Register Allocation using GPUs

Nilesh Mahajan, Kailash Nagarkoti

December 17, 2012
Abstract

Register allocation is one the most important compiler passes. The quality of the generated code both in terms of code size and performance depends on this important pass. Unfortunately, the problem is NP-complete so unless $P$ equals $NP$, there is no efficient solution to this problem. Thus various heuristics are applied to get an approximate solution. Our approach is to leverage the parallelism provided by GPUs to solve this problem.
0.1 Introduction

Computer Memory is arranged hierarchically as seen in Figure 1. Register access typically takes a single cycle but there are few registers. Register allocation maps variables in a program to registers. Because there are few registers, the compiler might not be able to assign a register to some variables. This might also happen because the heuristic might fail even if there was a valid assignment. In this case, the compiler spills a variable and assigns it to memory. Thus, we want to assign as many variables to registers as possible and spill those that are not used frequently.

![Figure 1: Memory Hierarchy](image)

A straight-line sequence of instructions in a program is called a basic block. Basic blocks form the control flow graph of the program. A point between two consecutive instructions in the control flow graph is called a program point. An instruction may assign value to a variable in which case that instruction defines the variable. An instruction may use a variable’s value in which case that instruction uses the variable. We say that a variable \( v \) is alive at a program point \( p \) if there is a path from \( p \) to an instruction that uses \( v \) where \( v \) is not re-defined by any instruction. The collection of program points where a variable is alive is called its live range. We can assign the same register to two variables that are not live at the same time or equivalently their live ranges do not overlap.

Some register allocation terminology

0.1.1 Spilling

There might not be enough registers to hold all variables, so some variables need to be assigned to memory. The act of storing a variable to memory and loading
it when the value is needed, is called spilling. We want to minimize spilling as much as possible.

0.1.2 Coalescing

If there is an assignment of the form \( v_1 := v_2 \), in the program and \( v_1, v_2 \) do not interfere, then we can allocate the same register to \( v_1 \) and \( v_2 \) and remove the assignment. This is called coalescing.

0.1.3 Splitting

We might want to split a long live range, by introducing move instructions in the code. This leads to lesser number of registers required because lesser variables interfere. This increases code size because we are inserting new assignments but might lead to lesser spills. This is the opposite of coalescing.

0.2 Related Work

Researchers have been trying to parallelize computationally-intensive compiler analyses recently\[5\], \[7\]. The literature overall is sparse on this and to our knowledge we have not seen anyone trying to use GPUs to parallelize register allocation or other backend passes.

0.3 Register Allocation Approaches

Different approaches to register allocation have been studied over the years.

0.3.1 Graph Coloring

We used the simpler Chaitin graph coloring algorithm\[1\] shown in Figure 2. This proceeds in four stages:

1. **Build**: construct the interference graph. This was done on CPU using ROSE’s in-built liveness analysis. There were some problems with that analysis that took time to resolve. We finally ended up using the interprocedural liveness analysis done by the new dataflow framework.

2. **Coalesce**: color the graph using a simple heuristic. Suppose the graph \( G \) contains a node \( m \) with fewer than \( K \) neighbors, where \( K \) is the number of registers on the machine. Let \( G' = G - \{m\} \) obtained by removing \( m \). If \( G' \) can be colored, then so can \( G \), for when adding \( m \) to the colored graph \( G' \) the neighbors of \( m \) have at most \( K - 1 \) colors among them; so a free color can always be found for \( m \). This leads naturally to a stack-based algorithm for coloring: repeatedly remove (and push on a stack) nodes of degree less than \( K \). Each such simplification will decrease
the degrees of other nodes, leading to more opportunity for simplification. The graph coloring paper\cite{2} we saw partitions the matrix blockwise. They use adjacency list to represent the matrix. We use the Compressed Sparse Row (CSR) format to represent the sparse matrix. The interference graphs we generated did not have many nodes. We wanted to use METIS\cite{4} to generate graph partitions with minimal cross-edges and balanced partitions. That seems to be an overkill now for such small graphs. Another difference is they don’t simplify the graph by deleting nodes and reconstruct it. Our kernel takes two arrays $C$ and $R$ representing the matrix. An output array $colors$ denotes the assignment. The kernel is also passed an array denoting degree per vertex, and an array for the vertex ordering. The kernel calculates the degrees per vertex. It then reduces the degrees according to the heuristic and calculates the vertex ordering to assign colors. If there was a spill, $-1$ is put in the output array, and the kernel returns. The node is spilled and graph is rebuilt on CPU. If there was a valid $k$-coloring, $k$ being the total number of registers, the output array contains the color assignment.

3. **Spill**: This is done on CPU where the a node is selected and the live ranges are split, and the graph is rebuilt.

4. **Select**: assigns colors to nodes in the graph.

### 0.3.2 Linear Scan

The graph coloring approach is a little expensive so researchers have tried faster approaches like linear scan\cite{6}. The algorithm starts by linearizing the basic
blocks of the source program, that is, arranging these blocks in a linear sequence; the exact ordering chosen is not important and will not compromise the correctness of the results. Given this linearization, linear scan replaces the live range of each variable with a contiguous interval, called the variable lifeline, and then proceeds to color these intervals. This greedy algorithm is inherently sequential in that it goes through the intervals sequentially spilling the longest interval(s). We decided against implementing this on GPU.

0.3.3 SSA-based Allocation

The interference graphs generated after SSA conversion are chordal[3]. Chordal graphs can be colored optimally in polynomial time. The problem is getting out of SSA will generate copies and how to generate minimum copies is NP-complete. One interesting aspect of this is to calculate SSA using GPUs but that is non-trivial effort.

0.4 Implementation Status

We can generate the interference graphs. The interference graph is stored in Compressed Storage format (CSR). CSR format is a widely used scheme for storing sparse graphs. The adjacency structure of graph with n vertices and m edges is represented using two arrays rowptr and adjlist. The rowptr array is of size n + 1 whereas adjlist is of size 2m (because for each edge between vertices v and u we store both (u, v) and (v, u). The rowptr contains pointer to start of an adjacency list in the adjlist for the particular index e.g. adjacency list for i would start at rowptr[i] and end at rowptr[i + 1] (but not including i + 1).

We use METIS library for partitioning to reduce the number of crosscutting edges and for proper load balancing. The two APIs that we use are METIS_PartGraphRecursive and PartGraphKway.
0.4.1 Heuristics

To sequentially color a graph there are many heuristics. Greedy coloring is one such heuristic which focusses on carefully selecting the next vertex to be colored. In these techniques once a vertex is colored the color never changes. Two of the techniques implemented in this project are First Fit and Degree Based Ordering.

1. **First Fit**: First Fit works by fixing any arbitrary order of vertices for coloring and then assigning them the smallest color not already assigned to one of its neighbors.

2. **Largest Degree Ordering - LDO**: Largest Degree Ordering chooses the vertex with maximum degree i.e. largest number of neighbors. It checks the color of the neighbors and assigns the lowest color, not same as any of the neighbors, to the vertex. This continues until the graph is fully colored. This runs in quadratic time. Program Implementation: The submitted version implements sequential LDO by reading an input graph file in adjacency list format and builds a data structure to store the graph in CSR format.

3. **Saturation Degree Ordering (SDO)**: Saturation degrees is defined by the number of distinct colors of the adjacent vertices. SDO is similar to LDO, in this the vertices are ordered by their saturation degree.

4. **Incidence Degree Ordering (IDO)**: IDO is a variant of SDO, the degree of vertex is defined as the number of colored vertices in the neighborhood.

The naive GPU graph coloring is also implemented. We wanted to use different coloring heuristics for which we have sequential implementations. But we were not able to do that and take any performance measurements. Given more time, we will be able to complete the implementation.

0.5 Future Work

We have to test the implementation on larger functions where the interference graphs are sufficiently large so that the CPU-GPU transfer cost can be amortized. There were two other ideas to amortize this cost

1. Run several heuristics in parallel by different blocks/warps to find a valid k-coloring. Even if one of them succeeds, we have avoided the costly spilling.

2. Color the graph of one method on GPU while building the graph of another method on CPU. We started with this optimization but unfortunately faced a lot of technical issues with ROSE and cuda integration. Again we need time to resolve those.
It might actually be worthwhile to look at LLVM backends to see if we can replace their backend passes like register allocation, instruction scheduling to make use of GPUs. If we find that the CPU-GPU transfers are more harmful then we can try CPU parallelization of these. We believe that the parallelization will help with these important but $NP$ – complete problems.
Bibliography


