Chapter 34
GPU Heuristic Search

► Part I. Preliminaries
► Part II. Tightly Coupled Multicore
► Part III. Loosely Coupled Cluster
▼ Part IV. GPU Acceleration
    Chapter 29. GPU Massively Parallel
    Chapter 30. GPU Parallel Reduction
    Chapter 31. Multi-GPU Programming
    Chapter 32. GPU Sequential Dependencies
    Chapter 33. Objects on the GPU
    Chapter 34. GPU Heuristic Search
► Part V. Big Data
► Appendices
Imagine you’re a burglar. You’ve broken into a house and found ten items ready to steal. Each item weighs a certain number of pounds, and each item will fetch a certain number of dollars when you go to fence it:

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight (lb)</th>
<th>Value ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stereo</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>Ashtray</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>TV</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Vase</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Book</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>Teakettle</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>Cellphone</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Vinyl LP</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Laptop</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Paperweight</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

You have a knapsack in which to carry off your loot. However, your knapsack is flimsy; if you try to cram more than 10 pounds into it, the knapsack will burst. Question: Which items should you steal so as to bring in the most money without bursting the knapsack?

This is an example of the **knapsack problem**, a well-known **combinatorial optimization** problem. In a combinatorial optimization problem, you have to find some **combination** of items that **optimizes** some quantity while obeying certain **constraints**. In this case, there are \( N \) items; \( W_i \) is the weight of item \( i \); \( V_i \) is the value of item \( i \); \( C \) is the knapsack’s capacity; the knapsack problem is to find a set of items such that \( \sum W_i \leq C \) and \( \sum V_i \) is maximized.

For the above set of items, you should steal the book, the cellphone, and the paperweight. Together these items weigh 10 pounds, so they (just) fit in your knapsack, and you can fence them for \$17.

Burglars aren’t the only ones interested in solving knapsack problems. Many practical, real-world applications can also be expressed and solved as knapsack problems. Packing goods into a container for shipping, deciding the best way to cut pieces of various sizes out of a sheet of wood or metal, and picking a portfolio of stocks so as to get the largest possible dividend return from a fixed amount of capital are just a few examples.

Many algorithms for solving the knapsack problem are known. An **exhaustive search**, which examines every possible subset of the items, is guaranteed to find an optimal solution. But with \( N \) items, there are \( 2^N \) subsets, and the running time to go through all of them quickly becomes impractical as \( N \) scales up. A **branch-and-bound search** is similar to an exhaustive search, except it considers only those subsets whose total weight does not exceed the capacity. While this does reduce the running time, the number of such subsets still increases exponentially as \( N \) increases. There is a **dynamic programming** algorithm for finding the solution, but it uses a matrix with \( N \cdot C \) elements.
Solving a high-precision knapsack problem—one where the weights, values, and capacity are large numbers (64-bit integers, say)—would require an impractical amount of storage for the dynamic programming matrix, not to mention an impractical amount of time to fill in all the matrix elements. Other published knapsack problem solving programs fail to find the solution for high-precision problems.* Bottom line, none of these approaches can scale up to handle large, high-precision knapsack problem instances.

Heuristic search to the rescue! In this chapter I’ll develop a massively parallel randomized approximation (MPRA) program to solve large, high-precision knapsack problems. Like previous MPRA programs we’ve studied, the MPRA knapsack program will generate a large number of random candidate solutions and report the best one it found. Here, “best” means $\sum W_i \leq C$ and $\sum V_i$ is a maximum. I’ll design the program to run on the GPU, to take advantage of the GPU’s large number of cores. I’ll also design the program to utilize all the GPUs on the node, in case the node has more than one GPU.

I need a heuristic for generating good candidate solutions. In the minimum vertex cover programs in Chapters 16 and 26, I generated each candidate solution starting from scratch. For the knapsack program, I’ll take a different approach. I’ll begin with an empty subset of items as the initial candidate solution. To get the next candidate, rather than starting from scratch, I’ll just make a small random change to the previous candidate. This is called a stochastic local search—“local” because each candidate is close to the previous candidate, “stochastic” because the small changes are chosen at random. Then I’ll have the program do multiple independent stochastic local searches in a massively parallel fashion and report the best candidate found overall. This is called a massively parallel stochastic local search (MPSLS) program.

Here’s how I’ll generate the next candidate solution from the current one. If the total weight of the items in the current subset is less than the capacity, I’ll choose a random item not in the subset and add that item to the subset. Otherwise, I’ll choose a random item in the subset and remove that item from the subset. This should focus the program’s search on the subsets whose total weight is near the capacity—the region of the search space where the optimum solution should be located. However, like any approximation algorithm, this does not guarantee that the program will find the true best solution. Still, the program should find a solution close to the true optimum solution; and this might be good enough for a practical application.

I need a data structure for a solution to the knapsack problem; that is, for a subset of the items. This is a subset of the integers 0 through $N – 1$. We’ve already seen the bitset data structure. For the knapsack problem, however, I need an efficient way to choose a random item in the subset or not in the subset. A bitset data structure doesn’t let me do that efficiently. Instead, I’ll use

---

an \((N + 1)\)-element array. Index 0 will hold the size of the subset, \(S\). Indexes 1 through \(S\) will hold the items in the subset, in no particular order. Indexes \(S + 1\) through \(N\) will hold the items not in the subset, in no particular order. With \(N = 10\) possible items, here is what an initially empty subset looks like:

<table>
<thead>
<tr>
<th>(S)</th>
<th>Not in the subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1 2 3 4 5 6 7 8 9</td>
</tr>
</tbody>
</table>

And here is what the subset \{4, 6, 9\} looks like; remember, the order of the items is not important:

<table>
<thead>
<tr>
<th>(S)</th>
<th>In the subset</th>
<th>Not in the subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6 4 9 8 1 7 0 5 2 3</td>
<td></td>
</tr>
</tbody>
</table>

To add a random element to the subset, pick a random index \(j\) in the range \(S + 1\) through \(N\); swap the elements at indexes \(S + 1\) and \(j\); and increment \(S\), the element at index 0. Suppose the random index is \(j = 7\); then the subset becomes \{0, 4, 6, 9\}:

<table>
<thead>
<tr>
<th>(S)</th>
<th>In the subset</th>
<th>Not in the subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6 4 9 0 1 7 8 5 2 3</td>
<td></td>
</tr>
</tbody>
</table>

To remove a random element from the subset, pick a random index \(j\) in the range 1 through \(S\); swap the elements at indexes \(S\) and \(j\); and decrement \(S\), the element at index 0. Suppose the random index is \(j = 3\); then the subset becomes \{0, 4, 6\}:

<table>
<thead>
<tr>
<th>(S)</th>
<th>In the subset</th>
<th>Not in the subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6 4 0 9 1 7 8 5 2 3</td>
<td></td>
</tr>
</tbody>
</table>

Based on these considerations, I can start writing the code for the GPU accelerated MPSLS knapsack problem solver, WalkSackGpu. It is named after Walksat,* a stochastic local search program that solves the Boolean satisfiability, or SAT, problem.

Listing 34.1 defines the \texttt{wv\_t} structure, with fields for the weight and value of one item. The fields are declared as 64-bit integers, to let WalkSackGpu solve high precision knapsack problems. Several “methods” (functions, really) are also defined: to initialize a \texttt{wv\_t} structure; to add one \texttt{wv\_t} structure to another; to subtract one \texttt{wv\_t} structure from another; and to decide which of two \texttt{wv\_t} structures is better with respect to a given knapsack capacity. If both weights are at or below the capacity, the structure with the larger value is better; otherwise, the structure with the smaller weight is bet-

* \[http://www.cs.rochester.edu/u/kautz/walksat/\]
// Weight and value of one item. Also used for total weight and
// total value of the knapsack.
typedef struct
{
  unsigned long long int weight;
  unsigned long long int value;
} wv_t;

// Initialize weight/value a to 0/0.
__device__ void wv_init
    (wv_t *a)
{ a->weight = 0;
  a->value = 0;
}

// Add weight/value b to weight/value a.
__device__ void wv_add
    (wv_t *a,
     wv_t *b)
{ a->weight += b->weight;
  a->value += b->value;
}

// Subtract weight/value b from weight/value a.
__device__ void wv_subtract
    (wv_t *a,
     wv_t *b)
{ a->weight -= b->weight;
  a->value -= b->value;
}

// Determine which weight/value is better, a or b, with respect to
// capacity C. Returns 1 if a is better, 0 if b is better.
__device__ int wv_isBetterThan
    (wv_t *a,
     wv_t *b,
     unsigned long long int C)
{ if (a->weight > C && b->weight > C)
    return a->weight < b->weight;
  else if (a->weight > C)
    return 0;
  else if (b->weight > C)
    return 1;
  else
    return a->value > b->value;
}

Listing 34.1. WV.cu
ter. I’ll use the \texttt{wv\_isBetterThan()} function to decide which solution is best among the many candidate solutions the program generates.

Listing 34.2 is the code for the GPU kernel. The kernel will be executed by a one-dimensional grid of one-dimensional blocks. The number of blocks in the grid \( NB \) and the number of threads in each block \( NT \) will be established by the Java main program. The total number of threads in the grid is \( NG = NB \cdot NT \). Each thread in the grid has a unique rank in the range 0 through \( NG - 1 \).

Lines 1–2 pull in structures and functions for a pseudorandom number generator \texttt{prng\_t} and a weight-value structure \texttt{wv\_t}. The \texttt{shrRank} array (line 8) is located in shared memory in each block of the grid; this will be used in a parallel reduction to find the rank of the thread that found the best solution in the block. The \texttt{bobRank} variable (line 5) is located in global memory; when the kernel is finished, \texttt{bobRank} will hold the rank of the thread that found the best-of-best solution in the entire grid.

The kernel function, \texttt{walk()}, begins on line 22. Its arguments, which are set up by the Java main program, are

- \( c \), the knapsack capacity.
- \( N \), the number of items.
- \( \text{item\_wv} \), a pointer to an array of \( N \) elements of type \texttt{wv\_t} in global memory. Each element gives the weight and value of one item.
- \( \text{curr\_soln} \), a pointer to a matrix with \( NG \) rows and \( N + 1 \) columns in global memory. The row at index \( r \) contains the current solution generated by thread rank \( r \). The solution uses the array-based data structure described earlier, consisting of the number of items in the subset, the indexes of the items in the subset, and the indexes of the the items not in the subset.
- \( \text{best\_wv} \), a pointer to an array of \( NG \) elements of type \texttt{wv\_t} in global memory. The element at index \( r \) gives the total weight and total value of the best solution found by thread rank \( r \).
- \( \text{best\_soln} \), a pointer to a matrix with \( NG \) rows and \( N + 1 \) columns in global memory. The row at index \( r \) contains the best solution found by thread rank \( r \), using the data structure described earlier.
- \( \text{best\_step} \), a pointer to an array of \( NG \) elements in global memory. The element at index \( r \) gives the step number at which thread rank \( r \) found its best solution.
- \( \text{seed} \), a seed for the PRNG.
- \( \text{steps} \), the total number of steps to perform. Each step comprises generating and evaluating the next candidate solution.

Each thread executing the kernel function begins with initializations. The thread computes its own rank (line 42); obtains a pointer to its own current
#include "Prng.cu"
#include "WV.cu"

// Thread rank that found the best-of-best solution.
__device__ int bobRank;

// For shared memory reduction of thread ranks.
__shared__ int shrRank[1024];

// Swap elements a and b in array x.
__device__ void swap(int *x, int a, int b)
{
    int t = x[a];
    x[a] = x[b];
    x[b] = t;
}

// Kernel function to solve a knapsack problem.
extern "C" __global__ void walk
(unsigned long long int C,
int N,
wv_t *item_wv,
wv_t *best_wv,
unsigned long long int *best_step,
unsigned long long int seed,
unsigned long long int steps)
{
    int r, i, j, M, oldRank, newRank;
    unsigned long long int s;
    wv_t curr_wv_r;
    wv_t best_wv_r;
    int *curr_soln_r;
    int *best_soln_r;
    prng_t prng;

    // Determine this thread's rank in the grid.
    r = blockIdx.x*blockDim.x + threadIdx.x;
    curr_soln_r = curr_soln[r]; // Items in current solution
    best_soln_r = best_soln[r]; // Items in best solution

    // Initialize this thread's data.
    wv_init (&curr_wv_r); // Weight/value of current solution
    wv_init (&best_wv_r); // Weight/value of best solution
    M = 0; // No. of items in current solution
    for (i = 1; i <= N; ++i) // Items in current solution
        curr_soln_r[i] = i - 1;
    prng_setSeed (&prng, seed + r);

Listing 34.2. WalkSackGpu.cu (part 1)
solution, a row in the \texttt{curr\_soln} matrix (line 45); obtains a pointer to its own best solution, a row in the \texttt{best\_soln} matrix (line 46); initializes the current and best solutions’ weights and values to 0 (lines 49–50); initializes the current solution itself to an empty subset (lines 51–53); and initializes its PRNG with a unique seed (line 54).

The thread performs the specified number of steps (line 57). At each step, the thread either adds a random item to the thread’s current subset of items (lines 61–68) or removes a random item (lines 72–79), using the algorithms described previously. When an item is added to or removed from the subset, the item’s weight and value are added to or subtracted from the current solution’s total weight and value (lines 66, 77). The thread compares the new current solution’s weight and value to the previous best solution’s weight and value (line 83). If the current solution is better, the thread records the current solution as the best solution (lines 85–88). The thread also records the step number at which this best solution was encountered (line 89); at the end of the run, the Java main program will report this for informational purposes.

After finishing all the steps, the thread stores the weight and value of the best solution it found in the \texttt{best\_wv} array (line 94). But before proceeding to the reduction phase of the kernel, the thread must wait until all threads in the grid have done the same. The thread does so by calling the special \texttt{__threadfence()} CUDA function (line 95). That function does not return until all pending memory transactions in all threads of the grid have completed. If this step is omitted, the contents of the \texttt{best\_wv} array will not be up-to-date, and the reduction will compute the wrong answer.

Like the reduction in the PiGpu kernel, the reduction in the WalkSack-Gpu kernel proceeds in two phases. First, in each block, the threads of the block determine the rank of the thread that found the best solution in that block. This is done by a parallel reduction tree using the \texttt{shrRank} array in shared memory (lines 99–113). The quantities being reduced together are thread ranks. Rather than summation, the reduction operation on two thread ranks is to compare the weights and values of the best solutions found by those thread ranks and to keep whichever thread rank corresponds to the better solution. At the end of this phase, \texttt{shrRank}[0] contains the thread rank of the best solution in the block.

For the second phase of reduction, one thread in each block does an atomic compare and swap operation involving the \texttt{bobRank} variable in global memory (lines 115–125). If this block’s best solution is better than thread rank \texttt{bobRank}’s best solution, \texttt{bobRank} is atomically set to the rank of the thread with the best solution in this block. Once all blocks have finished executing the kernel function, \texttt{bobRank} contains the rank of the best-of-best solution over all the threads in the grid; the \texttt{best\_wv} element at that index contains the weight and value of the best-of-best solution; and the \texttt{best\_soln} row at that index contains the subset of items in the best-of-best solution. The
for (s = 0; s < steps; ++ s)
{
    // If current solution's weight < knapsack capacity, add a
    // random item.
    if (curr_wv_r.weight < C)
    {
        i = M + 1;
        j = M + 1 + prng_nextInt (&prng, N - M);
        swap (curr_soln_r, i, j);
        wv_add (&curr_wv_r, &item_wv[curr_soln_r[i]]);
        ++ M;
    }

    // If current solution's weight >= knapsack capacity, subtract
    // a random item.
    else
    {
        i = M;
        j = 1 + prng_nextInt (&prng, M);
        swap (curr_soln_r, i, j);
        wv_subtract (&curr_wv_r, &item_wv[curr_soln_r[i]]);
        -- M;
    }

    // If new current solution is better than best solution,
    // record new best solution.
    if (wv_isBetterThan (&curr_wv_r, &best_wv_r, C))
    {
        best_wv_r = curr_wv_r;
        best_soln_r[0] = M;
        for (i = 1; i <= M; ++ i)
            best_soln_r[i] = curr_soln_r[i];
        best_step[r] = s + 1;
    }
}

// Record weight/value of best solution.
best_wv[r] = best_wv_r;
__threadfence();

// Shared memory reduction to determine thread rank with best-of-
// best solution in this block.
i = threadIdx.x;
shrRank[i] = r;
__syncthreads();
j = 1;
while (j < blockDim.x) j <<= 1;
j >>= 1;
while (j != 0)
{
    if (i < j && i + j < blockDim.x &&
        wv_isBetterThan (&best_wv[shrRank[i+j]],
                         &best_wv[shrRank[i]], C))
        shrRank[i] = shrRank[i+j];
    __syncthreads();
j >>= 1;
}

Listing 34.2. WalkSackGpu.cu (part 2)
Java main program will then download these kernel outputs from global memory.

Turning to the Java code, class WV (Listing 34.3) is the Java equivalent of the C wv_t structure, encapsulating a weight and a value along with various constructors and methods. In addition, class WV extends the Struct base class (line 5) and implements the sizeof(), toStruct(), and fromStruct() methods (lines 76–97). This lets the Java main program upload WV objects from the CPU to wv_t structures on the GPU and vice versa.

The Java main program is invoked with these command line arguments:

```
$ java pj2 [gpus=NA] edu.rit.gpu.example.WalkSackGpu [-p] "constructor" seed reps steps [NB [NT]]
```

- **NA** is the number of GPU accelerators to use. If omitted, the default is to use all the GPUs on the node.
- If the -p flag is present, the program will print the weights, values, and capacity of the knapsack problem to be solved.
- **constructor** is a “constructor expression” for an object that defines the knapsack problem to be solved. (More on this below.)
- **seed** is a seed for the PRNGs in the program.
- **reps** is the number of repetitions to do.
- **steps** is the number of steps per repetition to do.
- **NB** is the number of blocks in the kernel grid. If omitted, the default is one block for each multiprocessor on the GPU.
- **NT** is the number of threads per block in the kernel grid. If omitted, the default is 256.

The program performs **reps** repetitions. In each repetition, the program runs an instance of the GPU kernel on each GPU. Each kernel generates a series of **steps** candidate solutions, where a random item is added to or removed from the subset at each step. By passing in a different PRNG seed argument each time the program runs the kernel, each thread in each kernel invocation generates a different sequence of random candidates, so as to more thoroughly explore the space of solutions. By specifying **reps** and **steps**, I can control how many candidate solutions the program examines, which in turn determines how long the program runs.

I need a way to for the program to obtain the knapsack problem parameters—the number of items, the weight and value of each item, and the capacity. To do so, I’ll do the same as I did for the programs that solved graph problems in previous chapters. I’ll define a Java interface for an object that encapsulates the knapsack problem parameters. I’ll write classes that implement this interface, which define knapsack problems of various kinds. The program will construct an instance of such a class, by invoking the constructor expression given on the command line. Then the program will query the
Chapter 34. GPU Heuristic Search

// Global memory reduction to determine thread rank with best-of-
// best solution across all blocks.
if (i == 0)
  do
    {
      oldRank = bobRank;
      newRank =
      oldRank == -1 ||
        wv_isBetterThan (&best_wv[shrRank[0]],
          &best_wv[oldRank], C) ?
          shrRank[0] : oldRank;
      }
    while (atomicCAS (&bobRank, oldRank, newRank) != oldRank);

Listing 34.2. WalkSackGpu.cu (part 3)

package edu.rit.gpu.example;
import edu.rit.gpu.Struct;
import java.nio.ByteBuffer;
public class WV
  extends Struct
  implements Cloneable
  {
    public long weight;
    public long value;
    // Construct a new weight/value object.
    public WV()
      {
    }
    // Construct a new weight/value object with the given weight and
    // value.
    public WV
      (long weight,
        long value)
      {
        this.weight = weight;
        this.value = value;
      }
    // Construct a new weight/value object that is a copy of the
    // given weight/value object.
    public WV
      (WV wv)
      {
        copy (wv);
      }
    // Set this weight/value object to a copy of the given
    // weight/value object.
    public void copy
      (WV wv)
      {
        this.weight = wv.weight;
      }

Listing 34.3. WV.java (part 1)
constructed object to get the knapsack problem parameters. Let’s look at the details.

Interface KnapsackProblem (Listing 34.4) specifies the interface for a knapsack problem object. It is self-explanatory.

Class KnapsackSC (Listing 34.5) defines a class for a “strongly correlated” knapsack problem, one where each item’s value is the same as the item’s weight plus or minus some delta. The class implements the KnapsackProblem interface (line 5). The constructor arguments (line 15) specify the capacity, the number of items, the maximum weight of the items, the item value delta, and a seed for a PRNG; these are stored in private fields. The capacity() method (line 40) and the itemCount() method (line 46) merely return the corresponding parameters. Each time the next() method (line 52) is called, it generates a random item weight in the range 1 through the maximum weight; sets the item value to the item weight plus the delta; and returns a new WV object containing the weight and value. By querying these methods, the main program can get all the parameters of the knapsack problem to be solved.

The WalkSackGpu program needs a way to create an instance of a class that implements interface KnapsackProblem, such as class KnapsackSC. Like the graph solving programs, the WalkSackGpu program’s first command line argument is a constructor expression for an instance of a class that implements interface KnapsackProblem, like this:

```
$ java pj2 edu.rit.gpu.example.WalkSackGpu \
  "edu.rit.gpu.example.KnapsackSC(30,20,10,5,142857)"
```

The program passes the constructor expression to the Instance.newInstance() method in the Parallel Java 2 Library, which returns a reference to the knapsack problem object.

Using this technique, the same WalkSackGpu program can solve any kind of knapsack problem. Simply define a class that implements the KnapsackProblem interface, with a constructor that causes the class to return the desired knapsack problem parameters. Then specify the appropriate constructor expression on the WalkSackGpu command line.

The Java main program, class WalkSackGpu (Listing 34.6), begins by declaring the WalkKernel kernel interface with the walk() kernel method (lines 20–34). Next comes class SolutionVbl (lines 37–88), which encapsulates a solution to the knapsack problem. The solution consists of the capacity; the total weight and value of the items in the solution; and the subset of items in the solution, using the array-based data structure described earlier. The class has methods for setting the solution. The class is also a reduction variable (line 38), with the clone(), set(), and reduce() methods. The reduce() method compares the total weights and values of two solutions and keeps whichever solution has the better weight and value—that is, total
this.value = wv.value;
}

// Create a clone of this weight/value object.
public Object clone()
{
    try
    {
        return super.clone();
    }
    catch (CloneNotSupportedException exc)
    {
        // Shouldn't happen.
        throw new IllegalStateException (exc);
    }
}

// Determine if this weight/value object is better than the given
// weight/value object with respect to the given capacity.
public boolean isBetterThan
(WV wv,
   long C)
{
    if (this.weight > C && wv.weight > C)
        return this.weight < wv.weight;
    else if (this.weight > C)
        return false;
    else if (wv.weight > C)
        return true;
    else
        return this.value > wv.value;
}

// Returns the size in bytes of the C struct.
public static long sizeof()
{
    return 16;
}

// Write this Java object to the given byte buffer in the form of
// a C struct.
public void toStruct
(ByteBuffer buf)
{
    buf.putLong (weight);
    buf.putLong (value);
}

// Read this Java object from the given byte buffer in the form
// of a C struct.
public void fromStruct
(ByteBuffer buf)
{
    weight = buf.getLong();
    value = buf.getLong();
}

Listing 34.3. WV.java (part 2)
weight less than or equal to the capacity and total value maximized. Instances
of the SolutionVbl class will be used to determine the overall best solution,
using the standard multithreaded parallel reduction pattern.

The WalkSackGpu task’s main method (line 109), although somewhat
lengthy, is straightforward. After parsing the command line arguments and
determining the number of GPU accelerators on the node, the task uses the
\texttt{Instance.newInstance()} method to create a knapsack problem instance as
specified on the command line, extracts the knapsack problem parameters
from that instance, and (optionally) prints the parameters.

The task uses the multiple-GPU design pattern, like the PiGpu2 program
in Chapter 31. The task creates a global SolutionVbl reduction variable. The
task sets up a parallel thread team with one thread for each GPU accelerator.
Executing the parallel section’s \texttt{run()} method, each team thread creates a lo-
cal SolutionVbl reduction variable linked to the global reduction variable;
sets up the GPU object and the kernel module; allocates storage for the ker-
nel data structures in the CPU and in the GPU’s global memory; uploads the
item weights and values to the GPU; and configures the kernel with the de-
sired number of blocks and threads. The team thread then performs the speci-
fied number of repetitions. In each repetition, the team thread calls the kernel
function, which performs the specified number of steps on the GPU; down-
loads and prints the best solution the kernel found; and keeps track of the
best solution found across all the repetitions. The printout also includes sta-
tistics about the step numbers at which each kernel thread found its best solu-
tion, for informational purposes. When all the repetitions have finished, the
team thread stores the best solution in the local reduction variable. When the
whole parallel do statement has finished, the team threads’ best solutions are
automatically reduced together and the best of the best is stored in the global
reduction variable, which the task prints as the overall solution.

I want to know how the WalkSackGpu program fares at solving knapsack
problem instances when compared to programs that use other approaches. I
wrote GPU parallel programs that use an exhaustive search algorithm (class
\texttt{edu.rit.gpu.example.KnapsackExhGpu}) and a branch-and-bound search algo-

rithm (class \texttt{edu.rit.gpu.example.KnapsackBnbGpu}). Like WalkSackGpu,
these programs are designed to utilize all the GPU accelerators on the node.
(Space limitations prevent me from listing or describing these programs
here.)

I’m interested in comparing the running times of these programs on the
same knapsack problem instance. Also, while KnapsackExhGpu and Knaps-
ackBnbGpu are guaranteed to find the true optimum solution, WalkSackGpu
is not. I’d like to know how close WalkSackGpu’s solution comes to the true
optimum solution. I refer to this as the \textit{quality} of WalkSackGpu’s approxi-
mate solution.

To determine the solution quality, I need to know the true optimum solu-
package edu.rit.gpu.example;
public interface KnapsackProblem
{
    // Get the knapsack capacity.
    public long capacity();

    // Get the number of items, N.
    public int itemCount();

    // Get the weight and value of the next item. This method must be
    // called N times to get all the items. Each method call returns
    // a new WV object.
    public WV next();
}

Listing 34.4. KnapsackProblem.java

package edu.rit.gpu.example;
import edu.rit.util.Random;
import java.util.NoSuchElementException;
public class KnapsackSC
implements KnapsackProblem
{
    private long C;
    private int N;
    private long maxW;
    private long deltaV;
    private Random prng;
    private int count;

    // Construct a new strongly correlated knapsack problem.
    public KnapsackSC
(long C,
    int N,
    long maxW,
    long deltaV,
    long seed)
    {
        if (C < 1)
            throw new IllegalArgumentException (String.format
("KnapsackSC(): C = %d illegal", C));
        if (N < 1)
            throw new IllegalArgumentException (String.format
("KnapsackSC(): N = %d illegal", N));
        if (maxW < 1)
            throw new IllegalArgumentException (String.format
("KnapsackSC(): maxW = %d illegal", maxW));
        this.C = C;
        this.N = N;
        this.maxW = maxW;
        this.deltaV = deltaV;
        this.prng = new Random (seed);
        this.count = 0;
    }

Listing 34.5. KnapsackSC.java (part 1)
tion. For an arbitrary knapsack problem, there’s no way to know this without running a program that guarantees to find the exact solution, which might take a very long time. But for some kinds of knapsack problems, it’s possible to construct instances where the exact solution is known ahead of time.

One such class of problems is the *subset sum problem*, an NP decision problem. Given a set of integers and a value \( S \), the subset sum problem asks the question: Is there a subset of those integers whose sum is \( S \)? The subset sum problem can be recast as a knapsack problem in which each item’s weight is one of the original set’s integers, each item’s value is the same as its weight, and the capacity is \( S \). If the knapsack problem’s solution has a total value equal to \( S \), then the answer to the original subset sum problem is yes, and the items in the solution give the integers whose sum is \( S \). If the knapsack problem’s solution has a total value less than \( S \), then the answer to the original subset sum problem is no.

I can construct a knapsack problem instance with a known exact solution by picking a given number of items, choosing the item weights, setting the item values equal to the weights, and setting the capacity to the sum of the values of a selected subset of the items. Then the exact solution’s total value will be equal to that capacity. I wrote class `edu.rit.gpu.example.SubsetSum` to generate such a knapsack problem instance.

I did some experiments on the RIT CS kraken machine with four GPU accelerators. I generated a subset sum problem with 40 items, where the item weights and values were chosen uniformly at random in the range 1 through 10 billion, and where a subset of the items had sum \( S = 52832185565 \). The `KnapsackExhGpu` exhaustive search program found the exact solution in 1176720 milliseconds (19.6 minutes). The `KnapsackBnbGpu` branch-and-bound search program found the exact solution in 5521 milliseconds; the program took less time because it examined only the subsets whose total value did not exceed the capacity, rather than all possible subsets. For the `WalkSackGpu` program, the running time and the solution quality depend on the number of repetitions and the number of steps per repetition performed. I ran the program with `reps = 10`, so each GPU did ten repetitions, and with four GPUs the whole program did 40 repetitions. I got these results:

<table>
<thead>
<tr>
<th>Steps</th>
<th>Time</th>
<th>Observed</th>
<th>Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1891</td>
<td>52832185347</td>
<td>1 – 4.13×10⁻⁹</td>
</tr>
<tr>
<td>200</td>
<td>1936</td>
<td>52832185471</td>
<td>1 – 1.78×10⁻⁹</td>
</tr>
<tr>
<td>500</td>
<td>1970</td>
<td>52832185471</td>
<td>1 – 1.78×10⁻⁹</td>
</tr>
<tr>
<td>1000</td>
<td>1986</td>
<td>52832185471</td>
<td>1 – 1.78×10⁻⁹</td>
</tr>
<tr>
<td>2000</td>
<td>1918</td>
<td>52832185471</td>
<td>1 – 1.78×10⁻⁹</td>
</tr>
<tr>
<td>5000</td>
<td>2160</td>
<td>52832185524</td>
<td>1 – 7.76×10⁻¹⁰</td>
</tr>
<tr>
<td>10000</td>
<td>2463</td>
<td>52832185565</td>
<td>1</td>
</tr>
<tr>
<td>20000</td>
<td>3125</td>
<td>52832185565</td>
<td>1</td>
</tr>
</tbody>
</table>
Chapter 34. GPU Heuristic Search

Listing 34.5. KnapsackSC.java (part 2)

Listing 34.6. WalkSackGpu.java (part 1)
The “Observed” column gives the total value of WalkSackGpu’s solution. The quality figure is computed by the formula

\[ \text{quality} = 1 - \frac{\text{exact} - \text{observed}}{\text{exact}}. \quad (34.1) \]

If WalkSackGpu finds the exact solution, the quality is 1. If it finds an approximate solution, the quality is less than 1. The quality is expressed in the form “1 minus something” to show how close the approximate solution is to the exact solution.

The data shows that even with only 40 repetitions and 100 steps, WalkSackGpu found an approximate solution that agrees with the exact solution to eight significant digits; and it did so nearly three times faster than branch-and-bound search and over 600 times faster than exhaustive search. As the number of steps increased, the solution quality also increased. At 10,000 steps, WalkSackGpu found the exact solution.

I tried a larger subset sum problem with 48 items, having a subset whose sum was \( S = 74393637296 \). The exhaustive search program would have taken 256 times as long as it took for the previous problem, or 3.5 days, so I didn’t bother running it. The branch-and-bound search program found the exact solution in 9060418 milliseconds (2.5 hours). I ran the WalkSackGpu program with \( \text{reps} = 10 \) as before. With 1,000 steps, the program found a solution with total value 74393637140, which agrees with the exact solution to eight significant digits. The running time was 1956 milliseconds, over 4,600 times faster than branch-and-bound search, and over 150,000 times faster than exhaustive search. With 100,000 steps, the WalkSackGpu program found the exact solution in 7748 milliseconds.

Based on these and other experiments, I believe that the MPSLS algorithm embodied in the WalkSackGpu program is a viable approach for solving high precision knapsack problems. The program is able to find approximate solutions that are extremely close to the exact solutions with a reasonable amount of computation. This is fine for a practical application, where a slight reduction in value from the true optimum solution is acceptable.

**Points to Remember**

- A combinatorial optimization problem seeks to find a combination, or subset, of a set of items that optimizes some quantity while obeying certain constraints.
- The knapsack problem is a combinatorial optimization problem that is widely used in practical applications.
- A stochastic local search algorithm visits a series of candidate solutions; each candidate is generated by making a small, randomly chosen change to the previous candidate.
long seed,
    long steps);
}

// Reduction variable class for knapsack problem solutions.
private static class SolutionVbl
    implements Vbl
{
    public long C;
    public WV wv;
    public int[] soln;

    public SolutionVbl (long C)
    {
        this.C = C;
        this.wv = new WV();
        this.soln = new int[0];
    }

    public void set (WV wv, int[] soln)
    {
        this.wv = new WV (wv);
        this.soln = (int[]) soln.clone();
    }

    public void set (SolutionVbl vbl)
    {
        this.C = vbl.C;
        this.wv = new WV (vbl.wv);
        this.soln = (int[]) vbl.soln.clone();
    }

    public Object clone()
    {
        try
        {
            SolutionVbl vbl = (SolutionVbl) super.clone();
            vbl.set (this);
            return vbl;
        }
        catch (CloneNotSupportedException exc)
        {
            throw new IllegalStateException (exc);
        }
    }

    public void set (Vbl vbl)
    {
        set ((SolutionVbl)vbl);
    }

    public void reduce (Vbl vbl)
    {
        if (((SolutionVbl)vbl).wv.isBetterThan (this.wv, this.C))
            set (vbl);
    }
}

Listing 34.6. WalkSackGpu.java (part 2)
• A massively parallel stochastic local search (MPSLS) program runs multiple independent stochastic local searches in parallel, and reports the best solution found across all the searches.
• Each independent search in an MPSLS program should visit a different randomly chosen sequence of candidates in the space of solutions.
• For a GPU parallel MPSLS program, let each GPU kernel thread do an independent search, and use parallel reduction to find the best solution across all the threads.
• An MPSLS program can typically find an approximate solution that is very close, if not identical, to the exact solution.
• Consider using the Instance.newInstance() method in the Parallel Java 2 Library to obtain an object that encapsulates a problem’s parameters.

```java
// Command line arguments.
boolean pFlag;
String constructor;
long seed;
long reps;
long steps;
int NB;
int NT;
int NA;

// Knapsack problem parameters.
long C;
int N;
WV[] wv;

// Global reduction variable for solution.
SolutionVbl solution;

// Main program.
public void main(String[] args) throws Exception {
    long t1 = System.currentTimeMillis();
    
    // Parse command line arguments.
pFlag = false;
    int argi = 0;
    while (argi < args.length && args[argi].charAt(0) == '-')
    {
        if (args[argi].equals("-p"))
        {
            pFlag = true;
            ++ argi;
        }
    }

    Listing 34.6. WalkSackGpu.java (part 3)
```
else
  usage();
}
if (argi >= args.length) usage();
constructor = args[argi++];
if (argi >= args.length) usage();
seed = Long.parseLong (args[argi++]);
if (argi >= args.length) usage();
reps = Long.parseLong (args[argi++]);
if (argi >= args.length) usage();
steps = Long.parseLong (args[argi++]);
NB = -1;
NT = -1;
if (argi < args.length)
{
  NB = Integer.parseInt (args[argi++]);
  if (argi < args.length)
  {
    NT = Integer.parseInt (args[argi++]);
    if (argi < args.length) usage();
  }
}

// Determine number of GPU accelerators to use.
NA = Gpu.allowedDeviceCount();

// Print provenance.
System.out.printf
("# java pj2 gpus=%d edu.rit.gpu.example.WalkSackGpu", NA);
if (pFlag) System.out.printf(" -p");
System.out.printf(" \"\%s\"", constructor);
System.out.printf(" \%d", seed);
System.out.printf(" \%d", reps);
System.out.printf(" \%d", steps);
if (NB != -1) System.out.printf(" \%d", NB);
if (NT != -1) System.out.printf(" \%d", NT);
System.out.println();
System.out.printf("# Started %s\n", new Date (t1));
System.out.flush();

// Set up knapsack problem.
KnapsackProblem problem = (KnapsackProblem)
  Instance.newInstance (constructor);
C = problem.capacity();
N = problem.itemCount();
wv = new WV [N];
for (int i = 0; i < N; ++ i)
  wv[i] = problem.next();
if (pFlag)
{
  System.out.printf("p zeroOne \%d \%d\%n", C, N);
  System.out.flush();
  for (int i = 0; i < N; ++ i)
  {
    System.out.printf("i \%d \%d \%d\%n", 
                      i, wv[i].weight, wv[i].value);
    System.out.flush();
  }
}

Listing 34.6. WalkSackGpu.java (part 4)
public void run() throws Exception {
    // Set up thread local reduction variable for solution.
    SolutionVbl mySolution = threadLocal (solution);

    // Set up GPU and kernel module.
    Gpu gpu = Gpu.gpu();
    Module module = gpu.getModule
        ("edu/rit/gpu/example/WalkSackGpu.cubin");
    int myNB = NB == -1 ? gpu.getMultiprocessorCount() : NB;
    int myNT = NT == -1 ? 256 : NT;
    int myNG = myNB*myNT;
    System.out.printf
        ("# GPU %d of %d: NB = %d, NT = %d, NG = %d
        
        rank() + 1, NA, myNB, myNT, myNG);
    System.out.flush();

    // Set up GPU variables.
    GpuIntVbl bobRank = module.getIntVbl ("bobRank");
    GpuStructArray<WV> item_wv =
        gpu.getStructArray (WV.class, N);
    for (int i = 0; i < N; ++ i)
        item_wv.item[i] = wv[i];
    item_wv.hostToDevice();
    GpuIntMatrix curr_soln =
        gpu.getIntMatrix (myNG, N+1, 0, 0);
    GpuStructArray<WV> best_wv =
        gpu.getStructArray (WV.class, myNG, 1);
    best_wv.item[0] = new WV();
    GpuIntMatrix best_soln =
        gpu.getIntMatrix (myNG, N+1, 1, N+1);
    GpuLongArray best_step =
        gpu.getLongArray (myNG);
    GpuIntMatrix best_soln =
        gpu.getIntMatrix (myNG, N+1, 0, 0);
    GpuStructArray<WV> best_wv =
        gpu.getStructArray (WV.class, myNG, 1);
    best_wv.item[0] = new WV();
    GpuIntMatrix best_soln =
        gpu.getIntMatrix (myNG, N+1, 1, N+1);
    GpuLongArray best_step =
        gpu.getLongArray (myNG);

    // Set up GPU kernel.
    WalkKernel wkern = module.getKernel (WalkKernel.class);
    wkern.setGridDim (myNB);
    wkern.setBlockDim (myNT);
    wkern.setCacheConfig (CacheConfig.CU_FUNC_CACHE_PREFER_L1);

    // Set up best-of-best solution across all repetitions.
    WV bob_wv = new WV();
    int[] bob_soln = new int [N+1];

    // Perform repetitions.
    for (long R = 0; R < reps; ++ R)
        {
            // Perform random walks on the GPU.
            bobRank.item = -1;
            bobRank.hostToDevice();

            Listing 34.6. WalkSackGpu.java (part 5)
wkern.walk
   (C, N, item_wv, curr_soln, best_wv, best_soln,
    best_step, rank()*(1L<<32) + seed + R*myNG,
   steps);

   // Get best solution from GPU.
   bobRank.devToHost();
   best_wv.devToHost (0, bobRank.item, 1);
   best_soln.devToHost (0, 0, bobRank.item, 0, 1, N+1);
   best_step.devToHost();

   // Print best solution.
   System.out.printf ("# GPU %d of %d, rep %d of %d%n",
   rank() + 1, NA, R + 1, reps);
   System.out.printf ("#    Best rank: %d%n",
   bobRank.item);
   System.out.printf ("#    Best weight/value: %d %d%n",
   best_wv.item[0].weight, best_wv.item[0].value);
   long bestStepBobRank = best_step.item[bobRank.item];
   Sorting.sort (best_step.item);
   System.out.printf ("#    Best step: 5%% = %d, 50%% = %d, "+
   "95%% = %d, best = %d%n",
   best_step.item[(int)(0.05*myNG)],
   best_step.item[(int)(0.50*myNG)],
   best_step.item[(int)(0.95*myNG)],
   bestStepBobRank);
   System.out.flush();

   // Update best-of-best solution.
   if (best_wv.item[0].isBetterThan (bob_wv, C))
   {
      bob_wv.copy (best_wv.item[0]);
      int k = best_soln.item[0][0];
      System.arraycopy (best_soln.item[0], 0,
         bob_soln, 0, k+1);
   }

   mySolution.set (bob_wv, bob_soln);
}

// Print best-of-best solution.
System.out.printf ("s %d %d%n",
   solution.wv.weight, solution.wv.value);
System.out.printf ("k");
int k = solution.soln[0];
Sorting.sort (solution.soln, 1, k);
for (int i = 1; i <= k; ++ i)
   System.out.printf (" %d", solution.soln[i]);
System.out.println();

// Print running time.
long t2 = System.currentTimeMillis();
System.out.printf ("# Finished %s%n", new Date (t2));

Listing 34.6. WalkSackGpu.java (part 6)
System.out.printf("# %d msec\n", t2 - t1);
}

// Print a usage message and exit.
private static void usage()
{
   System.err.println("Usage: java pj2 [gpus=<NA>] " +
   "edu.rit.gpu.example.WalkSackGpu [-p] "<constructor>" " +
   "<seed> <reps> <steps> [NB] [NT]]");
   System.err.println("<NA> = Number of GPU accelerators to " +
   "use");
   System.err.println("-p = Print knapsack problem parameters");
   System.err.println("<constructor> = KnapsackProblem " +
   "constructor expression");
   System.err.println("<seed> = Random seed");
   System.err.println("<reps> = Number of repetitions");
   System.err.println("<steps> = Number of steps per " +
   "repetition");
   System.err.println("<NB> = Number of blocks");
   System.err.println("<NT> = Number of threads per block");
   terminate (1);
}

// This program requires one CPU core.
protected static int coresRequired()
{
   return 1;
}

// This program requires all GPU accelerators on the node.
protected static int gpusRequired()
{
   return ALL_GPUS;
}

Listing 34.6. WalkSackGpu.java (part 7)