Unified On-chip Memory Allocation for SIMT Architecture

Ari B. Hayes and Eddy Z. Zhang
Department of Computer Science
Rutgers University
Piscataway, NJ 08554
{arihayes, eddy.zhengzhang} @cs.rutgers.edu

ABSTRACT

The popularity of general purpose Graphic Processing Unit (GPU) is largely attributed to the tremendous concurrency enabled by its underlying architecture – single instruction multiple thread (SIMT) architecture. It keeps the context of a significant number of threads in registers to enable fast "context switches" when the processor is stalled due to execution dependence, memory requests and etc. The SIMT architecture has a large register file evenly partitioned among all concurrent threads. Per-thread register usage determines the number of concurrent threads, which strongly affects the whole program performance. Existing register allocation techniques, extensively studied in the past several decades, are oblivious to the register contention due to the concurrent execution of many threads. They are prone to making optimization decisions that benefit single thread but degrade the whole application performance.

Is it possible for compilers to make register allocation decisions that can maximize the whole GPU application performance? We tackle this important question from two different aspects in this paper. We first propose an unified on-chip memory allocation framework that uses scratch-pad memory to help: (1) alleviate single-thread register pressure; (2) increase whole application throughput. Secondly, we propose a characterization model for the SIMT execution model in order to achieve a desired on-chip memory partition given the register pressure of a program. Overall, we discovered that it is possible to automatically determine an on-chip memory resource allocation that maximizes concurrency while ensuring good single-thread performance at compile-time. We evaluated our techniques on a representative set of GPU benchmarks with non-trivial register pressure. We are able to achieve up to 1.70 times speedup over the baseline of the traditional register allocation scheme that maximizes single thread performance.

ICS'14, June 10–13 2014, Munich, Germany.
Copyright 2014 ACM 978-1-4503-2642-1/14/06 ...$15.00.
http://dx.doi.org/10.1145/2597652.2597685.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*code generation, compilers, optimization*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*

General Terms

Performance, Management

Keywords

GPU; Register Allocation; Shared Memory Allocation; Compiler Optimization; Concurrency

1. INTRODUCTION

Existing compilation techniques for on-chip memory resource allocation, including register allocation, mainly target single-thread performance. In the past several decades, efficient techniques have been studied and widely adopted in mainstream compilers. In the context of single instruction multiple threads (SIMT) architecture for general purpose Graphic Processing Unit (GPU), the whole program performance not only depends on single thread performance, but also the interaction between the group of threads that run concurrently – mainly the process to hide each other's latency caused by execution dependence, data request, synchronization and other reasons. The number of concurrent threads depends on the physical on-chip memory constraint as well as the per-thread on-chip memory demand from a given program. The latter mainly depends on compile-time decision. The traditional register allocation technique for CPU program tends to gives the maximal number of physical registers to a single thread according to its register pressure.

The goal of traditional register allocation technique is to minimize the number of register spills and maximize single thread performance. However, this strategy does not necessarily work well for programs running on SIMT architecture. Allocating registers according to a single thread's register pressure may lead to resource contention among concurrently executing threads and lead to sub-optimal performance. We show this phenomenon using the results of a case study over a set of important GPU applications in physics simulation, numerical analysis, and image processing [6] [20]. We control per-thread register count at compile-time and we compile one program into different versions over a range of register usage from 20 or 32[1]to the maximal register de-

[1]We choose 20 for Fermi and 32 for Kepler because this leads to the maximal number of concurrent threads. Having

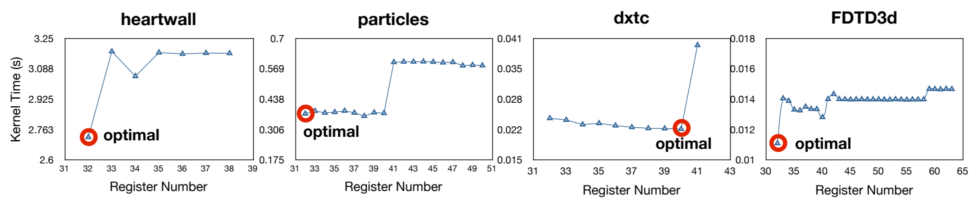

Figure 1: Performance of compiled programs with various per-thread register usage. The *x-axis* represents the per-thread register usage for every compiled version. The *y-axis* shows the running time in seconds. We used an NVIDIA Kepler [21] GTX680. The range of register count is from 32 to the maximal register demand.

mand. We show the results of four benchmarks *heartwall*, *particles*, *dxtc* and *FDTD3d* in Fig. 1. As can be seen in Fig. 1, using as many registers as needed to avoid spilling does not necessarily yield optimal performance. Furthermore, using the smallest number of registers per-thread does not necessarily yield best performance either. For instance, the *dxtc* benchmark's best per-thread register count is 40, which is too high to allow maximal occupancy, yet lower than the register demand.

A fundamental question arises: what is the best per-thread register usage for a given program on a given GPU architecture? An intuitive approach is to try all possible register count, compile the program and profile performance over different runs. However, exhaustive search is prohibitive when the range of possible register counts is large. It can be up to 255 for current GPUs[21]. Furthermore, depending on the input parameters, the number of profiling runs needed for a fully representative input set can be exponential.

To have a good answer to this fundamental question, we need to understand the implications of compile-time on-chip memory allocation decision on the efficiency of concurrent execution. Per-thread register usage, as well as other on-chip memory usage, is tightly correlated with the concurrency level a program can achieve. Using fewer registers per thread may lead to high concurrency, but more local memory loads and stores due to register spills. The local memory resides in DRAM, which has large access latency. How can we minimize per-thread register usage while maintaining good single-thread performance? And how can we strike a balance between the benefits brought by high concurrency and the overhead brought by extra off-chip memory operations?

In this paper, we study the implications of concurrent execution on many-core GPUs and exploit these implications to develop efficient compile-time on-chip memory allocation strategies. We address the above challenges from two main aspects. We first propose a unified on-chip memory allocation framework that not only uses registers but also on-chip scratch-pad memory, to store thread context. The scratch-pad memory acts as a buffering layer between registers and off-chip memory, alleviates single-thread register pressure, and increases the concurrency level. We develop a novel inter-procedure scratch-pad memory allocation scheme that maximizes reuse across procedure boundaries, and implement a prototype on-chip memory allocator. To ensure compatibility, we only use scratch-pad memory not already allocated by the user. Secondly, we characterize the relationship between the program performance and the concurrency level. Our characterization predicts the desired concurrency level for a given program, and guides the selection of per-thread register count and scratch-pad memory usage.

There is a large body of research work on register allocation for CPUs and embedded processors [4] [5] [23] [12] [22] [3] [7]. They have shown promising results for single-thread applications. Some studies investigated scratch-pad memory allocation techniques on embedded architectures [9] [17] [26], but not on GPUs equipped with a bulk synchronous parallel (BSP) execution model. Gebhart and others [11] proposed techniques to dynamically partition on-chip memory into cache, scratch-pad and register memory according to application's register/scratch-pad memory demand with architecture support. However, it does not address the problems of how to reduce and how to determine register pressure for a given program. Overall, there is a lack of exploration in the implications of compile-time on-chip memory allocation on the concurrent execution efficiency of GPU applications.

In this paper, we propose efficient on-chip memory allocation techniques to enable maximal utilization of many-core GPU processors. We summarize our contributions as follows:

- **On-Chip Memory Allocation** We build an unified on-chip memory allocation framework for GPU applications. We offload register pressure to scratch-pad memory when necessary and we determine the corresponding per-thread register and scratch-pad memory usage for maximal concurrency level. Under this framework, we develop a novel inter-procedure on-chip memory allocation strategy, which maximizes the reuse of on-chip memory across procedure boundaries.

- **Concurrency-oriented Program Analysis** We reveal that severe resource contention can be caused by static memory resource allocation for GPU programs. For the first time, we address the problem of mapping GPU program features to its achievable concurrency and its desirable concurrency level. We propose efficient characterization model to determine if increased concurrency level will always yield better whole program performance. Our model is a pure static model and yet it is effective.

- **Implemented Allocator for Real GPU Systems** We reverse engineered the NVIDIA hardware ISA and

a smaller per-thread register count does not improve concurrency; it only degrades single thread performance, leading to worse performance overall.

implemented our prototype on-chip memory allocator for programs that run on real GPUs. Our approach can be readily deployed and does not require any architecture level extension.

The rest of the paper is outlined as follows: Section 2 reviews background on GPU programming model. Section 3 details our unified on-chip memory allocation framework. Section 4 describes program characterization and concurrency selection techniques. Section 5 and Section 6 respectively analyzes experiment results and presents related work.

2. BACKGROUND

Although the GPU as a whole acts as a single instruction multiple thread (SIMT) processor, with different threads following different execution paths, it has small groups of threads execute in lockstep, in the manner of a single instruction multiple data (SIMD) processor. Such a SIMD-like processor is called a Streaming Multi-Processor (SM) in NVIDIA terminology. We use NVIDIA terminology to describe GPU architecture throughout this paper. A group of threads that run in lockstep on one SM is called a thread warp. A thread warp is the minimal scheduling unit on every SM. When one thread warp yields an SM, if there is another ready thread warp, it will be scheduled to run. Otherwise the SM processor remains idle until one thread warp is ready. Typically there is a much larger number of active warps than the total number of SMs. All of their states are saved in registers. When one thread warp is swapped out of the processor, its states remain in registers. When one thread warp is switched in to run, it does not need to load its states from off-chip memory into registers unless per-thread register allocation is not enough to hold its state. Therefore it is different from a traditional CPU ”context switch”. A set of warps form a block, whose threads share access to the same partition of scratch-pad memory. A set of blocks forms a grid, which is launched by the same function. A function that runs on GPU is called a *kernel function* .

There are two types of memory on a GPU card – on-chip memory and off-chip memory. On-chip memory includes registers, scratch-pad memory, and caches. Registers are the fastest on-chip memory storage. Every SM has a large register file and it is divided evenly among co-running threads. Since every thread executes the same *kernel function*, it uses the same number of registers. At one time, only a limited number of threads can run simultaneously due to hardware constraints on the size of register and scratch-pad memory. We refer to these threads as *active* threads. A *kernel function* typically launches a significant number of threads which are further partitioned into multiple batches of *active* threads. A batch does not yield the SMs until all threads within it complete execution. Only thread warps within the same batch can co-run and help hide each other's instruction latency. Every GPU architecture with different computing capabilities also specifies the maximal hardware allowed *active* threads per SM (*active* is used in NVIDIA terminology, however we denote these threads as concurrent threads throughout this paper). In NVIDIA terminology, the ratio between the actual number of active threads and the hardware limit is defined as *occupancy*, which is a number between 0 and 1.

Another important type of on-chip memory is scratch-pad memory. It is referred to as *shared memory* in CUDA. For the rest of the paper, we use *shared memory* to refer to GPU scratch-pad memory. The shared memory can be managed explicitly by software. It is fast, with a latency of several cycles, comparable to the L1 cache.Shared memory is also equally partitioned among different threads.

Off-chip memory includes *global memory* and *local memory*, which might be a hundred times slower than registers. *Local memory* is used to store local variables for every procedure, and is manageable during compile-time. If a register is spilled to off-chip memory, it resides in local memory. *Global memory* can be explicitly managed by programmers. Other types of off-chip memory include *constant* memory, *texture* memory and etc. They are used for special purposes, such as read-only memory storage or multi-dimensional data locality.

3. UNIFIED ON-CHIP MEMORY ALLOCATION

3.1 Framework Overview

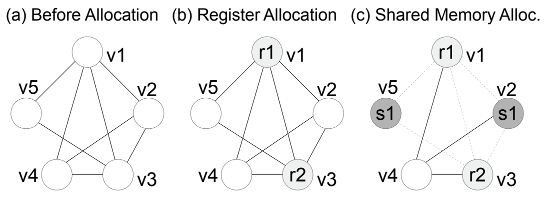

Figure 2: Unified On-Chip Memory Allocation

In this Section, we describe our transformation framework that uses shared memory to store local variables and to alleviate register pressure, which ultimately leads to better GPU concurrency and whole program performance. This framework uses shared memory to store live variables that cannot fit in registers, as if we are spilling registers into shared memory. We perform register allocation first and select the variables that can stay in registers. For the rest of the variables, we perform shared memory allocation and choose a subset of them to be stored in shared memory.

The target of the unified on-chip memory allocation is to store as many local variables into a fixed number of registers and shared memory slots as possible. We treat both registers and shared memory as one type of memory – the on-chip memory. Then we perform on-chip memory allocation as if we are performing register allocation for traditional CPU programs. We illustrate this idea in Fig. 2. Assume we have two registers, one shared memory slot, and one local memory slot. In Fig. 2 (a), we show the interference graph of five local variables *v1, v2, v3, v4, v5*. Two variables interfere with each other if they are both live at one or more instructions. It implies the two variables cannot be assigned to the same register or shared memory slot. If two variables interfere, there is an edge between the nodes representing them. In Fig. 2 (b), we show the result after register allocation. Variables *v1* and *v2* are assigned to registers *r1* and *r2* respectively. Now we have three variables that are not assigned and we have one shared memory slot. In Fig. 2 (c), we assign variables *v2* and *v5* to shared memory slot *s1*. By this step, we have completed assigning as many variables as we can to registers and shared memory. We then let the last variable *v4* stay in local memory. The minimal number of variables to be stored in local memory is 1 in this case.

Register allocation techniques have been extensively studied in the past three decades [4] [5] [23] [12] [22] [3] [7]. However, they mainly focus on single procedure register allocation. A few of them [15] [8] have studied reuse of registers across procedures but mainly focus on minimizing register pressure penalty at procedure calls. Typically, the content of most of the registers in the caller procedure are saved in local memory at procedure calls so that the registers can be reused in the callee procedure. Previous work [15] [8] avoid saving all the registers when procedure calls happen by determining if the registers will be used or not in the callee procedure. We leverage the register allocation algorithms for single procedure on-chip memory allocation and we develop an algorithm that maximizes reuse of on-chip memory across procedure boundaries. We describe this approach in Section 3.2.

In summary, with a given number of registers and shared memory, our unified on-chip memory allocation framework performs both register allocation and shared memory allocation. To separate the coupling effects from other phases of compilation, we build a experiment platform that takes binary as input. As in the binary file, the other phases like instruction scheduling have already completed, and we can simply replace the live variable accesses as shared memory access or off-chip memory accesses and add corresponding instructions. Therefore, the only effect we are testing is the placement of live variables. We use the binary generated by *nvcc* with a fixed register count. Then we analyze the other variables that are spilled into local memory and transform them correspondingly given a fixed number of shared memory slots. The NVIDIA GPU hardware instruction set architecture (ISA) and application binary interface (ABI) is proprietary. We reverse engineered part of the ISA and ABI for CUDA computing capability 3.0 with information from the open source project *asfermi* [13] on CUDA computing capability 2.0. We are able to decode the instructions, parse the assembly code and perform data flow analysis. In the following section, we elaborate inter-procedure shared memory allocation.

3.2 Shared Memory Allocation

3.2.1 Inter-procedure Shared Memory Reuse

We start describing our technique on enhancing inter-procedure reuse of shared memory with an example. Note that CUDA does not allow objects with virtual functions to be parameters, and every CUDA kernel we have seen has a call graph which can be determined statically.

We first show that there are opportunities to reuse shared memory slots across procedure calls. In Fig. 3, we show the stack status of a call sequence that involves three procedures: *proc_A, proc_B, proc_C* . *proc_A* calls *proc_B*. *proc_B* calls *proc_C*. Assume the stack memory space for every procedure can hold exactly four different variables. This means at any instruction in this procedure, at most four variables can be live at the same time. The variables that are live when *proc_A* calls *proc_B* are stored in locations *La1, La3*. In *proc_B*, when *proc_C* is called, variables are saved in locations *Lb2* and *Lb3*. In Fig. 3, we first show what the call stack looks like with traditional CPU procedure local memory management – labeled as *traditional*. The cells with dark

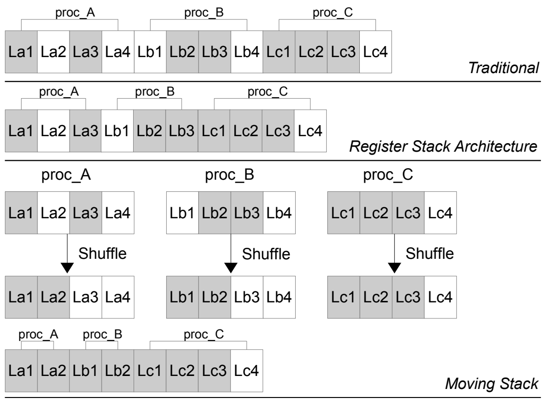

Figure 3: Reuse shared memory across procedure boundaries

background represent that the variable in the corresponding location is live when another procedure is called. With the traditional approach, we can see that the local memory space of different procedures in the call sequence is stacked. Therefore, for these three procedures, the size of the local memory space needed in the call context is 12 slots, assuming one slot can hold one variable. Even if some variables are not live when a procedure is called, the memory stack is incremented from its original maximal stack depth. This is because the size of local memory for a CPU procedure is trivial compared to the size of the off-chip memory.

In the second approach described in Fig. 3, we show a CPU architecture that utilizes a register stack when an architecture has relatively large register space to hold callee-saved registers across multiple procedures [7]. This is only made possible with special architecture support [7]. This architecture utilizes the available register region at the end of the register stack when a procedure is called. In the subfigure marked as *Register Stack Architecture* in Fig. 3, *proc_B* can reuse the last available slot in *proc_A's* stack, and its stack pointer starts from the end of *La3*. Similarly, *proc_C* can reuse the last slot in the stack for *proc_B*. In this case, we use 10 slots in the stack. This approach is related to inter-procedure register usage, but it can be applied to shared memory allocation across procedure boundaries. It saves 2 slots compared to the approach denoted as *Traditional* in Fig. 3. However, there are still slots that are not used when *proc_B* and *proc_C* are called.

In the third approach described in Fig. 3, we show our approach – *Moving Stack* approach, which minimizes unused stack space when there are nested procedure calls. Our resource allocator emits instructions to be inserted in the original binary, which shuffles variables in the stack so that the used variables will be stored in consecutive memory space. Then we emit code to shift the stack pointer before the callee procedure is invoked. For instance, in Fig. 3 *Moving Stack* section, when *proc_B* is called, the variable in *La3* is shuffled to the second slot. Then we let *proc_B* use the space from *La3*. Similarly, when *proc_B* calls *proc_C*, we move the variable in *Lb3* to *Lb1*. Therefore *proc_C* can use the third slot in *proc_B's* stack space, which is the 6th slot in the overall runtime stack space. In this case, we use 8 slots in total and no local memory slot is wasted. Compared to the original case that uses 12 slots, we save more than 30% stack space.

3.2.2 *Inter-procedure Shared Memory Assignment*

In the last section, we presented an approach to maximize the reusability of shared memory slots across procedures. If we have a large amount of shared memory to hold all local variables, then we can directly start assigning shared memory slots to individual variables. However, the shared memory is a scarce resource, as its size is the same or even smaller than the register file size. Therefore, we need to select a subset of local variables to reside in shared memory. Meanwhile, we need to determine how many shared memory slots every procedure gets assigned. Then we can perform shared memory assignment on a per-procedure basis.

In this Section, we describe our approach to map selected local memory variables to shared memory variables. We define *Live-on-exit* to be set of variables live at the exit of an instruction. *Max-live* is defined as the maximum number of simultaneously live variables at the exit of an instruction. *Max-live* of a procedure that does not call any other procedure is easy to acquire. We can traverse all instructions in the procedure and pick the largest *Live-on-exit* set. For procedures that call other procedure calls, we propose a recursive approach built on the following idea. We obtain the number of live variables for an instruction that calls another procedure P_{callee} as the sum of its local |*Live-on-exit*| and *Max-live*(P_{callee}). If the *Max-live* of the callee procedure is unknown, we recurse into the callee procedure to find its *Max-live*.

Assume we have N_{smem} available shared memory slots. If *Max-live* of the main GPU kernel function is greater than N_{smem}, we need to prune at least *Max-live* - N_{smem} variables from the *Live-on-exit* sets and let these variables reside in local memory. Our heuristic approach ranks different variables based on a pre-defined priority function. We prune low priority variables until the updated *Max-live* is less than or equal to N_{smem}. This approach is simple, yet effective.

We rank local variables from different procedures and give them a global ranking. We first define a *composition instruction*. It is a list of 2-tuples used to specify a call sequence. If the instruction $inst_2$ at $func_0$ calls $func_1$, and instruction $inst_3$ at $func_1$ calls $func_2$, and the specific executing instruction in $func_2$ is $inst_0$, then the resulting composition is $\{ (func_0, inst_2), (func_1, inst_3), (func_2, inst_0) \}$. The call context information exposed in a composition instruction helps keep track of caller instructions so that we can obtain the *Live-on-exit* set easily from a union of live variables at all relevant instructions in this calling context. Then we can compare these variables from different procedures as if they are from the same procedure.

Our inter-procedure variable pruning algorithm takes the following steps:

- **Step 1:** We find the set of all composition instructions whose *Live-on-exit* $> N_{smem}$. We call it the *Over-smem-limit* set.
- **Step 2:** For all live variables in the union of *live-on-exit var* sets of composition instructions in the *Over-smem-limit* set, we compute their priority values based on the priority function. We use the priority function of variable frequency in the union live variable set.
- **Step 3:** We eliminate one variable from the above set with lowest priority value and check whether *Max-live* is less than or equal to N_{smem} after this variable is eliminated from all *Live-on-exit* sets. If it is, then we go to Step 4. Otherwise we go back to **Step 3**.
- **Step 4:** We have successfully pruned all the necessary variables. We return the set of variables that are candidates to be placed in shared memory.

Individual Shared Memory Slot Assignment.

The eliminated variables are the ones that stay in local memory and the rest are mapped to shared memory. With this information, we can compute up-to-date *Max-live* for every procedure again. This is used as the maximal number of shared memory slots assigned to every procedure. Then we perform shared memory slot assignment in a way similar to register allocation. We use a heuristic graph coloring approach that starts with the node of highest degree in the interference graph. We assign this node a shared memory slot that does not conflict with any of its neighbors that are already assigned. If there are multiple choices, then we choose the shared memory slot that was previously assigned to some other variable. We process every node. If a variable cannot be assigned to any shared memory slot without conflicting with its neighbors, we map it to local memory. If the interference graph has a chordal property, then we will not have any spills [22]. In most cases, we don't need to spill any shared-memory mapped variable into local memory.

4. GPU PROGRAM OCCUPANCY CHARACTERIZATION

Our transformation framework in Section 3 tackles the problem of minimizing local memory spills given a fixed amount of registers and shared memory. What would be the best amount of registers and shared memory to allocate for every running thread in any given GPU program? Given a typically much larger number of registers than on CPUs, usually in the scale of tens of thousands, we have many possible combinations of register count and shared memory consumption per thread. In this large search space, exhaustive search is prohibitive. In this section, we address the problem of finding best per-thread register and shared memory usage.

The number of registers and the amount of shared memory used per-thread determine the number of concurrent threads on every streaming processor. The number of concurrent threads can be estimated using the formula [2] below:

$$\begin{aligned} &Active.Thread \\ &= min(\frac{Total.Reg.Num}{PerThread.Reg.Num}, \frac{Total.Smem}{PerThread.Smem}). \end{aligned}$$

Essentially, the specific questions on per-thread register and shared-memory usage all boil down to one fundamental question: what would be the most desirable concurrency level for any GPU program on a specific GPU architecture? If we know the best concurrency level, we can estimate per-thread register and shared memory usage by solving the above equation. The optimal concurrency level has

[2]The total number of threads is also bound by the register bank alignment and the thread block sizes for CUDA programs. This formula illustrates the idea that per-thread register/shared-memory usage dominates the number of concurrent threads. We use the GPU occupancy calculator [19] to get the accurate number of active threads based on all other factors in our experiments.

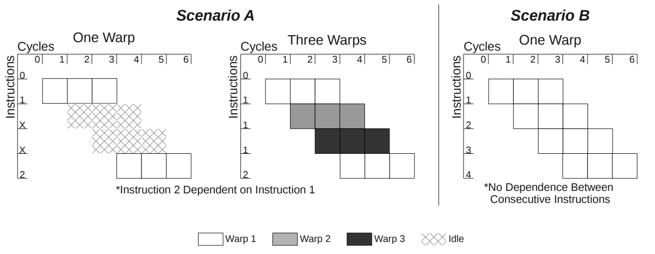
Scenario A
One Warp
Three Warps
Scenario B
One Warp
Cycles
Instructions
*Instruction 2 Dependent on Instruction 1
*No Dependence Between Consecutive Instructions
Warp 1
Warp 2
Warp 3
Idle

Figure 4: Concurrency Level Sensitivity

the capability of overlapping different types of operations and minimizing computing unit idleness. The number of different operations and how much they can be overlapped depends on the characteristics of a program. The problem of desired concurrency level is thus closely related to the problem of GPU program characterization in a many-thread cooperation/contention context. We describe our characterization approach first and concurrency level determination algorithm secondly.

4.1 Model Many-thread Running Process

We can start from the minimal concurrency level indicated by the maximal register/shared-memory request in the program, keep increasing the concurrency level by spilling live variables into shared memory and/or local memory, and keep increasing concurrency level until the point that the overhead of extra local memory spilling cannot be offset by the benefit brought by increased program concurrency. However, increasing concurrency level does not alway help; in fact it helps for most programs except one special case.

This exception is the case which we define as *computation intensive* case; it is when single thread instruction level parallelism (ILP) is inherently good in the program so that the latency is well hidden when a thread is running by itself. We illustrate it with an example in Fig. 4 *Scenario B.* In Fig. 4, assume every instruction takes three cycles, and the processor is able to dispatch one instruction in every cycle. The *x-axis* represents the cycle number. The *y-axis* represents the instruction number. In *Scenario B*, instruction 2 does not directly depend on instruction 1, and it can be dispatched immediately at the beginning of the second cycle. It is similar for instructions 2, 3, and 4. We only need one thread per-core in *Scenario B* to fully utilize the computation pipeline (one thread warp for one SM). In this cases, an increased concurrency level does not help improve performance, and they are not concurrency-bound cases. Next we show a concurrency-bound case in which increased concurrency helps improve performance. In Fig. 4 *Scenario A*, assume we have dependences between instruction 2 and instruction 1. Instruction 2 can't start until instruction 1 finishes. If we have only one warp, we cannot dispatch instruction 2 until the beginning of the fourth cycle in the *One Warp* case of *Scenario A*. The processor pipeline is thus not fully utilized. However, if we have three warps, in the *Three Warps* case in *Scenario A* of Fig. 4, at the beginning of the second cycle, we can schedule warp 2 to run instruction 1, and at the beginning of the third cycle, we can schedule warp 3 to run instruction 1. Therefore the processor pipeline is fully utilized. Overall, in *Scenario A*, we need three warps to fully utilize the computing units. Increasing the number of concurrent thread warps from one to three helps improve performance. These cases belong to the concurrency-bound category.

In summary, we want to optimize concurrency-bound programs with multi-level on-chip memory resource allocation strategies. We use a heuristic metric to distinguish between programs that can benefit from increased concurrency and the ones that do not necessarily benefit from increased concurrency. The metric is the average dispatch interval between every two adjacent instructions in the same thread specified statically in the GPU binary code. The average dispatch interval reflects the ratio between the idle cycles and the busy cycles in the pipeline. As illustrated in Fig. 4 *scenario A* and *scenario B*, the dispatch intervals are 3 and 1 respectively. The minimal number of warps to fully utilize the processor pipeline happens to be 3 and 1 for these two cases respectively. The average dispatch interval [3] can also be used as an initial estimate of the number of active warps for every SM. We elaborate our algorithm for concurrency selection in next section.

4.2 Concurrency Level Search

The main idea of our concurrency level search algorithm is to make the benefits of increased concurrency outweigh the overhead of local memory spilling. With limited registers and shared memory, increasing the concurrency level may force live variables to be spilled into slow local memory. How many local memory spills can be allowed depends on the concurrency level we select. We start from an initially estimated number of active threads as the product of average dispatch interval and the number of cores per SM (every SM is the same so we discuss how to find concurrency level for every SM). Based on the initial estimate, we derive the number of registers and the amount of shared memory for every thread. We then perform shared memory allocation with the initial per-thread register number and shared memory amount. Then we check the number of local memory spills to see if we should decrease or increase the concurrency level. In this algorithm, we use a heuristic criteria to check whether a given concurrency level is good enough; we keep increasing concurrency level above the initial concurrency level if the criteria is met, or keep decreasing concurrency level below the initial concurrency level until the criteria is met. We set the criteria in a way that the local memory spilling overhead can be overlapped with the arithmetic and other non off-chip memory instructions. We use $COMPUTE\_inst_{trsf}$ to denote the number of instructions that are not off-chip memory instructions after transformation and we use MEM_inst_{trsf} to denote the number of off-chip memory instructions after transformation. We refer to this criteria as Computation Interleaving predicate – CI_pred and we describe it as follows:

$$CI_{pred} : \frac{COMPUTE\_inst_{trsf} * AD\_int}{MEM\_inst_{trsf}} > MAX\_cmratio \tag{1}$$

[3]NVIDIA kepler GPU architecture uses a static instruction scheduling approach instead of architecture based instruction scheduling. Dispatch interval is encoded in the instructions if we compile with nvcc for computing capability of 3.0 and above. We reverse-engineered the ISA and parsed instructions to get the dispatch interval for every two adjacent instructions.

```
Find Desired concurrency Level
if ( Program is concurrency Bound) {
    $ActTnum = AD\_interval * SM\_cores$;
(Reg,Smem) = getRegSmem(ActTnum);
    transformProg(Reg,Smem);
    get(CI_pred);
    if (CI_pred) TraverseUp=TRUE;
    else TraverseUp = FALSE;
    $ActTnum\_cur = ActTnum$;
    if ( TraverseUp ) {
        while($ActTnum\_cur < MaxThrd$) {
            get(CI_pred);
            if ( CI_pred )
                ActThrd_opt = ActTnum_cur;
        ActTnum_cur += BlkSize; }
      else if($ActTnum \geq SmemFitTnum$) {
        while($ActTnum\_cur > ActTnum\_org$) {
            get(CI_pred);
            if (CI_pred ) {
                ActThrd_opt = ActTnum_cur;
                break; }
            $ActTnum\_cur -= BlkSize$; }
else
    ActThrd_opt = ActTnum_org;
return ActThrd_opt;
```

Figure 5: Concurrency Level Search

AD_int denotes the average instruction dispatch interval. We obtain this by decoding NVIDIA Kepler's binary ISA. *MAX_cmratio* is correlated with the number of cycles for an off-chip memory instruction; it is the number of computation instructions with a specific dispatch interval that are needed to hide the latency of one off-chip memory instruction. Its value varies from architecture to architecture. We obtain this parameter value by measuring the cycles of computation and off-chip memory instructions for a specific architecture. If the condition in Inequality 1 is satisfied, we consider this concurrency level to be beneficial. In our concurrency level search algorithm, if the initially estimated concurrency level is beneficial, we keep increasing it step by step (and we use thread block size as the step since it is the minimal unit to run on a SM). We choose the largest concurrency level which is beneficial. Otherwise, we keep decreasing the concurrency level step by step until we find the first concurrency level that is beneficial.

We illustrate the major components of our concurrency level search algorithm in Fig. 5. Note that when the algorithm traverses down, it stops when it hits a threshold *SmemFitTnum*. This is the number of concurrent threads which results in no spills into the off-chip memory, which means the live variables can completely fit into registers and shared memory. The variable *ActTnum*, denotes the number of active threads per SM, and the variable *ActThrd_opt* denotes the final number of active threads per SM we selected.

5. EVALUATION

In this section, we present our experiment results. We perform experiments on two different machine configurations. One is NVIDIA Kepler GTX680. It has 8 streaming multiprocessors (SM), with 192 cores on each of them and 1536 cores in total. It has CUDA computing capability 3.0. Every streaming multi-processor is equipped with 65536 registers and 48KB shared memory. The maximum number of concurrent threads that can run simultaneously on each streaming multi-processor is 2048. The second machine is configured with NVIDIA Fermi card - Tesla C2075. It has 448 cores in total, with 32 cores on each SM. It has CUDA computing capability 2.0. There are 32768 registers and 48KB shared memory per SM. The maximal number of concurrent threads that can run simultaneously is 1536. Notice that these two configurations impose different constraints on single-thread register count and single-thread shared memory with respect to maximal concurrency supported by hardware. We denote the Kepler card as *Kepler* and the Fermi card as *Fermi*.

We measured computation and off-chip memory instructions latencies with the *clock()* function. Normal algebra instructions like addition and subtraction take 9 cycles and an off-chip memory instruction takes between 300 and 400 cycles. Since reads and writes happen in parallel, we set the average of off-chip memory latency to be between 150-200. Therefore, we choose the larger one 200 and set the parameter *MAX_cmratio* in Section 4.2 to be 200. This parameter is used in our automatic occupancy level selection algorithm, and our experiments support this value as being effective.

To process a benchmark, we first extract the assembly code for the kernel function using NVIDIA binary listing tool *cuobjdump*. We decoded necessary parts of the binary instruction set for NVIDIA Kepler architecture, including scheduling instructions omitted by *cuobjdump*, based on *asfermi* [13]. Our binary analysis and modification pass is implemented with the *libelf* library. We implemented our parser with *flex and bison.*Our shared memory allocator then performs program analysis, determines the best occupancy level, and transforms the code. We use one fixed register as a shared-memory stack pointer. If necessary, we use a second fixed register to shuffle shared-memory slots for *Moving Stack* algorithm. Note that this process is done quickly, and takes less than a second on most benchmarks.

We evaluate our methods with seven benchmarks selected from the Rodinia benchmark suite 2.2 [6] and CUDA SDK 5.0. We choose them because they have non-trivial register demand. Note that a lot of benchmarks from Rodinia [6] and CUDA Computing SDK have a low register demand of below 20, which happens to enable maximal hardware supported concurrency for previous and current NVIDIA GPU architectures. Decreasing register pressure for these benchmarks will not help improve concurrency or improve single-thread performance. Our algorithm will choose not to transform these programs, thus we do not include them in discussion. We describe the list of benchmarks used for this paper in Table 1. *RegDemand* is the number of registers needed per-thread if no spilling to on-chip or off-chip memory happens. It is the default choice by nvcc and traditional CPU register allocation approach. *UserSmem* is the bytes of shared memory preallocated by the user per thread. Note that we only use the remaining shared memory left after users' preallocation, and we do not affect the existing concurrency when distributing the available shared memory among concurrent threads. *InstChange* is the increase in size to the transformed kernel function at the auto occupancy.

Benchmark	AppDomain	RegDemand		UserSmem		InstChange		CacheMissRate(%)	
		Fermi	Kepler	Fermi	Kepler	Fermi	Kepler	Fermi	Kepler
cfd [6]	Simulation	61	63	0.00	0.00	1.00	1.12	90.49/90.49	0.00/83.68
dxtc [20]	Imaging	43	49	10.00	12.00	1.03	1.00	32.81/54.39	0.00/0.00
FDTD3d [20]	Numerical Ana.	57	48	7.50	10.00	1.05	1.10	0.00/0.00	0.00/0.00
hotspot [6]	Simulation	36	39	12.00	12.00	1.03	1.03	0.00/48.26	0.00/43.09
imageDenoising [20]	Imaging	63	63	4.00	4.00	1.29	1.06	0.00/72.99	0.00/37.05
particles [20]	Simulation	50	52	0.00	0.00	1.09	1.12	0.00/44.44	0.00/43.41
recursiveGaussian [20]	Imaging	41	42	0.00	0.00	1.00	1.07	0.00/83.69	72.77/90.09

Table 1: Benchmark Description. AppDomain is the benchmark's application. RegDemand is the number of registers the compiler tries to use. UserSmem is the amount of user-allocated shared memory per thread. InstChange is the increase to the instruction count in the auto occupancy. CacheMissRate is the cache miss rate in the auto occupancy; the numerator is with use of shared memory after transformation, and the denominator is with purely global spills before transformation.

CacheMissRate lists cache miss rates for the auto occupancy before and after transformation.

We present both the results of automatically selected occupancy level through our approach, and the exhaustive search through all possible occupancy levels. When an occupancy level is selected, per-thread register and shared-memory limits are determined by the NVIDIA GPU occupancy calculator [19].

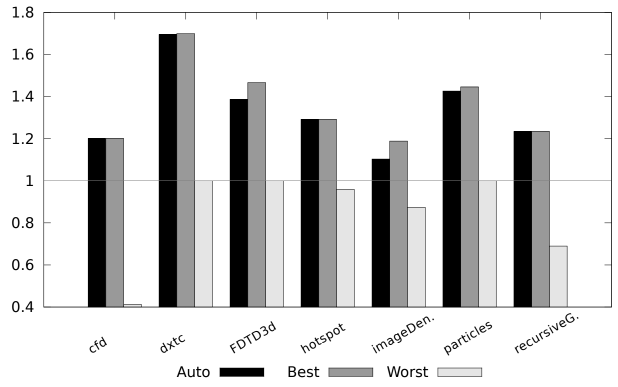

Figure 6: Kepler Performance Results. The Auto bar shows speedup with automatically selected occupancy. The Best bar shows highest speedup among all occupancies by exhaustive search. The Worse bar shows worst speedup among all occupancies. For our baseline, we used the runtime when compiling each benchmark with default settings.

We first present the overall performance results for Kepler in Fig. 6. Each group of bars along the x-axis represents a benchmark. The y-axis represents a particular kernel's speedup compared to its baseline. For our baselines, we compiled each benchmark using nvcc with the default settings, including no register limits. The first bar *Auto* represents the speedup at the concurrency level selected by our concurrency selector. The second bar *Best* represents the best speedup among all possible concurrency levels. The third bar *Worst* represents the speedup in the worst case among all different concurrency levels. The results demonstrate that the automatically transformed program is typically faster than the original, and in most cases is close to the best speedup (from an exhaustive search through concurrency levels). *FDTD3d* and *imageDenoising* fail to reach their best speedup due to our conservative algorithm, which avoids the highest occupancies in this case due to the number of static memory operations. In our future work, we plan to incorporate dynamic analysis in order to allow more optimal selections. Overall, although no dynamic analysis is performed, we still have good performance improvement for these benchmarks. This demonstrates the importance of static on-chip memory resource allocation.

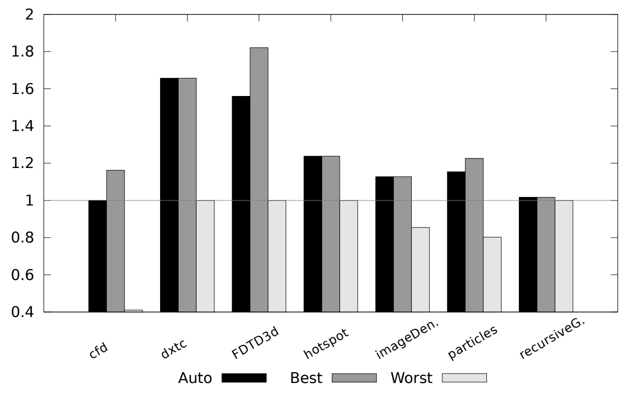

Figure 7: Fermi Performance Results. The Auto bar shows speedup with automatically selected occupancy. The Best bar shows highest speedup among all occupancies by exhaustive search. The Worse bar shows worst speedup among all occupancies. For our baseline, we used the runtime when compiling each benchmark with default settings.

Next we present the results for the Fermi GPU in Fig. 7. For our baseline, we compiled the benchmark using nvcc with the default settings, including no register limits except where necessary for the benchmark to run (due to hardware limitations). The bars and axes have the same meaning as in Fig. 6. Only the *cfd* benchmark here is not improved; this is because *cfd* has an unusually large number of memory instructions, even at the lowest occupancy, and so our conservative algorithm chooses not to increase the occupancy at all. Due to the many memory instructions, increasing the occupancy has less effect than in most benchmarks, regardless. The *particles* and especially the *FDTD3d* benchmarks also have auto speedups below their bestdue to the conservative algorithm choosing a lower occupancy than is optimal in these cases. The *recursiveGaussian* benchmark sees much less improvement than on Kepler. This has to do with the differing limitations of the hardware. On Kepler, a kernel can use up to 63 registers regardless of its block size, but on Fermi, programs with higher block size have a lower register limit, due to the smaller register file. Having a high block size, *recursiveGaussian* must be compiled with less registers to run at all, increasing the initial occupancy, and therefore lessening the extent to which it can be improved.

6. RELATED WORK

Many studies in the past few years have been proposed on register spilling between physical registers and off-chip global memory. A fundamental model is the graph coloring model [4]. In [2], the authors propose an integer linear program modeling of register allocation for CISC machines. In [27] and [14], the authors particularly tackled the problem of register spilling due to software pipelining in loops. Most of these previous studies are for sequential programs, instead of massively parallel architecture. In [1], the authors studied the register allocation schemes for vector machines. However, a vector processor is different from the SIMT processor on modern GPUs. Sampaio and others [24] proposed a divergence aware spilling strategy to save memory, but did not consider concurrency.

The previous studies on GPU also investigate the implication of interaction between concurrent threads on latency minimization. The authors of [25] point out that the ability for memory latency hiding among different vector thread groups is critical. The authors present a model for GPU programs that predicts the performance by calculating *memory warp parallelism (MWP)* and *computation warp parallelism (CWP)*. While this work focused on modeling of concurrent execution, it did not discuss how to achieve the desired concurrency level. Other relevant GPU work includes topics such as GPU exception handling [18], where register states need to be restored for resuming execution after exception, and energy saving [10], where the location of registers is critical to energy consumption because the distance between the registers and processors determines the amount of energy consumed during data movement, and hardware register space saving [29],which combines SRAM and DRAM to store more bits into the die area. In [28], a means of optimizing shared memory is explored in order to prevent user-allocated shared memory from reducing occupancy, whereas our approach makes use of non user-allocated shared memory to lessen the cost of improving occupancy. In [16], an integer programming technique is used to allocate scalars and arrays in shared memory, in order to optimize the code at a higher level than we consider. Most of the aforementioned studies on GPU architecture extensions are implemented and evaluated in hardware simulators.

7. CONCLUSION

In this paper, we propose a unified on-chip memory resource allocation framework for GPU programs. Our on-chip memory resource framework predicts near-optimal partition of on-chip memory resources and adapts GPU program to the best concurrency level according to program characteristics without any online or off-line profiling.

Acknowledgements

We thank Dmitry Mikushin for his insights on decoding NVIDIA Kepler hardware instruction set architecture. We thank the anonymous reviewers for their invaluable comments. This material is based upon the work supported by Rutgers University Research Council Grant and the Department of Education GAANN fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

8. 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] I. D. Baev, “Techniques for region-based register allocation,” in *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 147–156. [Online]. Available: http://dx.doi.org/10.1109/CGO.2009.31

[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] Y. Choi and H. Han, “Optimal register reassignment for register stack overflow minimization,” *ACM Trans. Archit. Code Optim.*, vol. 3, no. 1, pp. 90–114, Mar. 2006. [Online]. Available: http://doi.acm.org/10.1145/1132462.1132467

[8] F. C. Chow, “Minimizing register usage penalty at procedure calls,” in *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, ser. PLDI ’88. New York, NY, USA: ACM, 1988, pp. 85–94. [Online]. Available: http://doi.acm.org/10.1145/53990.53999

[9] A. Dominguez, N. Nguyen, and R. K. Barua, “Recursive function data allocation to scratch-pad memory,” in *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, ser. CASES ’07. New York, NY, USA: ACM, 2007, pp. 65–74. [Online]. Available: http://doi.acm.org/10.1145/1289881.1289897

[10] 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

[11] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally, "Unifying primary cache, scratch, and register file memories in a throughput processor," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 96–106. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2012.18

[12] 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.

[13] Y. Hou, J. Lai, and D. Mikushin, "asfermi: An assembler for the nvidia fermi instruction set." [Online]. Available: http://code.google.com/p/asfermi/

[14] 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

[15] G.-Y. Lueh and T. Gross, "Call-cost directed register allocation," in *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, ser. PLDI '97. New York, NY, USA: ACM, 1997, pp. 296–307. [Online]. Available: http://doi.acm.org/10.1145/258915.258942

[16] W. Ma and G. Agrawal, "An integer programming framework for optimizing shared memory use on gpus," in *High Performance Computing (HiPC), 2010 International Conference on*. IEEE, Dec 2010, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/HIPC.2010.5713187

[17] R. McIlroy, P. Dickman, and J. Sventek, "Efficient dynamic heap allocation of scratch-pad memory," in *Proceedings of the 7th international symposium on Memory management*, ser. ISMM '08. New York, NY, USA: ACM, 2008, pp. 31–40. [Online]. Available: http://doi.acm.org/10.1145/1375634.1375640

[18] 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

[19] NVIDIA, "Cuda occupancy calculator." [Online]. Available: http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

[20] Nvidia, "Gpu computing sdk." [Online]. Available: https://developer.nvidia.com/gpu-computing-sdk

[21] NVIDIA, "Nvidia's next generation cuda compute architecture: Kepler gk110." [Online]. Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[22] J. Palsberg, "Register allocation via coloring of chordal graphs," in *Proceedings of the thirteenth Australasian symposium on Theory of computing - Volume 65*, ser. CATS '07. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2007, pp. 3–3. [Online]. Available: http://dl.acm.org/citation.cfm?id=1273694.1273695

[23] 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

[24] D. N. Sampaio, E. Gedeon, F. M. Q. a. Pereira, and S. Collange, "Spill code placement for simd machines," in *Proceedings of the 16th Brazilian conference on Programming Languages*, ser. SBLP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 12–26. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33182-4_3

[25] 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

[26] S. Udayakumaran, A. Dominguez, and R. Barua, "Dynamic allocation for scratch-pad memory using compile-time decisions," *ACM Trans. Embed. Comput. Syst.*, vol. 5, no. 2, pp. 472–511, May 2006. [Online]. Available: http://doi.acm.org/10.1145/1151074.1151085

[27] 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

[28] Y. Yang, P. Xiang, M. Mantor, N. Rubin, and H. Zhou, "Shared memory multiplexing: A novel way to improve gpgpu throughput," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 283–292. [Online]. Available: http://doi.acm.org/10.1145/2370816.2370858

[29] 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