An Exploration of the Performance of Parallel Bin Packing Algorithms

Zachary Pomper, Maxwell Johnson
December 15, 2018

Project Website: http://www.contrib.andrew.cmu.edu/~msjohnso/418/

SUMMARY

Over the course of our project, we implemented and parallelized two heuristic algorithms to solve the bin packing problem. We chose one commonly used deterministic algorithm, Best-Fit Decreasing (BFD), and one randomized algorithm, WalkPack. We first implemented both algorithms sequentially in C++, then parallelized both using Nvidia’s CUDA. Neither algorithm is designed to be fast, and the different techniques used make a comparison between the speed of each algorithm misleading. Instead, our project focuses on comparing the quality of the solutions generated by the two algorithms and how amenable each is to parallelization. WalkPack was parallelized for the GPU using a Java library in a research paper from RIT [1]. Over the course of our project, we attempt to improve upon the speedup they achieved by using CUDA directly.

BACKGROUND

Bin packing is a known NP-hard problem with applications in task scheduling, logistics, and virtual machine allocation, and a host of other areas. Briefly put, the bin packing problem is this: given a set of objects of various sizes and an unlimited supply of uniformly-sized bins, what is the assignment of objects to bins that minimizes the number of bins required to hold all objects? Several approximation algorithms, including Best-Fit Decreasing, have a proven upper bound of
1 + 11/9 · optimal bins. The same guarantee does not hold for randomized algorithms such as WalkPack, although they consistently have averages much closer to optimal than 11/9 · optimal.

Best-Fit Decreasing is simple to describe and implement, making it a popular choice for solving the bin packing problem. Its proven upper bound is also an advantage, especially in fields where reliability is critical, such as in real-time scheduling. BFD sorts the objects in decreasing size, then places each object in the first bin into which it fits. The pseudocode can be written as follows:

```
Sort objects by decreasing size
bins = []
For object O in objects
    For bin B in bins
        If O fits in B
            Put O in B
            Break
    If O did not fit in any bin
        Add a new bin A to the end of bins
        Put O in A
```

BFD performs poorly on certain adversarial test vectors. These are coined the “⅔ Case” in [1], although what the authors describe actually results in a worst-case utilization closer to ⅘. In these adversarial cases, BFD is supplied with objects whose sizes fall close to ¼ the size of the bins, with some variation. BFD will first place all the objects larger than ¼, which will result in many bins filled with three objects too large to fit a fourth. Figure 1 shows an example of such a suboptimal packing, using four bins of size 10.
Figure 1: BFD’s solution to the “⅔ Case”. The number in each object indicate its size.

By comparison, an optimal packing for these objects uses only three bins, as shown in figure 2.

The randomized algorithm we worked with is WalkPack. It is described in [1] with the goal of solving BFD’s weakness to adversarial inputs. The algorithm employs randomization with repeated trials to improve on the quality of BFD in these cases. A notable first step in WalkPack is to order the bins using a seed algorithm. In [1] as well as in our own work, Next-fit is used. Next-fit produces solutions of reasonable quality in linear time. WalkPack reduces the bin count slowly, so starting with a decent packing speeds the process considerably. The basic sequential pseudocode for WalkPack is as follows. See the accompanying visuals below for a graphical example of one iteration.

Assign objects to bins using seed algorithm
Repeat for number of iterations
Pick a random bin S
For each object O in S

(fig. 3)  (fig. 4)  (fig. 5)
Pick a random bin $D \neq S$
Move $O$ to $D$
Destroy the now-empty $S$
For each bin $A$ with occupancy > bin capacity  
Create a new bin $B$
While $A$.occupancy > bin capacity
    Pick a random object $P$ in $A$
    Move $P$ to $B$

Figure 3: The objects after running Next-fit to seed the bins. The numbers in each object correspond to their index in the input.

Figure 4: The selected bin $S$.

Figure 5: Bin $S$ is emptied and deleted, its objects distributed randomly.
To achieve reasonable quality, heuristics must be added to the selection of bins $S$ and $D$. When selecting $S$, the bin to be destroyed, we must favor bins with more empty space to avoid destroying well-filled bins (see figure 7). Likewise, $D$ should be chosen in a way that favors bins with full space so as to create better-filled bins (see figure 8). Care must be taken when choosing $D$ not to pick a bin that cannot fit $O$, unless $O$ does not fit in any bins. This prevents needlessly creating overfull bins, which create additional bins when constrained. These heuristics improve quality significantly and make WalkPack a much more interesting problem to parallelize.
A final improvement to WalkPack that the authors of the RIT paper included is to run the algorithm repeatedly, taking the solution that uses the fewest bins once all trials have completed. This improves the quality and reliability of WalkPack by making it less likely to be caught in a low-quality local minimum.

COMPUTATION & PARALLELISM

Recall that for the Best-Fit Decreasing algorithm, the objects are sorted in decreasing order (an important feature for the quality of BFD). It is possible to achieve appreciable speedup by parallelizing sorting, especially on a GPU. From a programmer’s perspective, it is a very easy component to parallelize, as the means for parallelizing sorting are well-established.
Unfortunately, sorting does not take a large proportion of the time spent on BFD; in “Results” we measure that <1% of compute time is spent sorting.

The fact that the objects are sorted creates a dependency between each object placed into bins, so the outer loop cannot easily be parallelized. However, the most computationally-intensive part of each iteration is searching for the bin to place each object into. This process is approximately linear in the current number of bins and is repeated for each object. Fortunately, it presents some opportunities for parallelism. We can parallelize the search for a destination bin across many threads. We detail the approach we took for parallelizing this part of the algorithm below (see “Approach”), but we can see that this step requires a reduction across the bins, essentially decreasing the time to select a destination bin from linear to logarithmic in the number of bins. It is worth noting that in the sequential algorithm, we stop iterating through the bins once we find one that fits the target object. If we perform this search in parallel, we actually increase the total work performed because we must consider every bin. We still see an appreciable speedup from parallelizing this section (see “Results”).

For WalkPack, the most obvious form of parallelism is across different trials. This is how the algorithm is parallelized in the RIT paper. This approach requires a reduction across the trials in the end to determine the optimal solution, which is logarithmic in the number of trials. Because the nature of the problem is very divergent, there are limited opportunities for parallelization on a GPU. In computing the CDFs for the two heuristics, we find an appreciable level of parallelism. The execution is very regular, and with a large number of bins it is possible to assign some work to many threads in a block. During the deletion of bins, redistribution of objects, and
constraining of overfull bins, there is some opportunity for parallelization. However, these are very data-dependent tasks and highly divergent. Furthermore, these operations cannot benefit much from parallelization because the problem sizes are necessarily small. WalkPack is designed to work where BFD does not, and BFD is very effective when many objects fit in each bin (think filling a box with sand vs. beach balls—which fills the box more fully?). As such, WalkPack is only a reasonable choice of algorithm when the average number of objects per bin is low, which unfortunately limits the operations that are practical to parallelize.

**IMPLEMENTATION**

WalkPack and BFD use the same data structure to represent objects and bins. For ease of use, each test vector (bin size, number of objects, and list of object sizes) is represented by a JSON file, which we parse using the jsoncpp library. The output (number of bins used and assignment of objects to bins) is likewise converted to JSON. During program execution, objects are represented by their size and contained in a shared array. Each bin is represented as a vector of these objects, as well as metadata such as the total occupancy of the bin. We have an array of the bins currently used. For BFD, this is shared among all threads, while for WalkPack each trial has its own copy which is shared between the threads in this trial.

The data structures mentioned above are fairly mundane. We had some troubles with vectors (described in the “Difficulties” section below), but on the whole the bins and objects are not the interesting parts of our work. As mentioned above, the heuristics to WalkPack proved more challenging to implement and to parallelize. We accurately weight our random choices of bins by the amount of empty or full space in the bins, as appropriate for the heuristic. To accomplish this
weighting, we generate a table of the cumulative distribution function (CDF) that we wish to model. Based on the number of bins and the combined size of the objects, we know the total amounts of both empty and full space across all bins. For each bin, we can compute the probability that this bin is selected under the random distribution we wish to model. By prefix-summing the probabilities associated with each bin, we generate the CDF for the random function we wish to model. To sample this distribution, we select a random value between 0.0 and 1.0 and perform a binary search on the CDF to find the matching index.

**APPROACH**

We started by implementing the framework described in our reference paper in single-threaded C++. Once we had this working with acceptable quality, we moved on to a CUDA implementation of this highly dynamic code. This is where we spent much of our time, dealing with the difficulties of writing this kind of code in CUDA (for the GTX 1080) as we went.

For WalkPack, after writing the sequential code, we knew that there wouldn’t be much room for parallelization directly, due to the intense data dependencies involved, so we decided to do multiple searches in parallel over different thread blocks and do a reduction to get the best result back. We were only able to find minor optimizations in execution time from increasing thread counts within each search.

As mentioned above in “Computation and Parallelism”, BFD benefits from parallelizing the search for a destination bin for each object. To perform this search in parallel, we have two phases. In the first, we do a simple tabulation: Each index is mapped to itself if the bin at that index can fit the target object, otherwise it is mapped to INT_MAX. We perform a parallel
reduction on the resulting array of indices to find the lowest index that fits the target bin. We achieve this parallelism by mapping the problem onto a single block with many threads (see “Results” below for further details). Thrust provides the functionality to do both of these operations, but as discussed below in “Difficulties”, we achieved better performance on the device by home-brewing our own tabulation and reduction.

RESULTS

Using the RIT paper as a starting point, we looked primarily at the quality of our solutions until we were able to match their results, where quality was measured by comparing against BFD’s fit rate for a given test input. Although measuring against our own code was less than ideal, we had no reference code, and BFD seemed to give fairly comparable results to those described in said paper. In particular, we noticed that the aforementioned paper identified the so-called 2/3 case as the most relevant test case for deciding the utility of WalkPack over BFD: This was confirmed by our test data, which showed that WalkPack failed to produce superior results to BFD in most other cases than the 2/3 case. In this special case, we found that WalkPack could pack 1000 objects into 266 bins vs. 280 for BFD, which is a significant improvement (A loose lower bound for the optimal bin number for this test case was 250 bins). In less favorable cases, like a uniform 10,000 object input, WalkPack took twice as long (20 seconds) to achieve a 10% worse solution than BFD did (over 5,000 steps). While this result was not great, it was certainly better than that of the reference paper, which achieved comparable quality metrics between the two algorithm only after significantly longer runtimes (orders of magnitude better relative performance here, see fig. 4 in the reference paper). In terms of time taken to achieve acceptable quality for increasingly larger inputs, WalkPack’s runtime ramped up much faster in comparison to that of
BFD. We think this result is expected: As the input gets larger, it takes more passes to achieve the same packing ratio, and we know that the variance of WalkPack drops very slowly, so while more passes always run in the same time, there are diminishing returns on quality w.r.t. increasing numbers of passes. That said, we struggled to parallelize the WalkPack algorithm, encountering serious problems with divergent execution and data dependencies along the way. We were unable to find many regular access patterns to improve locality, limiting the utility of shared CUDA memory that we enjoyed in assignment 2. While it’s hard to say definitively which platform would be optimal for this problem, we found that bin packing was not a GPU-friendly problem, and would likely be better suited to a group of CPU cores like a Phi.
BFD certainly did not achieve linear speedup, but it did get some benefit out of higher thread counts, thanks to our home-brewed parallel search implementations.
Here, Walk Pack’s parallel reduction led to sublinear cost associated with running more trials, but only resulted in a slight increase in quality (560 bins at 1 thread compared with 558 bins at 64 threads), as the number of passes over this run was fairly low (400 passes).

In BFD, we measured a breakdown between the different components of our code. See “Approach” above for more details of the function of each of these sections. We measured this breakdown using the Thrust library calls for sort, tabulate, and reduce on a 10,000 object input vector using clock64():

- 0.5% Sorting time
- 80% Tabulation time
- 14.4% Reduction time

**DIFFICULTIES**

The biggest letdown we were faced with was the highly limited utility of the thrust library. Thrust class methods can often not be called on device, forcing us to cook up our own alternatives. In particular, this meant writing dynamic data structures like UBA’s on device—our inexperienece with C++ made this nontrivial. Although we were able to make use of Thrust library calls like upper_bound, transform, reduce, tabulate, etc al., we were not able to recover a measure performance benefit from having used them. In particular, we found that switching between Thrust’s cuda::par and seq execution policies had no significant effect on runtime when called from device code. Both policies worked as described, except they did not affect runtime; On a 10,000 object input vector, the BFD kernel took 7.5 seconds under both execution policies. Unfortunately, although we had planned to do some programming for computer club’s
Componium machine (featuring dual Titan X’s), the machine was unexpectedly taken out of state for a demo at an inopportune time.

**REFERENCES**


**LIST OF WORK**

All code was written via pair programming except for a few solo debugging sessions. Intermediate/Final reports and other miscellaneous work was done together as well. We see no reason not to award 50-50 credit.