Synergistic Data Placement for GPU On-chip Memory

Abstract

General purpose GPU architecture has various types of on-chip memory: registers, software-managed cache, and hardware-managed cache. These on-chip memory resources are powerful yet difficult to maneuver. Each type of on-chip memory has its advantages/disadvantages, making it suitable for different types of data. Further, the on-chip memory contention at different levels affects hardware concurrency that can be achieved on GPU. Unlike CPU architecture, on which on-chip memory allocation is performed under a fixed resource bound, GPU on-chip memory resource bound is a variable because of its relationship with the adjustable hardware concurrency.

In this paper, we look at the data values that are analyzable at compile-time for placement in registers, software-managed cache and hardware-managed cache. We propose an unified data placement strategy applicable to every type of on-chip memory, and yet flexible enough to maximize synergy among different types of on-chip memory.

1. Introduction

Emerged as an appealing many-core accelerator, general purpose GPU not only allows maximum parallelism but also concurrency. The number of threads that run simultaneously is typically 10 to 32 times of the number of GPU cores. The concurrent execution enables dozens of threads to time-share every single GPU core and hide memory latency.

The maximum hardware concurrency, on one hand, leads to potential performance enhancement, on the other hand, increases program tuning complexities, especially for on-chip memory allocation problem.

There are three types of on-chip memory: registers, software-managed cache and hardware-managed cache. The sharing of registers and software-cache is directly related to the amount of concurrency. The sharing of hardware-cache does not affect the amount of concurrency but can potentially affect the efficiency of concurrent execution.

Unlike CPU thread switches, GPU thread switches incur nearly “zero” overhead. It is achieved by a large register file holding the contexts of all simultaneously active running threads. No register saving/restoring through off-chip memory is necessary for switches among these active threads.

For this reason, only a limited number of threads can run simultaneously on GPU at one time, with respect to per-thread register usage. The number of simultaneously active threads is also bounded by per-thread software-cache usage. Every thread is assigned the same amount of registers and software-cache space.

The amount of concurrency for a GPU program is described as occupancy in NVIDIA terminology. It is the ratio between a program’s maximum number of simultaneously active threads and the hardware’s maximum number of schedulable threads. Large on-chip memory per-thread leads to small occupancy and vice versa.

A GPU compiler needs to designate per-thread register and software-cache usage and thus tune program occupancy, while a CPU compiler does not have to. In this paper, we aim to address the on-chip memory data placement problem.

We look at the set of values/variables that are traditionally placed into register and place them into not only registers but also software-managed and hardware-managed cache.

The GPU on-chip memory data placement problem is challenging for the following reasons. First, the problems of register, software cache, and even hardware-cache allo-
cation are not independent. They interact with each other by tuning the occupancy constraint. We show an example of the impact on occupancy by different register/software-cache/hardware-cache in Fig. 1.

Secondly, GPU kernels may involve non-trivial amount of small procedure calls. Most existing CPU register allocation algorithms work well for single-procedure allocation but not necessarily for multi-procedure allocation.

Thirdly, every type of on-chip memory has its pros and cons. For register and software-cache, we need to explicitly manage data placement/eviction, while for hardware-cache, we do not need to. Hardware-cache relies on least recently used (LRU) data policy but not others. Finally, the addresses in software-cache can be calculated at runtime using arithmetic expressions while register address has to be constant determined at compile-time.

Previous work by Sampaio [28] and Hayes [16] use shared memory (software cache) to reduce register pressure. The former [28] concentrates on register pressure caused by control flow. The latter [16] proposes moving-stack approach to use software-managed cache to store spilled register variables that are traditionally placed in off-chip memory. The moving stack approach is a best effort approach and does not provide any insight on the optimality of allocation.

The SSA-LLP approach in [5] proposes inter-procedure allocation with the use of pre-allocated global registers called launch and landing pads (LLP) for ASIP design. If register address can be dynamically calculated at runtime, in theory, the SSA-LLP approach guarantees optimal inter-procedure register usage under SSA form. However, current GPU architecture does not support dynamic register addressing. Besides, allocation under pure SSA form needs to be converted to remove the \( \phi \) instructions that do not exist in real architecture. Eliminating \( \phi \) function requires insertion of large number of MOV instructions in lieu of parallel copies [14], making it impractical.

In this paper, we aim to systematically explore the allocation policies for different types of on-chip memory. Our contributions are summarized as follows:

- We propose an unified allocation framework applicable to every type of on-chip memory and yet flexible enough to adapt to their distinct features.
- We show that an optimal polynomial-time on-chip memory allocation algorithm exists in terms of both the amount of space and data movement.
- Our proposed allocator is transparent to the programming language used to write GPU programs. It takes any GPU binary, extracts high level intermediate representation, performs resource allocation, and rewrites back to binary. It is evaluated on real architecture instead of simulator.

We organize the rest of our paper as follows. Section 2 defines the problem and the notations we use in the paper. Section 3 presents the overall allocation framework and optimality guarantee. Section 4 describes the assignment strategy. Section 5 and Section 6 respectively present evaluation and related work.

2. Problem Formulation and Notations

GPU on-chip memory includes registers, software cache and hardware cache. The software cache is named as shared memory with NVIDIA terminology and local memory with OpenCL terminology. The hardware cache is mainly L1 cache since L2 cache has similar latency to off-chip memory. Software cache has similar latency compared to L1 cache.

Depending on the GPU architecture, the size of software cache and hardware cache can be dynamically configured in two or three modes: smaller software cache (16KB) larger hardware cache (48KB), larger software (48KB) smaller hardware cache (16KB), or equal size of software cache (32KB) and hardware cache (32KB). In every configuration, the total size of software and hardware cache size remains unchanged.

GPU execution model is single instruction multiple thread (SIMT) model. Every threads gets an equivalent share of registers and software cache. Hardware cache is shared dynamically.

A GPU program consists of both CPU code and GPU code. A function that actually runs on GPU is called a GPU kernel. We perform on-chip memory allocation only for GPU kernel functions.

Executing a program requires to load a set of values into registers. Compiler analysis determines which variables and when they can be safely placed in register. This set of values can be represented in Static Single Assignment (SSA) form, in which every variable is defined once and only once. Our work assumes a SSA form is already present and it serves as our input. Our task is to map the set of SSA variables into a set of physical registers and software/hardware cache location. LOAD, STORE and MOV instructions are inserted for spilling (into off-chip memory) or parallel copy.

Recent GPU architecture divides the hardware cache into two parts: one part that is only used for thread-private local variables (L1 cache), and the other part that is used for global data variables (texture cache or read-only cache) [25]. We can place a variable into local memory and let it be mapped to L1 cache by controlling the bits in the binary instruction (bypass cache or not).

In this paper, we define the commensurate amount of on-chip memory space to a 32-bit physical register as one physical on-chip (poc) memory slot. A poc slot can be a register, a shared memory slot or a local memory slot. A object larger than 4 bytes is broken down and represented as multiple poc slots.

In the variable placement problem, the key is to minimize the physical on-chip memory usage per-thread, the number of poc slots, with the goal of maximum concurrent execution performance.
Problem Definition  Let a GPU program have \( m \) procedures in its call graph: \( p_i, 0 \leq i \leq m - 1 \). Assume there are \( n \) physical on-chip memory slots: \( poc_j \), \( 0 \leq j \leq n - 1 \). At every procedure \( p_i \), we need to find \( k_i \) physical on-chip memory slots \( poc_j \), \( 0 \leq j \leq k_i - 1 \), such that every SSA variable \( v \) maps to one physical on-chip memory slot \( v \rightarrow poc_j \).

More specifically, we aim to address two sub-problems:

- How to determine the minimal amount of on-chip memory to store all variables throughout the whole GPU kernel?
- How to assign every individual variable into a physical on-chip memory slot \( poc_j \) and determine the type of \( poc_j \) to be register, software or hardware cache?

Here, the constraint is that no two variables \( v_i \) and \( v_j \) that are simultaneously live at any point should be mapped to the same physical on-chip memory slot \( poc_k \).

Optimality  In general, the memory allocation problem can be modeled as a graph coloring problem [6][18]. Let \( G = (V, E) \) be the interference graph with the set of vertices \( V \) and the set of undirected edges \( E \subset V \times V \). A vertex \( v \in V \) represents a variable. An edge \( e = (v, u) \in E \) denotes the two variables \( v \) and \( u \) are at least simultaneously live at one point in the execution.

Finding minimal number of \( poc \) slots is equivalent to assigning the minimal number of colors to vertices such that no two incident vertices are assigned to the same color. The minimal number of colors is called the chromatic number.

In single procedure register allocation for CPU program, the chromatic number is the max clique size for SSA form [15]. The max clique size for a program \( P \) in SSA form is equivalent to the maximum number of simultaneously live variables, which we denote as \( MaxSimLive(P) \). Obviously, the \( MaxSimLive \) is the lower bound of the optimal number of physical registers needed since there are \( MaxSimLive \) variables live at a certain point of the program. Similarly, the \( MaxSimLive \) is the lower bound of optimal \( poc \) slot for our case.

The optimal number of \( poc \) slots can easily be greater than the \( MaxSimLive \) if we naively superpose interference graphs of individual procedures together.

We show an example in Fig. 2. In this example in Fig. 2, the \( MaxSimLive \) number is 2 at the code points marked by *, **, and ***. At the point marked by *, \( vA_1 \) and \( vA_2 \) are live. At the points marked by **, ***, \{ \( vA_1, vB \} \) are live and \{ \( vA_2, vB \} \) are live respectively.

However, the composed interference graph of \( vA_1 \), \( vA_2 \) and \( vB \) is a clique graph with a chromatic number of 3. If we use this composed interference graph, the number of necessary \( poc \) slots becomes 3, 50% more than the \( MaxSimLive \) number.

In this paper, we show that an approach that uses \( MaxSimLive \) \( poc \) slots for all variables exist. We use a different approach from the SSA-LLP approach that arrives at the same result for space optimality. We further show that we can also optimize the data movement to make it practical. We describe how the approach can be applied to different types of on-chip memory in Section 3.

3. Whole-program On-chip Memory Allocation

In our whole-program on-chip memory allocation framework, we perform allocation in two main iterative stages. In the first stage, we treat register, software cache and hardware cache in the same way, as being one general physical on-chip \( poc \) memory slot. This facilities the optimality analysis of the overall \( poc \) memory amount in the allocation phase as well as the assignment analysis of intra-procedure/inter-procedure \( poc \) variable.

In the second stage, we designate a \( poc \) slot to be in registers, software cache or hardware cache according to the features of the variables that are assigned to this \( poc \) slot and the occupancy constraint. This stage interacts with the
assignment phase in the first stage. It provides feedback on which variables are most suitable in which types of on-chip memory. We feed the information back to the assignment phase in the first stage and assign as many variables of the same type to the same poc slot as possible, while respecting the overall poc memory amount determined in the first stage.

Finally we determine the total usage of each type of on-chip memory and choose the best configuration, for instance, larger software cache and smaller hardware cache or the other way around. We also determine per-thread registers, software cache and hardware cache with respect to overall occupancy. Then we apply the allocation and re-write the binary code. We summarize the two stages on-chip memory allocator in Fig. 3.

### 3.1 Single-procedure Allocation

In single procedure, both strong theoretical results and efficient practical algorithms exist for register/memory coloring. Theoretically an optimal register (memory) coloring number MaxSimLive can be achieved in polynomial time by exploiting the chordal property of the interference graph for a program in static single assignment (SSA) form [15].

With perfect elimination order for chordal graph, one can assign optimal number of colors to all nodes in polynomial time. An ensuing step is to remove the \( \phi \)-function in SSA form that do not exist in real programs and this process can be made more efficient [13] through graph recoloring.

In this paper, we use existing register allocation algorithm for single procedure on-chip memory allocation. We focus on inter-procedure poc memory allocation in this paper. We describe the basic self-compressed stack approach for inter-procedure allocation in the next Section 3.2.

### 3.2 Self-compressed Stack

In this section, we show that it is possible to use the same number of on-chip memory slots as the maximal number of simultaneously live variables MaxSimLive for inter-procedure allocation in GPU programs.

We achieve this by using an inter-procedure allocation approach named self-compressed stack. The basic idea is to maintain a virtual compressible stack for variables across procedure calls.

Every procedure uses a contiguous region in the virtual stack to store its poc memory variables. We assume in this Section, the poc memory allocation within every procedure has already been achieved. A virtual stack frame is mapped to a set of poc memory variable assigned to the same color (during the coloring process).

On the entry of a procedure, the local poc memory stack frames are pushed into the virtual stack. On the exit of a procedure return, the local poc stack frames are popped out of stack.

In addition to regular push/pop operations of the virtual poc stack, in our self-compressed stack, we compress the stack to reuse inactive stack frame. Compression happens when the variable a stack frame is mapped to is not live at a particular call point (where a sub-procedure is called). Since one stack frame is mapped to multiple variables, it can be active and inactive at different points.

There are two different ways to compress the virtual poc memory stack: by only adjusting the stack pointer and by both adjusting the stack pointer and moving poc memory variable. We show an example in Fig. 4. In Fig. 4, we mark a stack frame as inactive by filling it with crossed lines. Fig. 4 shows that the inactive stack frame is right on top of the stack, therefore only the stack pointer needs to be adjusted to compress the stack. Fig. 4 shows (b) that the inactive stack frame is not on the top of the stack, therefore we move the content of inactive stack frame \( s_{S4} \) to fill the inactive stack frame and to make a contiguous active stack frame region. Then adjust the stack frame pointer to finish the compression.

The compression code is statically generated at compile-time and it automatically moves data at runtime. It bootstraps by compressing and decompressing itself. Therefore, we name it self-compressed stack.

We show our compile-time algorithm to generate code that compresses the stack in Fig. 5. On the entry of a procedure call, it checks if the caller’s stack has any inactive stack frame. If so, it generates code right before the procedure call to compresses the stack. On the exit, it generates code right after the procedure call, to restore the caller’s stack to the original layout. Therefore, memory offset used to access the same stack frame after the procedure call do not need to be transformed.

The code for compression and decompression should be inserted in the caller procedure. The key insight here is
that single procedure allocations is independent due to the compression and decompression steps.

In Fig. 5 (c), we show the algorithm to perform data movement for compression and decompression. It generates code to iteratively “swap” the top active stack frame with the bottom inactive stack frame until all active stack frames are placed consecutively. In the decompression step, it emits code for swaps in the opposite direction. Note that, the swap” here is actually a single direction move.

3.3 Self-compressed Stack Optimality

Assuming single procedure allocation is already performed locally, it is trivial to see that the basic self-compressed stack uses an optimal number of on-chip memory slots for cross-procedure allocation. Note that it also guarantees the optimal number of physical on-chip memory slots to be MaxSimLive when the program is represented in SSA form.

The self-compressed stack scheme requires minimal number of poc memory slots – MaxSimLive to store all live variables for a multi-procedure GPU program represented in SSA form.

Proof We sketch the proof as follows. With self-compressed stack scheme, we show that at any execution point of a GPU program P, the on-chip memory stack depth $d(P)$ is bounded by the number of simultaneously live variables $d(P) \leq \text{MaxSimLive}$.

Let’s assume we are in the middle of executing a program and we are in procedure $p$. The on-chip memory stack depth $d(P)$ at this point is equivalent to $p$’s local stack size $|\text{plocal_stack}|$ plus the stack size of its calling context $d(\text{pcontext})$. Since self-compressed stack only allows active stack frames in the calling context stack, the stack size of the $p$’s calling context $\text{pcontext}$ is equivalent to the simultaneously live variables in the context.

Since $p$ is represented in SSA form and the interference graph for SSA form is chordal, the number of poc memory slots for $p$ (the number of colors to color the chordal graph) is also bounded by the number of the simultaneously live local variables in $p$ [15]. Therefore the summation $d(P) = d(\text{pcontext}) + |\text{plocal_stack}|$ is bounded by the total number of simultaneously live variables at this point of execution $d(P) \leq \text{MaxSimLive}$. Since we also at least need $\text{MaxSimLive}$ poc memory slots, $d(P) \geq \text{MaxSimLive}$, $d(P) = \text{MaxLiveSim}$. The optimality is then proved.

3.4 Self-Compressed Stack Assignment

Our basic self-compressed stack introduced above minimizes caller stack size by swapping pairs of inactive and active stack frames. The number of swaps is the overhead of data movement in the self-compressed stack. The SSA-LLP[5] method does not do swapping. Instead, it copies all the live variables in the calling context into another global register region. Apparently, it costs a lot of data movements when there are a lot of live variables in the calling context, which occurs quite often based on our experiment results, for instance, the CFD benchmark in Rodinia[8] has about 60 variables live when it calls a sub-procedure. Ours is already better than SSA-LLP allocator. But we show that we can further improve it.

Before we describe our optimal stack movement strategy, we want to point out that in conventional CPU register allocation the address of a register does not matter, meaning a color can be mapped to any register address. However, the index of the on-chip memory slot a color is mapped to has non-trivial performance impact when we consider data movements.

We illustrate the importance of poc memory index selection Fig. 6. In Fig. 6, we use the column to show the active/inactive status of a slot $\text{slot}_i$ at different call points in a program. If a box corresponding to a call point is in solid blue color, then the slot (the variable it is mapped to at that point) is live. Otherwise if it is in grey color, the slot (the variable it is mapped to) is not live. There are three call points in the program in Fig. 6.

Fig. 6 shows with two different ways for the layout of the on-chip memory slots, respectively in Fig. 6 (a) and Fig. 6 (b). The total number of inserted “MOV” (or the
Figure 6: The impact of on-chip memory slot layout on the number of data movements at compression and decompression phase. Every column represents a slot. The blocks in the column represent the live variables of the variable assigned to the slot. If the block is in blue, the variable is live, otherwise it is not. Bold arrow represents direction of data movement for the inserted compression instruction.

Table 1: Notations

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSi</td>
<td>i-th set of variables mapped to the same stack frame</td>
</tr>
<tr>
<td>SLOTi</td>
<td>i-th slot from the bottom of the stack</td>
</tr>
<tr>
<td>X_{ijk}</td>
<td>mapping between SSi and j-th slot (0 or 1)</td>
</tr>
<tr>
<td>L_{ik}</td>
<td>liveness of SSi at k-th sub-procedure call</td>
</tr>
<tr>
<td>B_{ij}</td>
<td>desired stack height at k-th sub-procedure call</td>
</tr>
<tr>
<td>C_{ijk}</td>
<td># of swaps incurred by placing SSi at j-th slot for k-th sub-procedure</td>
</tr>
<tr>
<td>W_{ij}</td>
<td># of swaps incurred by placing SSi at j-th slot</td>
</tr>
<tr>
<td>N</td>
<td># of sub-procedure calls</td>
</tr>
<tr>
<td>M</td>
<td>maximum # of simultaneously live variables in this procedure</td>
</tr>
<tr>
<td>P_{k}^{swap}</td>
<td># of swaps incurred because of k-th sub-procedure call</td>
</tr>
</tbody>
</table>

We denote the number of swaps incurred for compression/decompression at k-th procedure call as $P_{k}^{swap}$. The objective is to find the poc slot each variable set $SSi$ for $i = 0...M-1$ is mapped to such that the total number of swaps is minimal.

$$\min \text{Total}_\text{swap} = \sum_{k=0}^{N-1} P_{k}^{\text{swap}}$$

We reduce this problem into a maximum-weight bipartite matching problem. In order to do this, we have the following corollary.

**Corollary 1.** With the notations defined in Table 1, in the minimal-swap-assignment (MSA) problem, the total number of swaps contributed by placing an arbitrary variable set $SSi$ at an arbitrary location $j$--th poc memory slot SLOTj 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 to at most $B_{ik}$ slots, we have:

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

while

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

**Proof** Since a swap will be invoked if and only if there is an active stack slot placed beyond the top of the desired compressed stack, we can determine if a placement of a $SSi$ set will incur a swap by checking its location.

If and only if $SSi$ is live across 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 $B_{ik}$, $j \geq B_{ik}$, the number of swaps invoked by placing set $SSi$ at SLOTj for $k$-th sub-procedure $C_{ijk}$ is 1; otherwise, $C_{ijk}$ is 0. Therefore, we get all the swaps contributed by placing $SSi$ into SLOTj by summing up $C_{ijk}$ for $k = 0...N-1$. The corollary is thus proved.

Next, we reduce the minimal-swap-assignment problem into a maximum-weight 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 a matching such that the sum of the weights of the edges in the matching is maximal as shown in Fig. 7.

We let one of the two disjoint sets of nodes correspond to the sets of variables -- $SSi$ for $i = 0...M-1$. The other set of nodes correspond to the poc memory slots in a particular order $SLOT_0...SLOT_{M-1}$ 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 $SSi$ and the $j$-th stack slot as the negative of $W_{ij}$ defined in Corollary.
4. On-Chip Memory Mapping

Section 3 discusses the first stage of the whole program on-chip memory allocator, which is mapping every variable into a physical on-chip memory slot. In this Section, we describe the second stage, which determines the type of physical on-chip memory slot a variable is mapped to.

The overall on-chip memory mapping stage consists of four steps: 1) total on-chip memory demand analysis, 2) partition per-thread on-chip memory demand into per-thread register demand, shared memory and cache demand, 3) perform on-chip memory allocation at register level, shared memory level and cache level, and 4) transform binary instructions according to the allocation strategy.

4.1 On-chip Memory Demand

The minimal number of on-chip memory slots in self-compressed stack scheme is the maximal number of simultaneously live variables MaxSimLive if we use SSA form and perform copy coalescing. MaxSimLive of a multi-procedure program can be found using the MaxSimLive of every individual procedure through a post topological order traversal of the call graph.

Using SSA form and enhanced copy coalescing [14] gives optimal number of poc memory slots. However, copy coalescing uses non-trivial amount of move instructions in some cases. Therefore, we choose over 2 approaches for intra-procedure allocation. We do allocation on merged SSA variables in which the variables in φ—function are union into one variable. We achieve this by finding connected components in a graph represents the relationship between all φ—function variables. Then we use the chordal graph allocation algorithm. If it incurs any spills, we fall back to the traditional Chaitin-Briggs allocator [4] that uses a stack to track the order of the variables to be colored and we also perform iterative register coalescing [12].

Therefore, we either use MaxSimLive as the overall on-chip memory demand or use the maximum runtime stack depth given intra-procedure allocation result using Chaitin-Briggs allocation algorithm.

In recursive procedures, the runtime stack size may grow according to the input, which is not determinable at compile-time. We do not place any variable that is live across the call to a procedure itself (directly or indirectly) into on-chip memory for recursive procedure.

4.2 On-chip Memory Type Determination

A variable that is mapped to a physical on-chip memory slot can be placed in registers, software cache or hardware cache. They are have advantages and disadvantages. We first determine the total demand of every type of on-chip memory, then we determine the best software and hardware cache configuration.

Register selection We choose the variables to place in physical registers according to the following principle: (1) we prioritize the variables that are not live across any sub-procedure call, and (2) we prioritize the variables that are more frequently used (especially the ones in loops). We use these two parameters to rank all variables.

Variables that are live across procedure calls need to be managed using the self-compression stack approach. In the self-compression stack based allocation approach, the stack slot index is relative to stack pointer. If a procedure is called in difference contexts, its absolute stack location might vary. However, for registers, we need to have a non-variable (fixed) register number for all local variables at compile-time. Therefore, it’s better to place these variables into software cache and hardware cache. In the worst case, if we run out of software or hardware cache size, we
keep register stack and still do compression/decompression. The approach we adopted is that for a sub-procedure, we check all the places it is called and determine the maximum number of registers in its calling context. Then we let this sub-procedure’s register usage starts from this register number that hasn’t been used by any procedure in this sub-procedure’s calling context. We do this by building a call point graph [5] and recursively update the base register number in the graph.

**Software Cache Selection**  We select another set of variables to place in $V_{\text{smem}}$ shared memory. We prioritize the variables that are live across procedure calls. We live range length of a variable [16] as tie breaker.

Note that we organize shared memory (software cache) stack in a way that avoids bank conflicts. We place adjacent shared memory stack slots for one thread with a stride being the number of threads in a block multiplied by the size of every variable. This layout avoids bank conflicts entirely.

**Hardware Cache Selection**  For variables to be placed in hardware cache, we prioritize the variables that are frequently used in shorter time period. It is because hardware cache can recognize data that is frequently reused in shorter time, the data that has good temporal locality.

Note that hardware cache allocation is essentially local memory allocation. In some GPU architecture, the L1 cache is solely used for local memory variables.

Further, if a variable can fit into either software cache or hardware cache, we let it be placed into hardware cache, since software cache can be saved for programmers to do other types of optimization. Hardware cache is usually more difficult to be exploited by programmers.

Based on the above selection criteria, we give every variable a prioritized variable type, register or cache. A physical on-chip memory slot is mapped to a set of variables and thus we check how many variables in this set are mapped to registers or software/hardware cache type. We determine the type of the poc memory slot using the most frequent type of its variables.

If it is difficult to determine for a lot poc memory slots, we feedback the information of variable types to the graph coloring phase in the first stage described in Section 3 and perform graph coloring again. It is because at the graph coloring phase, at every step, a node can be colored in different ways without violating the interference constraint. Therefore, using the variable type information, we assign a color that is the same as the color of a previous colored variables of the same type, if it does not violate the interference constraint. After the recoloring phase, we perform type determination again. We iteratively run the process until the type determination results cannot be improved anymore.

Then we can determine the best amount of register, software cache and hardware cache. Since current GPU architectures provide at most 3 size configurations, we choose the one that is closest to the desired partition. Then we perform actual per-thread on-chip memory assignment based on the selected configuration. We do this with respect to the best achievable concurrency described in next Section 4.3.

### 4.3 Concurrency Adaptation

Since the register and software cache variables for concurrent threads need fit into registers and software cache, the maximal achievable concurrency is inversely proportional to per-thread on-chip memory usage$^1$.

The achievable concurrency is the minimal of the two concurrency upper-bounds determined by registers, shared memory (software cache) per-thread. If we want to fit everything into hardware cache as well, it is the minimal of the three concurrency upper-bounds.

Assume we need $V_{po}c$ memory slots per-thread. The total number of physical registers is $N_{\text{reg}}$. The total number of software cache slots is $N_{\text{smem}}$. The total number of hardware cache slots is $N_{\text{cache}}$. The user allocated software cache per-thread is $V_{\text{user,smem}}$. We need to determine the three free variables: $V_{\text{reg}}$ – per-thread register usage, $V_{\text{smem}}$ – per-thread shared memory usage, and $V_{\text{cache}}$ – per-thread cache usage, that gives us best concurrency.

The achievable concurrency level is:

$$\text{ConLevel} = \min \left( \frac{N_{\text{reg}}}{V_{\text{reg}}}, \frac{N_{\text{smem}}}{V_{\text{smem}} + V_{\text{smem, user}}}, \frac{N_{\text{cache}}}{V_{\text{cache}}} \right)$$

subject to these constraints:

$$V_{\text{onchip}} = V_{\text{reg}} + V_{\text{smem}} + V_{\text{cache}} \text{ and } V_{\text{reg}}, V_{\text{smem}}, V_{\text{cache}} \geq 0 \quad (2)$$

**Given $V_{\text{onchip}}$, allowable on-chip memory slots to use for all variables, the optimal concurrency is either the value obtained by making $N_{\text{reg}}, N_{\text{smem}}$, and $N_{\text{cache}}$ equivalent if these three items can be equivalent, or if the three items cannot be equivalent, with the constraints: (a) $V_{\text{reg}} + V_{\text{smem}} + V_{\text{cache}} = V_{\text{onchip}}$, (b) the register, shared memory or cache portion is greater than or equal to 0.**

**Proof**  If the three items in equation (1) can be made equivalent, we call the resulting concurrency equilibrium concurrency. Assume there is another partition of the $V_{\text{onchip}}$ on-chip memory slots which generates a better concurrency level. We call this partition $P$. Since $P$ generates a higher concurrency level than the equilibrium concurrency, at least one of the denominators in equation (1) should be must be smaller than the one achieved in case of equilibrium concurrency. Since all denominators in Equation (1) sum up to a fixed value, there must be another denominator which becomes larger; this makes the corresponding concurrency upper-bound smaller than equilibrium concurrency in partition $P$. The final concurrency is the minimal of the three

---

$^1$Since the number of active threads need to be rounded up to a multiple of thread block sizes, and the registers usage need to be aligned according to register bank size constraints, we need to further adjust the concurrency level. We use the formula in the NVIDIA occupancy calculator [23] to get accurate optimal concurrency level.
concurrency upper-bounds. Partition $P$ leads to an achievable concurrency smaller than the equilibrium concurrency, which contradicts the assumption.

The point (b) in Theorem 4.3 handles the case when equilibrium concurrency cannot be achieved. It is possible when $N_{\text{smem}}/V_{\text{smem-user}}$ is smaller than equilibrium concurrency. Since the variables $V_{\text{smem}}$, $V_{\text{reg}}$, $V_{\text{cache}}$ cannot be negative, the optimal concurrency level can’t be larger than $N_{\text{smem}}$. In this case, the optimal concurrency is $N_{\text{smem}}/V_{\text{smem-user}}$.

### 4.4 Putting It All Together

We summarize the whole program on-chip memory allocator process in Fig. 3. We start with compiler analysis that obtains the set of variables that can be placed in registers. We initially use the SSA form for variable live range analysis. Then we analyze how many physical on-chip memory slots per-thread are needed.

Next we partition per-thread on-chip memory usage into per-thread register, software cache and hardware cache usage according to techniques in Section 4.2.

Then we determine the portion of each type of on-chip memory based on both software/hardware cache size configuration and best achievable concurrency. Then we designate every poc memory slot to every specific type. We perform code generation to emit code at compile-time for compression/decompression at runtime. For software cache and hardware cache, it is mainly stack operations. For registers, it involves multiple “MOV” instructions and careful register assignment in the allocation phase.

Finally, we apply the allocation result in the final binary code and update the exact number of registers used, the actual amount of shared memory (software cache) and local memory used in the elf file.

**Discussion** The maximal stack depth can be calculated at compile-time under the assumption that the call graph can be resolved at compile-time. If a procedure takes function pointers as input, and the input cannot be resolved at compile-time, then we cannot determine the static call graph. However, as far as we know, current GPU kernels do not allow function pointer as input parameters. Besides, even current GPU programs allow virtual function, it does not allow virtual function for device functions, which are the only functions a GPU kernel calls. Therefore our assumption in this paper is a realistic assumption.

### 5. Evaluation

#### 5.1 Methodology

To evaluate the techniques presented in this paper, we implemented a prototype on-chip memory allocator. It takes a GPU binary generated by nvcc compiler as input. Given the binary file, we extract high level program information. We reconstruct control flow graph (CFG), call graph and perform liveness analysis on register variable and off-chip memory (spilling) variable. We generate single static assignment (SSA) for the register and spilled variable information present in the binary. Once we have SSA, we perform whole-program on-chip memory allocation and we rewrite the binary back.

We used the reverse-engineered NVIDIA binary instruction set architecture (ISA) based on the open-source work [17] and [16]. We use libelf to modify the other information of the binary, including modified register number, shared memory usage, local stack size and binary size. Our allocator handles different types GPU architecture, from CUDA computing capability 2.0 to 3.0. Note that our allocator framework would also work for other types of GPU architecture as long as we add a new front end to parse the binary and a new backend to perform rewriting. The internal transformation and analysis is the same.

#### 5.2 Platform and Benchmarks

<table>
<thead>
<tr>
<th>Codename</th>
<th>GPU</th>
<th>CUDA Cap</th>
<th>Reg</th>
<th>SCache + HM Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kepler</td>
<td>GTX680</td>
<td>SMX (192 Core)</td>
<td>64K</td>
<td>64 KB</td>
</tr>
<tr>
<td>Fermi</td>
<td>Tesla C2075</td>
<td>448 (1152 Core)</td>
<td>32K</td>
<td>64 KB</td>
</tr>
</tbody>
</table>

Table 2: Machine Configuration

We perform experiments on two different machine configurations. One is NVIDIA Kepler GTX680. It has 8 streaming multi-processors (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 64KB shared memory/L1 cache. 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 64KB shared memory and L1 cache per SM. The maximal number of concurrent threads that can be executed simultaneously is 1536. We list the configuration in Table 2.

Note that these two configurations impose different constraints on the single thread register count for maximal concurrency provided by hardware. The first configuration allows at most 32 registers per thread and the second configuration allows at most 20 registers per thread for best hardware concurrency. We refer to the first machine configuration as Kepler and the second one as Fermi.

We evaluate the effectiveness of our allocator on 11 benchmarks shown in Table 3. These benchmarks come from Rodinia [8] and CUDA Computing SDK [24]. We excluded the benchmarks from Rodinia [8] that require less than 20 registers per-thread. First of all, these benchmarks have low register pressure. Secondly, they have already achieved the maximum hardware concurrency for the reason mentioned above. Reducing register pressure wouldn’t increase concurrency or releasing significant number of registers (the num-
ber is already small). Similarly we excluded a lot of CUDA Computing SDK benchmarks since they have low register pressure.

In Table 3 we present more detailed information for the 11 benchmarks. We notice that for the same program, the generated GPU binaries use different set of instructions for these two different architectures even if we compile them with the same CUDA version 5.5. This is partly a result of the different hardware computing constraints, and partially due to the compiler using different instruction set architecture and optimizations for each architecture. In Table 3 “User Shared-Mem” indicates if a GPU benchmark has user-allocated shared memory (software-managed cache). If a program already has shared memory pre-allocated by the user, our allocator handles it as well. We detect the amount of pre-allocated shared memory and choose to append our allocated shared memory to it. Recall that in Section 4.3, we also use \texttt{smem\_user} to help determine best concurrency.

We run every benchmark using different thread block sizes: 256, 384, 512, 768 and 1024. Note that some benchmarks have excessive pre-allocated shared memory and therefore prohibit larger thread block size. For these benchmarks, we run them to the largest block size as possible.

For every program, we generate several GPU binaries. The first one is the default one when we compile with the highest \texttt{nvcc} optimization flag or the compilation flags described in the benchmark’s \texttt{Makefile}. This one is the default binary generated in the original benchmark.

By default, \texttt{nvcc} uses the maximum number of registers it can, which is typically sub-optimal [16] because it does not consider the interaction between different on-chip memory resources and concurrency. We will show how our transformation are better than the default one. Next we run a pre-analysis on the default binary and check how many registers we can save by our allocator in this paper. Then we use this estimated number and pass it as register limit to \texttt{nvcc}. Note \texttt{nvcc} does not choose best per-thread register number for user, but it provides the option of letting user set per-thread register limit as a compilation flag. We then perform our transformation on this register number. The reason is that we find out \texttt{nvcc} might be selecting different instructions for different register number limit. When \texttt{nvcc} is using maximum register number possible, it performs aggressive optimization and uses a lot of on-chip memory resource. When \texttt{nvcc} is given a smaller register number, it select a different set of instructions that requires less on-chip memory, amenable to further optimization. We compare these two versions and choose the better one (by profiling) as our next comparison baseline. We perform transformation on the better binary and compare with it as well.

\textsuperscript{2} It is also the register number that gives us nearest best occupancy we can achieve based on our pre-analysis. The occupancy is also a multiply of thread block size. So we need to take thread block size into consideration.
Table 3: Detailed benchmark information. “UserSmem” represents if the original program used shared memory or not. “Cache Config” shows the best cache configuration selected by the allocator. The entry to the left of “/” is for Fermi, and the entry to the right of “/” is for Kepler.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>AppDomain</th>
<th>Cache Config</th>
<th>UserSmem</th>
</tr>
</thead>
<tbody>
<tr>
<td>FDTD3d</td>
<td>Machine Learning</td>
<td>L. Cache/L. Cache</td>
<td>Yes</td>
</tr>
<tr>
<td>Heartwall</td>
<td>Machine Learning</td>
<td>L. Cache/L. Cache</td>
<td>Yes</td>
</tr>
<tr>
<td>ImageClassifying</td>
<td>Machine Learning</td>
<td>L. Cache/L. Cache</td>
<td>Yes</td>
</tr>
<tr>
<td>Imagery</td>
<td>Machine Learning</td>
<td>L. Cache/L. Cache</td>
<td>Yes</td>
</tr>
<tr>
<td>Particles</td>
<td>Machine Learning</td>
<td>L. Cache/L. Cache</td>
<td>Yes</td>
</tr>
<tr>
<td>FastMD</td>
<td>Machine Learning</td>
<td>L. Cache/L. Cache</td>
<td>Yes</td>
</tr>
<tr>
<td>SortCluster</td>
<td>Dynamic Programming</td>
<td>L. Cache/L. Cache</td>
<td>Yes</td>
</tr>
<tr>
<td>StreamCluster</td>
<td>Dynamic Programming</td>
<td>L. Cache/L. Cache</td>
<td>Yes</td>
</tr>
</tbody>
</table>

5.3 Results Discussion

We first present summary of the evaluation results. Figure 8 shows the speedup of different benchmarks on Fermi and Kepler. The baseline is the `nvcc`-generated binary on which we perform transformation. The left most bar in the chart depicts the minimum speedup across different block-sizes. The middle bar shows the average speedup and the right most bar represents the maximum speedup attained. Note that because Fermi has smaller register threshold — 20 to enable maximal hardware concurrency, therefore we have more benchmarks in the Fermi graph. Our experimental results demonstrates noticeable improvement in almost every case here. It can be seen clearly that several benchmarks have substantial speedup, with the best speedup of 1.81 on Fermi and 1.49 on Kepler respectively for FDTD3d.

In a couple of benchmarks, we do not gain significant performance improvement, even when a nontrivial number of registers is reduced; this occurs on Fermi and Kepler with the `recursiveGaussian` benchmark. This is the result of Fermi’s usage of the L1 cache for both global and local memory and `recursiveGaussian` makes use of multiple global/local arrays which pollute the cache. However, even there is no speedup, there is little performance degradation on average, indicating the low overhead of our on-chip memory allocator.

Figure 9 illustrates the cache miss rate reduction on Fermi and Kepler for the first type of benchmarks. The left bar shows the reduction on Fermi and right bar depicts the reduction on Kepler across different benchmarks. The baseline is the case where all the variables that should go to both shared memory and L1-cache in my transformation go to L1-cache only. It is evident that high cache miss reduction is obtained for most benchmarks. This occurs either when the shared-memory shared the pressure of registers with cache or when all variables are stored in registers and shared memory, meaning no cache requests are made. In certain cases, when the variables requires smaller number of on-chip memory slots, this is possible.

Figure 10 represents the normalized register usage across all different benchmarks, including both types of benchmarks. The baseline in this case is the default binary generated by `nvcc`. Recall that `nvcc` by default uses as many registers possible so that no spilling occurs or it hits the per-thread register limit set by the ISA, for instance, 63 for CUDA computing capability 3.0 (8 bits reserved for register number in the binary instruction). The left bar depicts the results obtained on Fermi and the right bar shows the results on Kepler. The baseline is the original register usage. We compare our automatically transformed binary with the default one generated by `nvcc`. The smaller a bar is, the better. The register pressure is reduced nearly all benchmarks. The lowest values obtained for Fermi is 47% while for Kepler it is 66%, which indicates as much as 54% register pressure is reduced for Kepler and 34% register pressured reduced for Fermi. As mentioned before, the reason is that `nvcc` splurges registers without considering the interaction with resource allocation and concurrency. The results suggest that, even if concurrency improvement does not contribute to much performance improvement (nor much average slow down as mentioned), the register number is significantly reduce and it can contribute to other types of optimizations by programmers. For instance, the matrix multiplication kernel in CUBLAS uses excessive number of registers for improved instruction level parallelism (ILP).

Finally in Table 3, we also show the cache configuration selected by our allocator. Cache Config. column shows the final cache configuration selected for every benchmark. The left number is for Fermi and the right number is for Kepler.

6. Related Work

In this paper, we proposed shared memory allocation approach modeled as the graph coloring problem, similar to register allocation work in the past. Register allocation has been extensively studied in decades [6] [7] [27] [2] [15] [26] [3] [9] [29] [20]. Most of them have been focused on single procedure register allocation. A few of them [19] [10] have studied reuse of registers across procedures but mainly focus on minimizing register pressure penalty at procedure calls. Typically, most of the registers in the caller procedure are saved in local memory so that they can be reused at callee procedure. Previous work [19] [10] avoid saving all the registers when procedure calls happen by determining if the registers will be used or not in the callee procedure. The traditional approaches start from the baseline that all registers should be saved at procedure calls (either by caller or callee) and then prune the number of register stores/re-stores, instead of considering the interference relationship among all live variables across procedures in the same program. Further, most of previous register allocation studies are for sequential CPU programs, instead of parallel GPU architecture. In [1], the authors studied the register allocation for vector machines. However, GPU architecture is different from the vector processor architecture.

The most relevant GPU work [28] is to identify the opportunities in saving register for control divergent statements
in GPU programs. However, it does not consider register saving opportunities for general statements in GPUs. Other relevant literature on GPU architecture includes GPU exception handling [21], where register states need to be restored for resuming execution after exception, and register/shared-memory size reconfiguring [11] according to register/shared-memory pressure in different kernels. Neither of them tries to determine register pressure or reduce register pressure. Furthermore, reconfiguring register or shared memory size incurs runtime overhead every time the GPU kernel is invoked, while our shared memory allocation approach only happens at compile-time without incurring any runtime overhead.

7. Conclusion
In this paper, we propose a on-chip memory management scheme for GPU programs. It aims to perform inter-procedure on-chip memory allocation for register variables. Our exploration in this paper leads to an optimal polynomial time inter-procedure allocation strategy. Our on-chip memory allocator improves performance by up to 1.81 times over the mcci compiler for a large representative set of applications on two different types of GPU architecture.

References
[25] NVIDIA. NVIDIA’s next generation CUDA compute architecture: Kepler GK110. URL http:


