Orion: A Framework for GPU Occupancy Tuning

Ari B. Hayes
Rutgers University
Piscataway, NJ 08854
United States
arihayes@cs.rutgers.edu

Lingda Li
Rutgers University
Piscataway, NJ 08854
United States
lingda.li@cs.rutgers.edu

Daniel Chavarría-Miranda
Pacific Northwest National Lab
Richland, WA 99354
United States
Daniel.Chavarria@pnnl.gov

Shuaiwen Leon Song
Pacific Northwest National Lab
Richland, WA 99354
United States
Shuaiwen.Song@pnnl.gov

Eddy Z. Zhang
Rutgers University
Piscataway, NJ 08854
United States
eddy.zhengzhang@cs.rutgers.edu

Abstract

An important feature of modern GPU architectures is *variable occupancy*. *Occupancy* measures the ratio between the actual number of threads *actively* running on a GPU and the maximum number of threads that can be scheduled on a GPU. High-occupancy execution enables a large number of threads to run simultaneously and to hide memory latency, but may increase resource contention. Low-occupancy execution leads to less resource contention, but is less capable of hiding memory latency. Occupancy tuning is an important and challenging problem. A program running at two different occupancy levels can have three to four times difference in performance.

We introduce ORION, the first GPU program occupancy tuning framework. The ORION framework automatically generates and chooses occupancy-adaptive code for any given GPU program. It is capable of finding the (near-)optimal occupancy level by combining static and dynamic tuning techniques. We demonstrate the efficiency of ORION with twelve representative benchmarks from the Rodinia benchmark suite and CUDA SDK evaluated on two different GPU architectures, obtaining up to 1.61 times speedup, 62.5% memory resource saving, and 6.7% energy saving compared to the baseline of optimized code compiled by *nvcc*.

Categories and Subject Descriptors

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

Keywords

GPU Compiler; Occupancy Tuning; Register Allocation; Shared Memory Allocation; Concurrent Program Compilation

Middleware'16, December 12-16, 2016, Trento, Italy
© 2016 ACM. ISBN 978-1-4503-4300-8/16/12...$15.00
DOI: http://dx.doi.org/10.1145/2988336.2988355

1. INTRODUCTION

GPU performance tuning and optimization is challenging because of the complexities in GPU architecture and the massive scale of threads in its simultaneous execution environment – a GPU typically runs 10,000s of active threads at one time.

Previous GPU tuning frameworks exploit different factors to improve performance. The PORPLE framework [7] helps programmers determine which type of memory to use with respect to data access patterns. Liu and others [15] leverage input-sensitivity to select the best program compilation and execution parameter. Many studies have focused on domain-specific performance tuning due to the complexities in tuning general-purpose applications. Anand and colleagues [25] explored the tuning of computation transformation parameters and data representation for sparse matrix code. The Halide [23] framework targets image processing applications and focus on tuning the locality and parallelism parameters.

In this paper, we introduce *occupancy tuning* for GPU programs. *Occupancy* is an important performance tuning factor that is unique to GPU programs. *Occupancy* [18] is defined as the ratio between the number of threads *active* at one time and the maximum number of threads the GPU hardware can schedule. A program running at two different occupancy levels can have up to three or four times difference in performance. We show an example in Figure 1. We use the imageDenoising program from CUDA SDK [20] and show its performance at different occupancy levels. We normalize the performance with respect to the best occupancy running time (at 50% occupancy). It can be seen from Figure 1 that the difference in running time between the best and worst cases can be up to three times, indicating the importance of occupancy tuning.

Occupancy tuning is tightly coupled with resource allocation. Efficient occupancy tuning requires efficient resource allocation. The occupancy level is controlled by tuning the amount of on-chip memory assigned to every thread [19] [11]. High occupancy leads to high contention, high latency-hiding capability, and high resource allocation pressure (every thread gets less resources). Low occupancy results in low contention, low latency-hiding capability, and low resource allocation pressure (every thread gets more resources). Tuning occupancy and determining the best trade-off point is

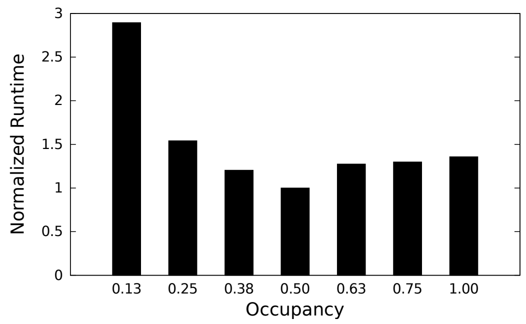


Figure 1: Running time v.s. occupancy for the imageDenoising benchmark on GTX680. Occupancy is between 0.125 and 1.0.

challenging, as it depends on multiple factors including compile-time resource allocation efficiency, dynamic program behavior, and execution configuration.

In this paper, we design and implement the first occupancy tuning framework for GPU programs – the ORION framework. ORION determines the most desirable occupancy level and generates the occupancy-adaptive code for any GPU program by combining iterative static compilation and dynamic program adaptation. It has two components: a compiler that generates and selects binaries at several different occupancy levels, and a runtime adaptation component that chooses one of these binaries at execution time. The ORION compiler narrows down the search of best occupancy level to five or fewer possibilities, and the runtime component selects the occupancy level that adapts to dynamic program behavior. These two stages work together to provide users with the code running the best occupancy level.

Previous work that models the relationship between GPU occupancy and performance does not address the problem of determining and achieving the best occupancy. The GPU performance model proposed by Hong and others [13] uses off-line profiled information to predict the performance of a GPU program. While the prediction method by Hong and others provides satisfactory accuracy, it requires fine-grained information based on an architecture simulator. And it does not provide pro-active occupancy tuning solution. Previous work for GPU resource allocation optimizes per-thread resource allocation. The studies in [24] and [11] alleviate GPU per-thread register allocation pressure via static compile-time transformation. However, the static optimization does not adapt to runtime program behavior. Furthermore, while alleviating register pressure does indirectly increase occupancy, it is not necessarily true that higher occupancy is always better than lower occupancy [26]. Overall, efficient GPU program execution requires a systematically exploration of both single thread performance and concurrent thread dynamics.

In this paper, we develop the **o**ccupancy-o**ri**ented tuning and **on**-chip resource allocation (ORION) framework. We are not aware of any prior work that systematically explores the influence of occupancy tuning for GPU programs. Our contributions are summarized as follows.

- ORION is the first occupancy tuning framework that taps into both single-thread resource allocation and concurrent thread interaction.
- It combines static and dynamic occupancy tuning, enabling a fast and accurate search for the best occupancy. (Section 3.3 and 3.4).
- ORION's compiler provides an inter-procedure resource allocation model that is rigorously proved to be optimal in both memory space and movement cost (Section 3.2).
- ORION not only improves performance – up to 1.61 times speedup – but also resource & energy efficiency, with up to 62.5% memory resource saving, and 6.7% energy reduction over the highly optimized code generated by *nvcc* (Section 4).
- ORION is immediately deployable on real systems and does not require hardware modification.

2. BACKGROUND

GPUs deliver high performance via massive multithreading through the single instruction multiple thread (SIMT) execution model. Every thread runs the same code on different input sets. GPU threads are organized into *thread warps*. A thread warp, the minimum execution unit, typically consists of 32 threads. Thread warps are further organized into thread blocks. A GPU is composed of multiple SMs. A thread block runs on at most one SM. The number of active threads on one SM is a multiple of the thread block size.

GPU on-chip memory includes registers and cache. *Shared memory* is the software-managed cache in NVIDIA terminology. We use NVIDIA terminology throughout the paper. Access to shared memory is explicitly managed by software. Every active thread gets an even partition of register file and shared memory. The amount of registers and shared memory used by every thread determines how many threads can be active at one time (the occupancy). The hardware-managed cache is L1/L2 cache. Unlike registers and shared memory, hardware cache usage does not impose any constraints on the occupancy.

GPU *occupancy* [18] is defined as the ratio between the actual number of active thread warps and the maximum number of thread warps the hardware can schedule. The occupancy can be calculated using per-thread register usage, per-thread shared memory usage, and thread block size. At runtime, occupancy is set by the GPU driver, based on these parameters.

Assume in a program every thread uses V_{reg} register space and V_{smem} shared memory space. The total register file size is N_{reg} and the total shared memory size is N_{smem}. The maximum number of threads the hardware can schedule at one time is S_{max}. The formula below gives the occupancy.

$$\text{Occupancy} = \text{Min}({}^{N_{\text{reg}}}/_{V_{\text{reg}}}, {}^{N_{\text{smem}}}/_{V_{\text{smem}}})/S_{max}. \quad (1)$$

Since the number of active threads needs to be rounded up to a multiple of thread block size, and the register partition needs to be aligned according to register bank size constraints, the occupancy may be smaller than above. We use the formula in NVIDIA occupancy calculator [19] to obtain the accurate occupancy.

A GPU program consists of both CPU code and GPU code. The code that runs on the GPU side is organized into

GPU kernels. A *GPU kernel* is a function that runs on the GPU. Every GPU kernel embodies an implicit barrier since all threads that are launched by this kernel needs to finish before the next kernel starts. We perform occupancy tuning for GPU kernels only.

3. ORION SYSTEM DESIGN

The ORION system automatically tunes occupancy. The design of ORION addresses two important questions respectively. The first question is how to generate the code given any occupancy level. Since one occupancy level gives the number of registers (shared memory) every thread can use – see Equation (1) – it implies a register allocation and register allocation can only be done at compile time. How to perform register allocation efficiently at compile-time and adapt it to GPU program characteristics is critical.

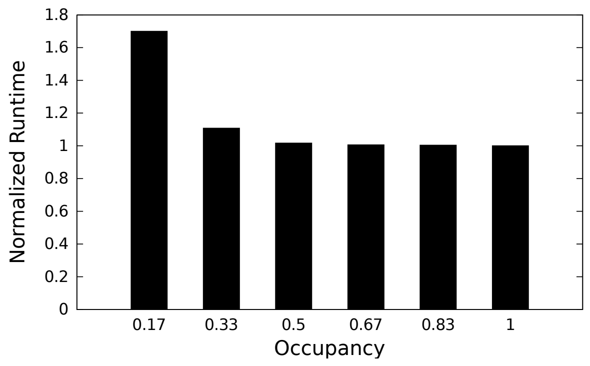


Figure 2: Effect of occupancy on performance for matrixMul

The second question is how to choose the best code version out of all the code versions corresponding to different occupancy levels.We use two-level selection: compile-time tuning and runtime tuning. The ORION compiler selects a set of candidate occupancy levels that potentially correspond to the optimal occupancy level. The compiler generates multiple candidate versions because the dynamic factors (such as inputs, control divergence, irregular memory accesses) might affect the choice of best occupancy level in a way that no single occupancy is always the best. Therefore we let the runtime component chooses the best occupancy code to use and to adapt to complex runtime program behavior.

The design and implementation of the dynamic occupancy selection process is based on two *principles.* Note that we discovered that there are two patterns of performance variation with respect to occupancy changes. When concurrency increases, the performance increases, until it reaches a point that benefits from concurrency improvement cannot offset the performance overhead due to individual thread's resource allocation pressure (recall that individual thread gets less resource when occupancy increases). From this point, there are two patterns: 1) the performance deteriorates, as illustrated in Figure 1, and 2) the performance plateaus, as illustrated in the matrix multiplication program in Figure 2, due to the fact that the program itself does not have much register pressure.

Therefore, **the first principle** is based on the fact that the occupancy optimization function is a function that has only one local minimum in running time. In other words, the local minimum is also the global minimum, which makes the optimization problem of occupancy tuning much simpler. We can start from any initial occupancy level, find the right tuning direction, and then move step by step, until we reach the optimal occupancy case.

The second principle is based on the observation for the matrix multiplication example in Figure 2, that when we reach best performance occupancy, we should keep tuning until we find the range of occupancy levels that yields performance similar to the best performance. In Figure 2, from occupancy 50% to 100%, the performance is similar. Therefore, we can potentially lower the occupancy, increase resource usage per thread for other purposes (e.g. caching or loop unrolling), and improve intra-thread performance significantly. If we know the range in which performance is stable, we know the safe occupancy reduction range without hurting performance. This coincides with the discovery made in [26] in 2010, which demonstrated that by lowering occupancy and applying optimizations made possible by the reduced concurrency, *matrixMul* can achieve superior performance. However, the work in [26] only looked at matrix multiplication and also it did not give an approach on how to tune the occupancy levels. In our design of occupancy tuning, we not only find the point where the performance is best, but also find the range of occupancies where the performance is best, in particular identifying the lowest occupancy that gives the best performance.

Next, we discuss the detailed design and implementation of the ORION compiler and runtime adaptation component.

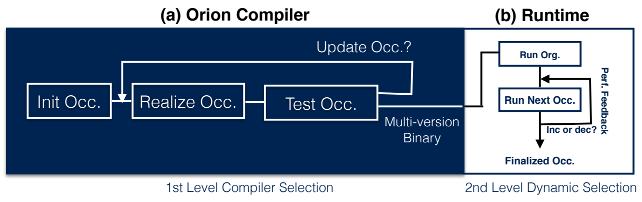


Figure 3: The ORION compiler and runtime collaboration

3.1 Overview

The overall ORION framework works as follows (Figure 3). In the first step, the compiler sets an initial occupancy level, generate corresponding code and determines the occupancy tuning direction: increasing or decreasing based on the performance model in Section 3.3. The initial occupancy is defined such that all live values fit into the minimal number of registers, or the maximum registers per thread limit (by hardware) has been reached. Then the compiler performs occupancy testing, if the occupancy needs to be increased or decreased, the ORION compiler performs on-chip memory allocation, assigning register and shared memory per-thread to achieve the updated occupancy. And we repeat the *testing and updating process* until a termination condition is met, defined by our performance model in Sec 3.3. The compiler generates and selects ≤ 5 different code versions for a GPU kernel function and in most cases ≤ 3 versions (in evaluation), which helps the runtime quickly adapt to the best version. Runtime adaption is shown in Figure 3 (b).

There are three major stages in the ORION framework.

- The *realizing occupancy* stage (Section 3.2) ensures that the code generated to achieve a certain occupancy level is efficient. For instance, it avoids excessive spilling from on-chip memory (register and shared memory) to off-chip memory.

- The *compile-time occupancy tuning* stage (Section 3.3) ensures that we select a small set of kernel binaries with good occupancy levels, ruling out the versions that are unlikely to perform well.
- The *runtime occupancy adaptation* stage (Section 3.4) ensures that we select the best kernel to execute at runtime, and also that we avoid aggressive optimization.

Every stage is important. The first two stages are performed at compile-time, and the last stage is performed at runtime.

3.2 Realizing Occupancy

To realize a certain occupancy, we bound the number of registers and the size of shared memory used per thread. We first represent a program in the Static Single Assignment (SSA) form, in which every variable is defined once and only once. Then we generate the pruned SSA form to eliminate ϕ functions. Next we start assigning the pruned SSA variables, first placing them into registers with spills into local memory, and then reassigning a subet of local memory variables to shared memory. Since every thread executes the same binary code, allocating for one thread is equivalent to allocating for all the threads. We call the commensurate amount of space for a 4-byte register in on-chip memory (including shared memory and cache) an *on-chip memory slot.* A variable can be placed into register, shared memory, or L1 cache (via local memory).

Minimizing Space Requirement.

The on-chip memory space to store the variables should be minimal since if it does not fit into on-chip memory space, there will be spilling into off-chip DRAM memory, which is significantly slower. Therefore the first thing we need to optimize is the on-chip memory space needed for a set of live variables.

Optimal register allocation for *single procedure* has been studied extensively in CPU literature, and the technique can be applied to on-chip memory allocation. We adopt the Chaitin-Briggs register allocator [3] and build a variant of it by taking into consideration the wide variables (64-bit, 96-bit, or 128-bit) that need consecutive and aligned registers. We use a stack to track the priority of variables to be allocated (colored). Our single-procedure allocation algorithm is detailed in Figure 4.

However, there is limited inter-procedure allocation research on GPUs. While functions can be in-lined, it is not practical to in-line every function. For instance, after aggressive inlining by the nvcc compiler, the *cfd* program still has 36 static function calls (please see Evaluation Section 4 Table 2). Moreover, although certain GPU programs exhibit no procedure calls in source code, there are still function calls in the binary. An example is the intrinsic *division* function, which is implemented as a function call for GPU architecture. This may appear frequently in scientific programs in Rodinia [6] benchmark suite that use the floating-point division function.

We propose an allocation algorithm for multi-procedure GPU kernels which is **optimal in terms of both space and data movement requirements**.

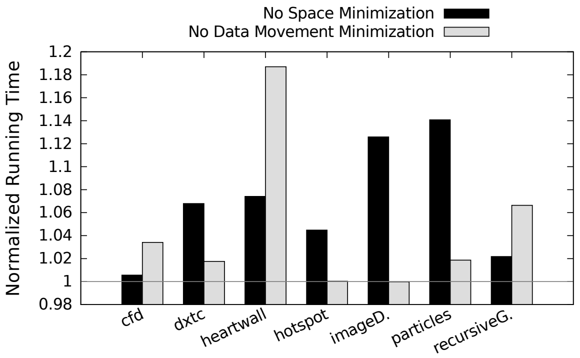


Figure 5: Optimized v.s. unoptimized inter-procedure allocation. The benchmarks are from Rodinia benchmark suite and CUDA SDK.

We describe our inter-procedure allocation algorithm as a *compressible stack*. The idea of the *compressible stack* is simple. With *compressible stack*, right before entering a sub-procedure, we compress the stack by coalescing the used on-chip slots such that the sub-procedure can use the maximum number of contiguous on-chip memory slots. Right after the sub-procedure returns, we restore the location of moved slots back to their original locations so that the program can continue execution correctly.

We show how the compressible stack works with an example in Figure 6. In Figure 6 (a), we use the column S_i to represent an on-chip memory slot. After single procedure allocation, one or more variables are mapped to one on-chip memory slot. We show the live range of every variable using the vertical black bar in Figure 6 (a). The liveness information indicates the used/unused status of an on-chip memory slot S_i at different call points in the program. The left “Program” column represents the code of the program including the procedure calls. If no variable is mapped to an on-chip memory slot at an execution point, the slot is unused. For instance, S3 is unused when *foo2* is called in Figure 6 (a).

In Figure 6 (a), before calling *foo1*, we move the slot that contains variable “var5” into the slot between the slots that contain variables “var3” and “var1” so that we have larger contiguous free space for procedure foo1. This is important since a sub-procedure uses contiguous stack space. Right after the sub-procedure returns, we restore the location of variable “var5” to resume execution.

We demonstrate that it is important to minimize space for inter-procedure allocation by showing the performance difference between the space minimize version and the space unoptimized version in Figure 5. The “no-space minimization” bar corresponds to the unoptimized version and the running time is normalized to the optimized one.

Minimizing Data Movement.

The above shows how to minimize on-chip memory space used across procedure calls. This comes at the cost of increased data movements. Therefore, minimizing data movement is also important.

In CPU single procedure allocation literature, there has been work that proposes to trade-off data movement for space. The chordal graph coloring model [21] is a well known model that minimizes register usage while increasing data

(a) Input parameters:
G: the interference graph; each node is a variable.
C: the number of colors (physical registers)

(b) Stack Order:

```
S = empty stack
while G is nonempty
  nextVar = null
  for each variable v in G
    if v.width + v.edges <= C
      if (nextVar==null ||
          nextVar.width > v.width)
            nextVar = v
  if nextVar==null
    nextVar = first variable in G
    for each variable v in G
      if (nextVar.width > v.width ||
         (nextVar.width==v.width &&
         nextVar.edges > v.edges))
           nextVar = v
  S.push(nextVar)
  G.remove(nextVar)
```

(c) Coloring pseudocode:

```
s = S
while s is nonempty
  v = s.pop()
  usedColors = {}
  for each colored variable u in v.edges
    usedColors = usedColors U u.color
  for c = 0 to C - v.width
    if {c, ..., c + v.width - 1} U usedColors == {}
      v.color = {c, ..., c + v.width - 1}
      break
  if v was colored
    s.remove(v)
  else
    S.remove(v)
    spillList.add(v)
    s = S
```

Figure 4: Single procedure multi-class allocation alg.

movements for removing ϕ-functions. Later work by Hack and other further optimize data movements caused by removal of ϕ-functions [9]. However, there is no such work in minimizing data movements for inter-procedure allocation. As far as we know, our work is the first one that minimizes data movement for inter-procedure on-chip memory allocation.

We first show an example of how data movements can be reduced. Figure 6 (b) is the same as Figure 6 (a) except that the addressing of the on-chip memory slots is different. For the original layout in Figure 6 (a) at the point call(foo1), the sub-procedure $foo1$ needs to use three consecutive slots, thus var5 in S4 needs to be copied to S2. In Figure 6 (a), altogether three data movements are necessary before entering three call points call(foo1), call(foo2) and call(foo3) as indicated by the three arc arrows for “before call”. However, if we place the variables of the original $S2$ slot to the location of original $S4$, $S3$ to $S2$, $S4$ to $S3$, as illustrated in Figure 6 (b), the total number of data movements is reduced to 1 at *call(foo2)* point, while at call(foo1) and call(foo2), all available on-chip memory slots are contiguous.

Our compressible stack optimizes the layout by changing the address of physical on-chip memory slots. We provide a polynomial time algorithm that finds the optimal address mapping. We achieve this by modeling the problem as a *maximum-weight bipartite matching* problem [28].

Let N be the number of static sub-procedures calls in the procedure of interest. M represents the number of variable sets, SS_i, assigned during the single-procedure graph coloring (allocation) process such that $i = 0...M-1$ and each SS_i is mapped to one on-chip memory slot.

We denote the number of data movements incurred for entering/leaving the k-th procedure call as P_k^{mov}. The objective is to find the physical on-chip memory slot each variable set SS_i for $i = 0...M-1$ is mapped to such that the total number of data movements is minimal.

Table 1: Notations

Notation	Description
SS_i	set of variables mapped to the i-th stack slot
$SLOT_i$	i-th slot from the bottom of the stack
X_{ij}	mapping between SS_i and j-th slot (0 or 1)
L_{ik}	liveness of SS_i at k-th sub-procedure call
B_k	desired stack height at k-th sub-procedure call
C_{ijk}	# of swaps incurred by placing SS_i at j-th slot for $k-$th sub-procedure
W_{ij}	# of swaps incurred by placing SS_i at j-th slot
N	# of sub-procedure calls
M	maximum # of simultaneously live variables in this procedure
P_k^{mov}	# of swaps incurred because of k-th sub-procedure call

$$\textbf{min } T\_mov = \sum_{k=0}^{N-1} P_k^{mov}$$

We model this problem as a *maximum-weight bipartite matching* problem by making use of the following Theorem.

Theorem 1. *With the notations defined in Table 1, in the minimal-mov-assignment (MMA) problem, the total number of data movements contributed by placing an arbitrary variable set SS_i at an arbitrary location $j-th$ memory slot $SLOT_j$ across all k immediate sub-procedures is a constant. We define this number as W_{ij}. Assume at the k-th sub-procedure call we need to bound the compressed caller stack*

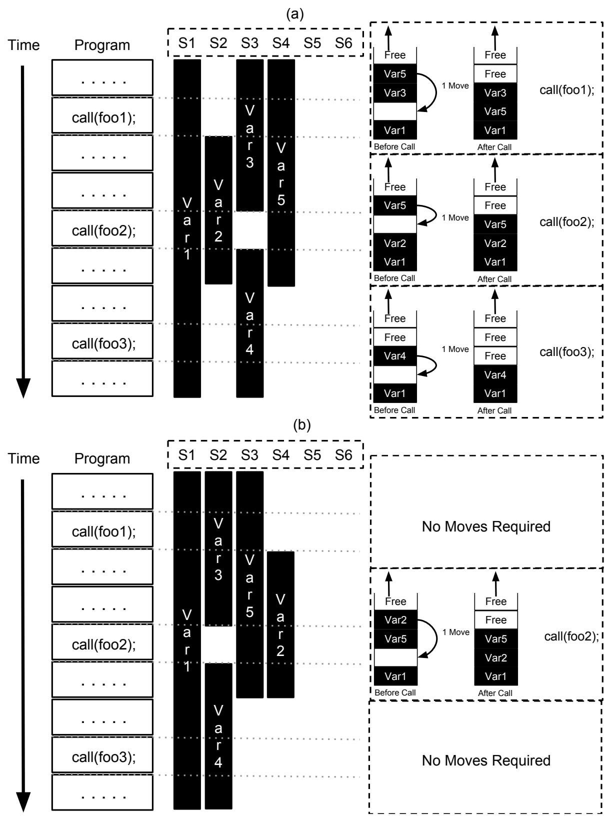


Figure 6: The impact of on-chip memory slot layout on the number of data movements for a compressible stack. Every column S_i represents a slot. The vertical blocks in the column represent the liveness of the variable assigned to the slot. Bold arrows represent direction of data movement for entering or leaving a sub-procedure.

to at most B_k slots, we have:

$$W_{ij} = \sum_{k=0}^{N-1} C_{ijk}$$

while

$C_{ijk} = 1$ if ($L_{ik} == 1$ && $j \geq B_k$), otherwise $C_{ijk} = 0$.

Proof. Since a movement will be invoked if and only if there is an available stack slot placed beyond the top of the after-compression stack, we can determine if a placement of the SS_i set will incur a movement by checking its location (address) in the stack.

If and only if SS_i is live during the lifetime of the k-th sub-procedure (liveness indicated as L_{ik} in Table 1), and placed at $j-$th slot (j starts from 0) counting from the current procedure's stack bottom is greater than or equal to B_k, $j \geq B_k$, the number of data movements invoked by placing variable set SS_i at $SLOT_j$ for k-th sub-procedure C_{ijk} is 1; otherwise, C_{ijk} is 0. Therefore, we get all the movement contributed by placing SS_i into $SLOT_j$ by summing up C_{ijk} for $k = 0...N-1$. The theorem is thus proved. □

We then transform the problem into a bipartite matching problem. A bipartite graph is a graph whose nodes can be decomposed into two disjoint sets such that no two nodes within the same set are adjacent as shown in Fig. 7. A perfect matching in a bipartite graph is a set of pairwise non-adjacent edges that covers every node in the graph. A maximum weighted bipartite matching is one where the sum of the weights of the edges in the matching is maximal as shown in Fig. 7.

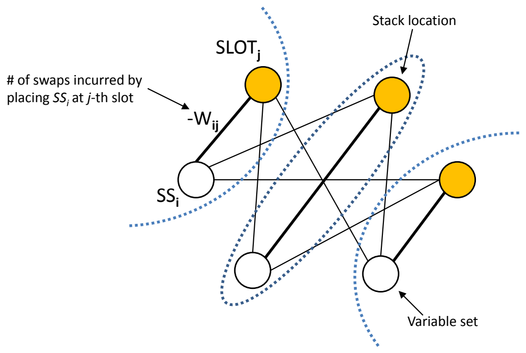


Figure 7: Bipartite matching model for minimal-mov-assignment problem. SS_i represents the i-th set of variables that can be mapped to one on-chip memory slot, $SLOT_i$ represents the i-th physical on-chip memory slot in the procedure of interest.

We let one of the two disjoint sets of nodes correspond to the sets of variables – SS_i for $i = 0...M-1$. The other set of nodes correspond to the memory slots $SLOT_0...SLOT_{M-1}$ from the bottom of the stack. One edge connects between a variable set and a stack location (address).

We set the weight of the edge between a set SS_i and a j-th stack slot to negative W_{ij} as defined in Theorem 1. This value is the total number of movements invoked by placing SS_i at $SLOT_j$. Therefore, a maximum weighted matching will indicate a minimum number of movements as indicated in Fig. 7.

We solve the maximum weighted bipartite matching problem using the modified Kuhn-Munkres algorithm [17], with O(M^3) time complexity, with M being the total number of variable sets. Once a matching is found, we can infer where every variable set can be placed.

Our model works for the case where we need not compress the stack to the minimal size possible. For example, if the stack can be compressed to allow for four free slots, but we only need three, then it is sufficient to compress the stack to allow three slots. Therefore, we avoid extra overhead from pointless stack compression movements. Let the parameter B_k be the size the stack needs to be compressed to, such that B_k is greater than the minimal possible compressed stack size. Then the optimality and complexity results still hold.

We demonstrate the effectiveness of the data movement optimization Figure 5. The bars that correspond to the case of "unoptimized data movement minimization" are the case without data movement optimization. Note that without data movement optimization, the performance might be even worse than not doing stack compression again. Therefore, minimizing data movement is extremely critical for minimal space optimization to work well (which is critical

```
original: the kernel at original occupancy
canTune: true iff benchmark has enough iterations

if MAXLIVE >= 32
    direction = increasing
else
    direction = decreasing
if canTune
    kernels.add(original)
    if direction = increasing
        for each occupancy from conservative to max
            kernels.add(occupancy)
    else
        kernels.add(conservative)
else
    kernels.add(get_static_selection())
```

Figure 8: Occupancy update algorithm

for occupancy tuning).

3.3 Compile-Time Occupancy Tuning

During compile-time tuning, we perform test and update steps, as shown in Figure 3 (a). The *occupancy testing* component checks the generated kernel binary and determines if the occupancy needs to be further increased and decreased. If the occupancy needs to be further updated, shown as the back loop in Figure 3 (a), then the *realizing occupancy* component will be invoked again to generate a new kernel binary. If the occupancy does not need to be further updated, then the previously checked versions will be saved into the candidate set of kernel binaries for runtime adaptation. Our compiler determines and generates a set of kernel versions in which the optimal occupancy version is most likely to appear. We are able to narrow down the set of candidate kernel versions to within six, and in most cases less than three, making it easy for the runtime component to choose the right kernel version.

We set the initial occupancy such that all variables fit into the minimal number of registers, or the maximum number of registers per thread (by hardware) is reached. We call it the *original* version since this is the version for which we decide the tuning direction (increasing or decreasing). The original version is not necessarily the best version, but it is a safe version. Once the initial occupancy is set, next we decide the direction of increasing or decreasing occupancy in the iterative selection process. Once the direction is determined, we keep increasing/decreasing the occupancy levels and testing the generated code. We stop increasing/decreasing at a certain point if a termination condition has been met according to our performance model below. We show the detailed occupancy *test and update* algorithm in Figure 8. In Figure 8, we define another version called the *conservative version*, which is the version where all variables fit into on-chip memory. The *conservative* version usually provides better occupancy than the *original* version since we fit more threads using all on-chip memory (register, smem, and cache) than

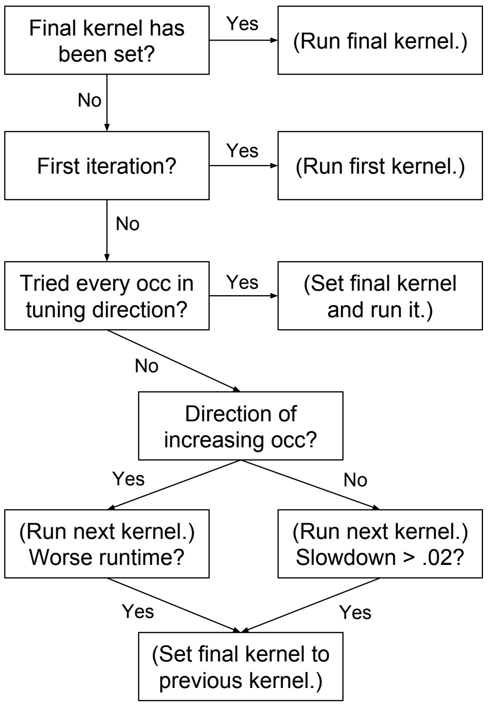

Final kernel has been set?
Yes
(Run final kernel.)
No
First iteration?
Yes
(Run first kernel.)
No
Tried every occ in tuning direction?
Yes
(Set final kernel and run it.)
No
Direction of increasing occ?
Yes
No
(Run next kernel.) Worse runtime?
(Run next kernel.) Slowdown > .02?
Yes
Yes
(Set final kernel to previous kernel.)

Figure 9: Dynamic occupancy selection algorithm

only using registers.

To determine the direction of tuning, we rely on a metric determinable at compile-time, max-live, which indicates the total number of registers and memory slots necessary for the program. The analytical model [12] uses off-line profiled information, including memory throughput and dynamic instruction count, to estimate the performance of a GPU program. Our approach does not require off-line profiling and yet is lightweight in determining the amount of memory/computation parallelism.

In cases where the kernel function cannot be tuned (for example, if it only has a single iteration), the selection process will use the static selection algorithm described in [11] to generate the final kernel function.

Max-live.

We use a metric called *max-live*, which is equal to the number of registers necessary to hold all simultaneously live variables. When this value is low, the on-chip memory resource demand per-thread is low, and thus a high occupancy is reached (potentially hitting the maximum number of active threads that hardware can handle). For this type of application, we can tune only by decreasing the occupancy from the initial *original* occupancy. We set to *max-live* threshold to 32 in our experiments, which is the number registers needed to achieve the hardware maximum occupancy level for Kepler architecture. If a program's max-live is less than the number of registers which allows the hardware maximum occupancy, then occupancy cannot be increased through allocation.

Finally, we provide a fail-safe option in case the direction predicted at compile-time does not work at runtime (although this has rarely happened in our evaluation). We generate kernel codes in the increasing occupancy direction

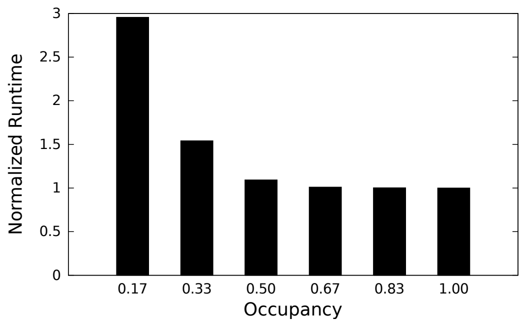


Figure 10: Normalized srad performance on Tesla C2075.

(the conservative code and the next occupancy up) if the direction is predicted to be decreasing, and also generate code that enables decreasing occupancy (if the direction is predicted to be increasing). This way, in the rare event that our initial direction is wrong, we can try the other direction as a fail-safe. Note that we do not need to generate multiple versions of code to correspond to decreased occupancy levels, since we can tune occupancy down by dynamically increasing shared memory usage per thread as shown in Figure 8.

3.4 Runtime Occupancy Adaptation

Given the candidate list of kernel binaries generated by the compiler, the ORION runtime monitors kernel performance and dynamically selects the best kernel versions. In a loop that calls the GPU kernel of interest, we run the original kernel in the first iteration. From the second iteration we start running the next version in the list and updating occupancy level in the predicted tuning direction by the ORION compiler, until we see performance degradation. If an iteration has no performance degradation, then in the next iteration we simply run the next occupancy in the current direction. This algorithm is shown in Figure 9.

In practice, we find that the tuner usually only needs three iterations to adapt to the best occupancy. As long as the kernel runs for many iterations, the overhead of tuning is low. The algorithm can be augmented to handle misprediction of the tuning direction, by switching direction if the original occupancy is selected as the final kernel. We find that this is not typically necessary.

Most GPU programs contain a loop around the GPU kernel of interest. If there is no loop but there are enough threads, then we perform *kernel splitting* [30]. We split one kernel invocation into multiple invocations, such that every invocation of the split kernel launches a subset of the threads and the total threads across invocations is the same as the original kernel invocation.

It is worth noticing that in certain cases, a decreased occupancy can yield the same performance while significantly reducing resource usage. We show the normalized running time of the *srad* program on NVIDIA Tesla C2075 in Figure 10. The performance in Figure 10 is normalized to the performance when there are maximal active threads on each SM. In *srad*, even reducing the occupancy by half yields nearly the same performance, and so reducing occupancy is suggested for this program.

4. EVALUATION

Our framework consists of two components. One is the ORION compiler and the other is the runtime adaptation component. The ORION compiler's front end, middle end, and back end take on different responsibilities. The front end is responsible for taking a GPU binary file as input, converting it into assembly code, and analyzing the assembly to extract a high level intermediate representation (IR). The middle end utilizes the IR generated in the front end and transforms the IR. The IR includes the control flow graph and the call graph. The middle end obtains a single static assignment (SSA) form of the code, extracts live ranges, performs resource allocation, updates the control flow graph, and writes back to the assembly code. The static multi-kernel selection and generation is in the middle end of the ORION compiler. The back end converts the transformed assembly code back to binary code.

Candidate kernels are generated at multiple different occupancy levels. For evaluation, we let the ORION compiler generate code at all occupancy levels, allowing for identification of the best and worst cases. We compare these with the ORION selected occupancy and the default code generated by nvcc.

The runtime adaptation component performs the feedback-based tuning algorithm as described in Section 3.4. The runtime component works with the compiler component, as it can only choose the kernel binaries that are generated from the multi-kernel binary generation stage in the ORION compiler.

We perform transformation directly on the binary code (SASS) rather than PTX so that the transformation effects can immediately be reflected in the final binary code. Using PTX requires a further compilation using *ptxas* from PTX to binary, and the changes made in PTX may be lost since *ptxas* may perform another level of register allocation.

We build the ORION framework upon the following tools. The front end of ORION compiler is built upon the parser generator tools *flex* and *bison*. To encode and decode binary, we use the binary instruction set architecture (ISA) of NVIDIA GPUs documented in the open-source project *asfermi* [14] for cuda computing capability 2.0, and we also extended the ISA support by reverse engineering the ISA using the same approaches in *asfermi* [14].

While the ORION framework currently only supports a subset NVIDIA GPU architectures, it can easily be extended to support additional GPU architectures if we add a new front-end and back-end to decode and encode the binary, since the middle-end and the transformation algorithms remain the same.

Platform.

We perform experiments on two different machine platforms. One is equipped with an NVIDIA GTX680 GPU. It has 8 streaming multi-processors (SMs) with 192 cores, for a total of 1536 cores. Every SM has 65536 registers, and 64KB of combined shared memory and L1 cache. The maximal number of active thread warps on one SM is 64, and the maximal number of active threads per SM is 2048.

The second machine platform is configured with an NVIDIA Tesla C2075 GPU. It has 14 streaming multi-processors (SM) with 32 cores, for a total of 448 CUDA cores. Each SM has 32768 registers and 64KB of combined shared memory and L1 cache. The maximal number of active thread warps on

Table 2: Detailed benchmark information. *Reg* is the number of registers needed to avoid spilling. *Func* is the number of static function calls. *Smem* indicates whether there is user-allocated shared memory.

Benchmark	Domain	Reg	Func	Smem
cfd [6]	Fluid dynam.	63	36	No
dxtc [20]	Image proc.	49	11	Yes
FDTD3d [20]	Numer. analysis	48	0	Yes
hotspot [6]	Temp. modeling	37	6	Yes
imageDenoising [20]	Image proc.	63	2	Yes
particles [20]	Simulation	52	0	No
recursiveGaussian [20]	Numer. analysis	42	21	No
backprop [6]	Machine learning	21	0	No
bfs [6]	Graph traversal	16	0	No
gaussian [6]	Numer. analysis	11	2	No
srad [6]	Imaging app	20	7	Yes
streamcluster [6]	Data mining	18	0	No

every SM is 48 and the maximal number of active threads is 1536.

For both platforms, each register is 4 bytes. If there are wide variables (i.e., 64-bit, 96-bit, or 128-bit variables), then they must be placed in aligned, consecutive 32-bit registers. We refer to the first machine configuration as *GTX680*, and the second one as *Tesla C2075*.

Benchmarks.

We evaluate the effectiveness of the ORION framework on benchmarks shown in Table 2. These benchmarks are chosen to cover GPU programs from various domains with different characteristics: high register pressure v.s. low register pressure, with and without function calls, and with and without user-defined shared memory. Note that the number of function calls is counted after function inlining. In GPU program compilation, function calls are inlined as much as possible since there is a local stack for every thread, which needs to be minimized for a large number of threads running at the same time. However, as shown in Table 2, there is still a non-trivial number of function calls that are not practical to be inlined. This confirms the necessity of efficient inter-procedure register allocation. These benchmarks come from the Rodinia [6] benchmark suite and the CUDA Computing SDK [20].

Metrics.

We evaluate the ORION occupancy tuning framework using three different metrics. The first metric is performance - in particular, the effectiveness of the ORION compiler at generating good code, as well as the effectiveness of the tuner to adapt to the best occupancy as compared with exhaustive search. The second metric is the resource usage efficiency of *Orion*: whether it uses minimal resources (registers) to achieve the best performance, where the best performance is defined as the best running time for all different occupancy levels. The third metric is energy usage. We discovered that there is also energy saving when there is resource usage saving. The reason is that when occupancy decreases (while maintaining the same performance), the power usage of the register file (and/or cache) is also reduced, thus effecting energy saving.

4.1 Performance

We show performance evaluation results first, using the seven benchmarks which the ORION compiler determined would benefit from increased occupancy. For comparison purpose, we let the ORION compiler generate code for every occupancy level. Figure 11 shows the worst performance across different occupancy levels (longest running time – the Orion−Min bar), the best performance across all occupancy levels (shortest running time – the Orion−Max bar), and the performance of the code generated by nvcc. For all of these benchmarks, the difference between the best performance and worst performance across occupancy levels is significant - in some cases, more than 75% - demonstrating the importance of occupancy selection. It is not clear how nvcc chooses the occupancy since the nvcc compiler backend is not open source. We can observe from Figure 11 that the version selected by nvcc typically is not the worst case scenario among all occupancy levels, but it certainly has missed performance optimization opportunities for most cases. The nvcc selected version is rarely the best occupancy, as illustrated in Figure 11, except in the case of recursiveGaussian.

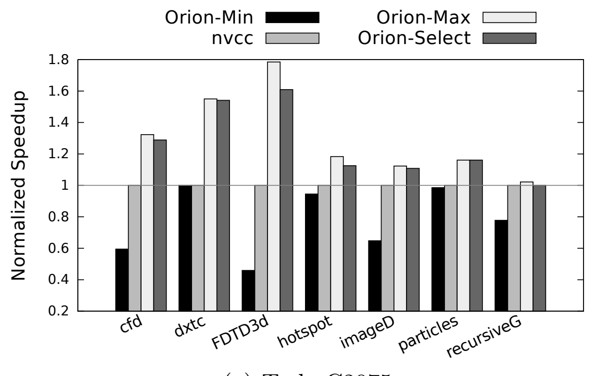
We also show the occupancy selection result of our ORION framework. Figure 11 shows ORION performance – the Orion-Select bar. This bar includes the overhead of dynamic tuning. We can see that Orion-Select is close to the best performance obtained by exhaustively searching all occupancy levels. The performance of Orion-Select come from two aspects – static selection that narrows down the possible kernel versions and dynamic selection that chooses the best kernel version at runtime. The static selection ensures that there are no more than five different kernel versions selected at compile-time. During runtime selection, ORION required less than three iterations on average to to tune each benchmark. We find that in most benchmarks, either there are sufficiently many iterations to perform dynamic tuning, or there are sufficiently many threads that we can split the kernel call into multiple, smaller invocations in order to create additional iterations. The particles benchmark, however, is an exception to this, and so ORION chooses the compiler-picked, statically-tuned kernel version as described in Section 3.3, which still provides significant speedup over the default code generated by nvcc.

On average, ORION achieves 26.17% speedup on C2075 and 24.94% speedup on GTX680.

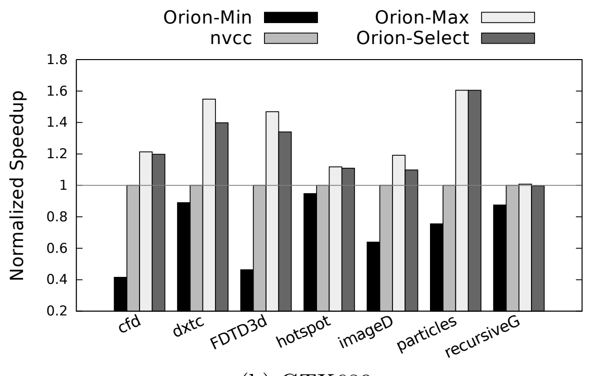
Another factor which affects the performance is cache configuration. On both Tesla C2075 and GTX680 devices, the shared memory and the L1 cache can be reconfigured to use a different size ratio. In Table 3, we show the comparative speedup of using a small cache configuration (16KB L1 cache and 48KB shared memory) versus using a large cache configuration (48KB L1 cache and 16KB shared memory) for the Orion-Select occupancy level. Note that the results presented in Figure 11 are all for small cache configuration.

With different cache configurations, we distribute variables differently. For the smaller shared memory configuration, we fit fewer variables into the shared memory but spill more variables into local memory, which can reside in on-chip memory using L1 cache. To determine how many shared memory slots to use for a given occupancy, we use the formula described in Section 2).

From Table 3, we can see that performance is often similar for both configurations. However, cases such as FDTD3d on Tesla C2075 show more degradation in the large cache performance. When we spill the variables to local memory, due to the non-deterministic feature of thread interleaving, it is



(a) Tesla C2075



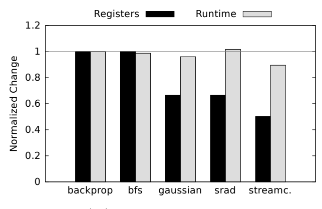
(b) GTX680

Figure 11: Normalized speedup over nvcc version. We show the best/worst performance at all occupancy levels. Orion-Max is the best performance. Orion-Min is the worst. Orion-Select is the performance with tuning. nvcc is the performance by nvcc generated code.

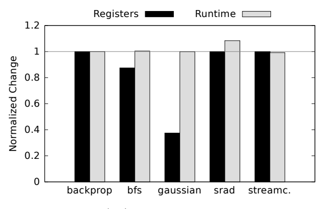
Table 3: Speedup with Small Cache (SC) vs Large Cache (LC) at Orion's selected occupancy. In some cases, hardware constraints prevent the LC case from running.

Benchmark	**C2075**		**GTX680**	
	SC	**LC**	**SC**	**LC**
cfd	1.3230	1.2939	1.1656	1.1588
dxtc	1.5409	-	1.3980	-
FDTD3d	1.5674	1.2811	1.2605	1.2552
hotspot	1.1834	1.1780	1.1175	-
imageD	1.1229	1.1225	1.0834	1.0850
particles	1.1608	1.1492	1.6045	1.6774
recursiveG	1.0000	1.0002	1.0008	1.0001

difficult to guarantee that the L1 cache will behave as expected. For example, cache thrashing may happen when thread execution is interleaved at different times. Overall, it is safer to use shared memory to explicitly store live variables than to use hardware cache. Note that for programs that use a significant amount of user-defined shared memory per thread block, the large cache configuration cannot be used due to the occupancy requirement, and therefore three entries in Table 3 are empty.



(a) Tesla C2075



(b) GTX680

Figure 12: Results of downward occupancy tuning.

4.2 Resource & Energy Usage

For five of our benchmarks, the Orion compiler predicts that dynamic tuning should lower occupancy, since these benchmarks have small register pressure (small max−live as described in Section 3.3). They have already reached the maximum occupancy supported by hardware, therefore the

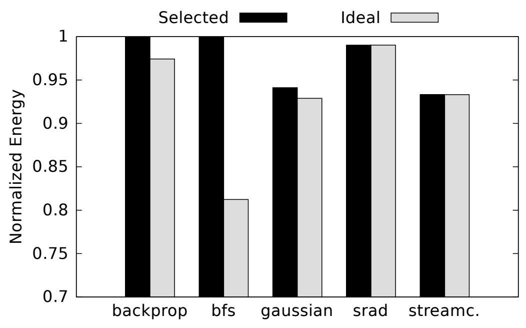


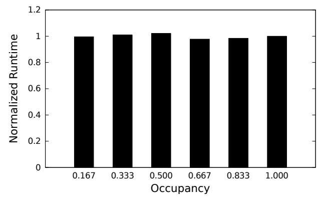
Figure 13: Energy usage of selected kernel.

only potential tuning direction is downwards. The register file utilization and runtime for these five benchmarks are shown in Figure 12. These values are normalized to those of the program generated by *nvcc*. The normalized occupancy is the same as the register file utilization, and so we exclude it from the figure. Overall, we decrease occupancy and register usage by 19.17% on average, with little loss in performance.

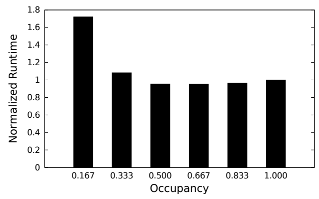
We are unable to tune the backprop benchmark, due to the simplicity of its kernel function, which contains less than 100 binary instructions and has no loops or subroutines. Attempting to tune this benchmark's kernel function would lead to significant overhead, as the runtime of the kernel function is similar in scale to the overhead of launching an empty kernel function, especially if kernel splitting is employed. In such cases, it makes more sense to simply default to the original version of the kernel. Orion is also unable to get significant reduction of occupancy for bfs, despite lower occupancy being advantageous in this benchmark, because bfs does different amounts of work in each iteration, making it difficult to compare consecutive invocations. Even so, we do get some reduction for bfs on the GTX 680 architecture. In future work, we may be able to improve tuning for such cases by calculating the amount of work at each iteration and applying a multiplicative factor to the runtime.

Besides saving registers, lowering concurrency has the additional benefit of reducing power consumption due to the reduced utilization of the register file. We measured this using NVIDIA's CUPTI API. GTX680 does not allow for power measurement in this manner, and so we show the energy savings only for Tesla C2075 in Fig. 13. We include both the energy saving at the selected occupancy level, and the ideal energy saving determined via exhaustive search.

It is demonstrated in Figure 12 that we can sometimes attain speedup when lowering occupancy, due to the decreased resource contention that results from fewer active threads. We find that this occurs more easily on Tesla C2075, where the L1 cache is used for both global memory and local memory, than on GTX680 where the L1 cache is used exclusively for thread-private local memory. Overall, we get an average speedup of 3.24% for these five benchmarks.

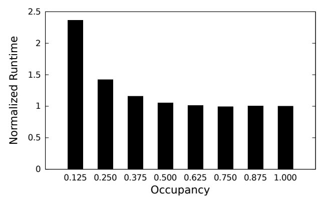


(a) gaussian

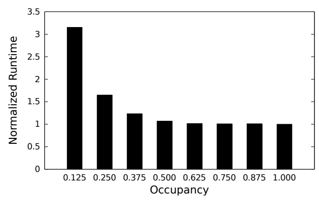


(b) streamcluster

Figure 14: Effects of occupancy on performance for C2075.



(a) backprop



(b) bfs

Figure 15: Effects of occupancy on performance for GTX680.

Finally, in Figure 14 and Figure 15, we show in detail how performance varies as occupancy changes. We show two benchmarks for each architecture since the performance variation pattern is similar. In Figure 14(a), we show the benchmarks *gaussian*, which is insensitive to occupancy tuning. Its performance changes very little across occupancy levels, demonstrating significant potential for resource saving and energy saving. Figure 14(b) (*streamcluster*) and Figure 15(a) (*backprop*) show skewed bell curves, with performance being best at around 75% occupancy but not changing significantly above 50%. In 15(b) (*bfs*), performance is best at highest occupancy, but again changes only a little when above 50%.

In all four of these cases, performance as a function of occupancy plateaus, demonstrating a range of occupancy values between which performance is very similar. This observation confirms the advantage of dynamic tuning: even when the performance is the best, we can still keep tuning and obtain potential saving for resources and energy usage. Further, in such programs, we can use this information for additional optimization. For example, loop unrolling is a common technique which reduces branch penalties, but may increase register pressure and therefore lower occupancy. By finding this range of similar occupancies, however, we can determine the amount of leeway available with which to perform such optimizations without experiencing slowdown.

5. RELATED WORK

In this paper, we propose an occupancy tuning framework for GPU programs. There have been GPU performance tuning frameworks that focus on different factors. The PORPLE framework [7] determines which type of memory to use according to different data access patterns. Liu and colleagues [15] utilize input-sensitivity to select the best program compilation and execution parameter for every input. Yang and others [29] have developed a GPU compiler that focus on static-time memory optimization and parallelism management. There are also domain-specific program tuning studies by compiler designers. Anand et al. [25] explores the dimension of data representation methods for sparse matrix code. The Halide [23] framework tunes the locality and parallelism parameters for image processing pipeline. However, none of the tuning frameworks exploited the impact of occupancy tuning on general-purpose GPU programs.

Since GPU program occupancy tuning is correlated with resource allocation, we also compare our work with previous resource allocation studies. Register allocation for CPU programs has been extensively studied [4] [5] [22] [1] [10] [21] [2] [8] [27] [16]. For GPU register allocation, Sampaio and others [24] identified the opportunities in saving registers for control flow statements in GPU programs. Our prior work [11] places the spilled register variables by nvcc into available shared memory, however it does not perform register allocation and it does not handle wide variable registers. Further it is purely static compile-time approach and does not adaptively tune the occupancy (downwards or upwards) based on runtime program behavior. None of the above work thoroughly explores the relations between register (resource) allocation and occupancy tuning.

6. CONCLUSION

We propose the first framework for GPU occupancy tuning, ORION. The ORION framework performs two-level occupancy selection. The ORION compiler performs the first-level occupancy selection and generates a set of kernel binaries for dynamic tuning. The ORION runtime performs a second-level of occupancy selection which adapts to dynamic program behavior. It is able to find the best occupancy or an occupancy close to the best one within three iterations on average. The ORION compiler and runtime not only improve performance – achieving up to 1.61 times speedup – but also resource & energy efficiency, with up to 62.5% memory resource saving, and 6.7% energy reduction over the highly optimized code generated by the *nvcc* compiler.

Acknowledgement

We thank Thomas Gross for his insightful comments on the draft of the paper. We owe a debt of gratitude to the anonymous reviewers for their invaluable comments. This work is supported by NSF Grant NSF-CCF-1421505, Google Faculty Research Award, Rutgers University Research Council Grant, and the DOE ASCR CESAR Project. 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.

7. REFERENCES

[1] Appel, A. W., and George, L. Optimal spilling for cisc machines with few registers. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (New York, NY, USA, 2001), PLDI '01, ACM, pp. 243–253.

[2] Baev, I. D. Techniques for region-based register allocation. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2009), CGO '09, IEEE Computer Society, pp. 147–156.

[3] Briggs, P., Cooper, K. D., and Torczon, L. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst. 16*, 3 (May 1994), 428–455.

[4] Chaitin, G. J. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction* (New York, NY, USA, 1982), SIGPLAN '82, ACM, pp. 98–105.

[5] Chaitin, G. J., Auslander, M. A., Chandra, A. K., Cocke, J., Hopkins, M. E., and Markstein, P. W. Register allocation via coloring. In *Computer Languages* (1981), vol. 6, pp. 47–57.

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

[7] Chen, G., Wu, B., Li, D., and Shen, X. Porple: An extensible optimizer for portable data placement on gpu. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2014), MICRO-47, IEEE Computer Society, pp. 88–100.

[8] Choi, Y., and Han, H. Optimal register reassignment for register stack overflow minimization. *ACM Trans. Archit. Code Optim. 3*, 1 (Mar. 2006), 90–114.

[9] Hack, S., and Goos, G. Copy coalescing by graph recoloring. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2008), PLDI '08, ACM, pp. 227–237.

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

[11] Hayes, A. B., and Zhang, E. Z. Unified on-chip memory allocation for simt architecture. In *Proceedings of the 24th ACM International Conference on Supercomputing* (New York, NY, USA, 2014), ICS '14, ACM.

[12] Hong, S., and Kim, H. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2009), ISCA '09, ACM, pp. 152–163.

[13] Hong, S., and Kim, H. An integrated gpu power and performance model. In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2010), ISCA '10, ACM, pp. 280–289.

[14] Hou, Y., Lai, J., and Mikushin, D. asfermi: An assembler for the nvidia fermi instruction set.

[15] Liu, Y., Zhang, E., and Shen, X. A cross-input adaptive framework for gpu program optimizations. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* (May 2009), pp. 1–10.

[16] Lueh, G.-Y., Gross, T., and Adl-Tabatabai, A.-R. Fusion-based register allocation. *ACM Trans. Program. Lang. Syst. 22*, 3 (May 2000), 431–470.

[17] Munkres, J. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial & Applied Mathematics 5*, 1 (1957), 32–38.

[18] NVIDIA. Cuda c programming guide.

[19] NVIDIA. CUDA occupancy calculator.

[20] NVIDIA. GPU computing sdk.

[21] Palsberg, J. Register allocation via coloring of chordal graphs. In *Proceedings of the thirteenth Australasian symposium on Theory of computing - Volume 65* (Darlinghurst, Australia, Australia, 2007), CATS '07, Australian Computer Society, Inc., pp. 3–3.

[22] Poletto, M., and Sarkar, V. Linear scan register allocation. *ACM Trans. Program. Lang. Syst. 21*, 5 (Sept. 1999), 895–913.

[23] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2013), PLDI '13, ACM, pp. 519–530.

[24] Sampaio, D. N., Gedeon, E., Pereira, F. M. Q. a., and Collange, S. Spill code placement for simd machines. In *Proceedings of the 16th Brazilian conference on Programming Languages* (Berlin, Heidelberg, 2012), SBLP'12, Springer-Verlag, pp. 12–26.

[25] Venkat, A., Hall, M., and Strout, M. Loop and data transformations for sparse matrix code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2015), PLDI 2015, ACM, pp. 521–532.

[26] Volkov, V. Better performance at lower occupancy. *Proceedings of the GPU Technology Conference, GTC 10* (2010), 16.

[27] Wang, J., Krall, A., Ertl, M. A., and Eisenbeis, C. Software pipelining with register allocation and spilling. In *Proceedings of the 27th annual international symposium on Microarchitecture* (New York, NY, USA, 1994), MICRO 27, ACM, pp. 95–99.

[28] West, D. B. *Introduction to Graph Theory*, 2 ed. Prentice Hall, September 2000.

[29] Yang, Y., Xiang, P., Kong, J., and Zhou, H. A gpgpu compiler for memory optimization and

parallelism management. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2010), PLDI '10, ACM, pp. 86–97.

[30] Zhang, E. Z., Jiang, Y., Guo, Z., Tian, K., and Shen, X. On-the-fly elimination of dynamic irregularities for gpu computing. In *ACM SIGARCH Computer Architecture News* (2011), vol. 39, ACM, pp. 369–380.