shared memory and 16384 registers.

Our proposed TLP-aware register-pressure reduction framework were evaluated by 36 benchmarks (comprising 81 kernels) collected from the sample code in NVIDIA's CUDA SDK [7] and NVIDA's CUDA Developer Zone [6]. These kernels can be divided into two categories according to their characteristics of whether they may benefit from reducing their register usage and 44 of them may benefit from reducing their register usage so as to increase one extra active thread block.

3.2 Performance Evaluation

Performance Improvement for CUDA Kernels In order to evaluate the effectiveness of the two schemes (register rematerialization and register spilling) of the proposed framework, these two schemes were firstly evaluated separately and then combined together to estimate the overall effect of the framework. Figure 10 shows the performance improvement and the number of instructions inserted for each kernel using register rematerialization/spilling for nine selected kernels. It shows that the execution time of the kernels is reduced by geometric averages of 14.4% and 12.7% when rematerialization and spilling is applied, respectively. In some cases, such as *RandomGPU*, *matrixMul*, *bitonicSortShared*, and *bitonicSortShared1*, the scheme of register spilling has

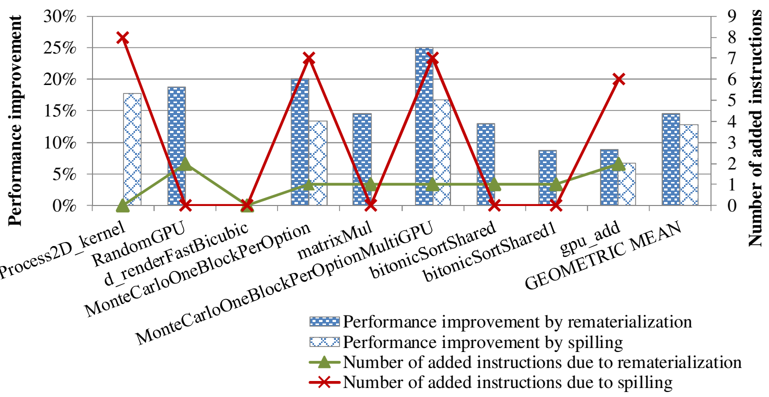


Fig. 10. The performance improvement and the number of instructions inserted for each kernel using register rematerialization/spilling.

no benefit in terms of reducing register usage so as to increase the number of active thread blocks and thereby to improve performance, while, in an other case (*spProcess2D_kernel*), register rematerialization fails to do so. *d_renderFastBicubic* is a case that none of both schemes is helpful to performance enhancement. The numbers of instructions being inserted also explain that register rematerialization has better performance improvement than register spilling since the numbers of instructions inserted due to register rematerialization are less than those due to register spilling when both schemes take effect.

Figure 11 displays the expected performance improvement due to the increased number of active thread blocks (assuming that the register usage of the kernels can be reduced without any extra instructions to be inserted) and the performance improvement of register rematerialization, register spilling, and the combination of register rematerialization and spilling for the nine kernels. It shows that the combination of register rematerialization and spilling takes advantage of the two schemes and performs as well as the best of both schemes. The execution time of the kernels was reduced by a geometric average of 14.8%. An interesting case is that for kernel *d_renderFastBicubic*, the scheme of register rematerialization or register spilling alone failed to reduce its execution time since the register usage of the kernel could not be decreased to a specified degree so as to make one more thread block active, but the combination of the two schemes did have 15.1% of performance improvement since register rematerialization and register spilling usually eliminate live range of different variables and the interleaving processes of them explores more opportunities to decrease the register usage.

Performance Improvement of GPU Time for CUDA Programs The results in the previous section demonstrate the effectiveness of the proposed framework for kernels that may benefit from reducing their register usage so as to increase one extra active thread block, but some CUDA programs consist of multiple kernels and some of the kernels might not be optimizable. We evaluated the performance improvement of GPU time for CUDA programs to further examine the proposed framework.

Figure 12 shows the performance improvement of overall GPU time for each benchmarks using register rematerialization, register spilling, and combination of register rematerialization and spilling, respectively. As expected, the combination of the two

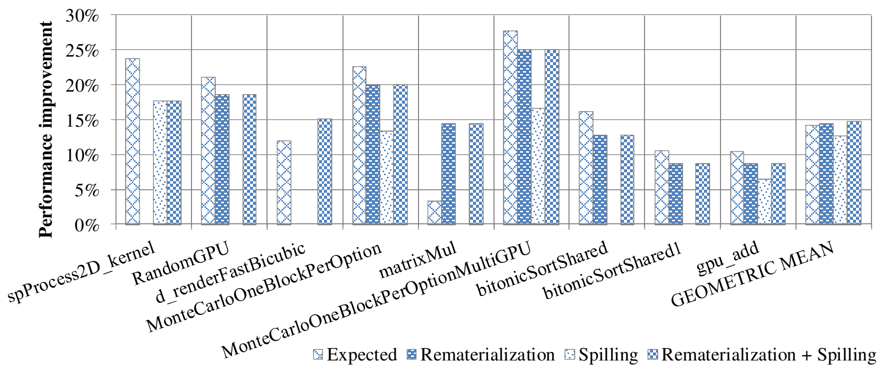


Fig. 11. The expected performance improvement and the performance improvement of register rematerialization, register spilling, and the combination of register rematerialization and spilling for each kernel.

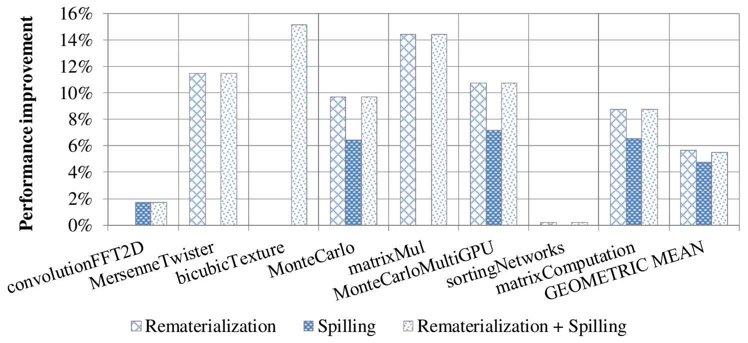


Fig. 12. The performance improvement of GPU time for each CUDA program using register rematerialization, register spilling, and combination of register rematerialization and spilling, respectively.

schemes has the most performance improvement and offers about 10% or more of enhancement for most benchmarks. Unfortunately, the proposed framework did not work well for two benchmarks (*convolutionFFT2D* and *sortingNetworks*) and thus the geometric average of the reduction in overall GPU time is decreased to 5.5% and arithmetic average is decreased to 9.0%. The results seem to show that the TLP-aware register-pressure reduction framework is not effective to many kernels, but this is attributed to the implementation of the closed-source *ptxas* assembler. As mentioned in Section 2.4, the *ptxas* assembler involves optimizations that counteract the efforts done in the framework, so the effectiveness of the proposed framework might be decreased by the *ptxas* assembler because *ptxas* is not aware of these optimizations. Therefore, we investigated the kernels that that may benefit from the proposed framework (i.e., the 44 kernels mentioned in Section 3.1) to examine how many kernels became unoptimizable due to the *ptxas* assembler, while the register usages of these kernels were indeed decreased to the desired degree by the framework. Figure 13 shows the investigation result and indicates that there are only 18.2% (8 of 44) of kernels on which the proposed framework took no effect, while 58.7% (27 of 44) kernels could be optimized by the framework if the *ptxas*

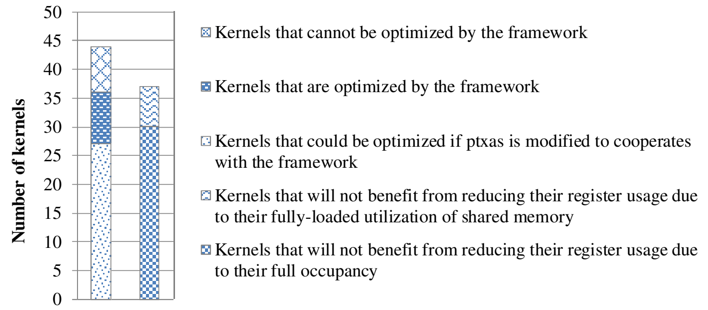


Fig. 13. The number of kernels that may benefit or not from reducing their register usage with regard to having an extra active thread block.

assembler is modified to cooperates with it. As a result, the TLP-aware register-pressure reduction framework should be applicable to more than 44.4% (36 of 81) kernels.

4 Related Work

In recent years, parallel computing attracts more attention because of the development of multi-core processors and, especially, heterogeneous multi-core systems (CPU+GPU). Many researchers attached great importance to use compiler optimization techniques to gain better performance. Some researches focused on managing hardware resources to provide better utilization and operations of these resource. Stratton et al. proposed a framework, called MCUDA, which consists of a set of source-level transformations and a runtime system, which allow CUDA programs to be executed efficiently on shared memory and multi-core CPUs [10]. Ueng et al. proposed CUDA-lite, which involves a source-to-source translator, to generate CUDA programs that access global memory in a coalesced manner in order to relieve the burden on programmers [11]. Han and Abdel-rahman proposed hiCUDA, a source-to-source translator, which produces better CUDA code in comparison with CUDA-lite. Their framework lightens the programmers' burden of programming global-memory-access code in a coalescing way, of packaging GPU code in separate functions and of managing data transfer between the main memory and global memory [4]. Yang et al. presented a novel optimizing compiler to address two major challenges of developing CUDA programs: effective utilization of GPU memory hierarchy and judicious management of parallel threads [12]. However, none of these work addressed the issue on managing "registers", which are also an important fact that influences the TLP of CUDA programs.

Until recently, Dominguez et al. proposed a framework that reduces the register pressure of CUDA programs to increase the resource occupancy and thereby gain the performance speed-up of CUDA programs [3]. The objective of this concurrent research is similar to ours. Both of the frameworks use techniques of rematerialization to reduce register usages, but Dominguez et al.'s framework only rematerialize memory operations that load function parameters and special registers (e.g. threadId, blockId), while our framework rematerialize not only values from memory but also values that are computed by common instructions. In addition, we introduced the concept of using shared memory as spilling space to further reduce the register usage. Moreover, we proposed an approach to determine a minimum number of registers to economize for each kernel so as to minimize the negative effect on ILP, whereas in Dominguez et al.'s framework, register usages are reduced until no further changes can be done, and it is highly possible that the decreased ILP counteracts the benefits from the increased TLP. The evaluations demonstrate that our framework improved the performance for benchmarks by an arithmetic average of 9.0% (15.1% in a best case), while only one benchmark achieved 4% of performance improvement in Dominguez et al.'s framework.

5 Conclusion

In this paper, we proposed a framework, called TLP-aware register-pressure reduction framework, to reduces the register usage of threads so as to increase the number of active thread blocks and thereby the TLP. The framework involves a simple cost model to determine a minimum number of registers to economize for each kernel so as to minimize the negative effect on the ILP and two register-pressure reduction schemes: (1) register rematerialization that splits live ranges of variables by recomputing values of variables and (2) register spilling to shared memory that utilizes the unused space of shared memory to speed up the spilling processes. The experimental results indicate that the proposed framework is effective in reducing the GPU execution time of CUDA programs, while the scheme of register rematerialization has less overhead than that of register spilling. The evaluation shows that the framework produced an geometric average performance improvement of 5.5% for all examined benchmarks.

References

1. D. Bernstein, M. Golumbic, Y. Mansour, R. Pinter, D. Goldin, H. Krawczyk, and I. Nahshon. Spill Code Minimization Techniques for Optimizing Compliers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'89, pages 258–263, 1989.
2. P. Briggs, K. D. Cooper, and L. Torczon. Rematerialization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'92, pages 311–321, 1992.
3. R. Dominguez, D. R. Kaeli, J. Cavazos, and M. Murphy. *Improving the Open64 Backend for GPUs*. Google Summer of Code (GSoC), October 2009.
4. T. D. Han and T. S. Abdelrahman. hiCUDA: A High-level Directive-based Language for GPU Programming. In *Proceedings of the 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 107–116, 2009.
5. The IMPACT Research Group. *Parboil Benchmark suite*, 2009.
6. NVIDIA Cooperation. *NVIDIA CUDA Developer Zone*, April 2008.
7. NVIDIA Cooperation. *NVIDIA CUDA SDK*, April 2008.
8. NVIDIA Cooperation. *NVIDIA GeForce GTX 285 Architecture Overview*, March 2009.
9. NVIDIA Cooperation. *NVIDIA CUDA C Programming Guide, Version 3.2*, October 2010.
10. J. A. Stratton, S. S. Stone, and W. mei W. Hwu. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs. In *Proceedings of the 21th International Workshop on Languages and Compilers for Parallel Computing*, LCPC'08, pages 16–30, 2008.
11. S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W. mei W. Hwu. CUDA-lite: Reducing GPU Programming Complexity. In *Proceedings of the 21th International Workshop on Languages and Compilers for Parallel Computing*, LCPC'08, pages 1–15, 2008.
12. Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU Compiler for Memory Optimization and Parallelism Management. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'10, pages 86–97, 2010.