Parallelizing Branch-and-Bound on GPUs for Optimization of Multiproduct Batch Plants

Andrey Borisenko[1], Michael Haidl[2], and Sergei Gorlatch[2,3]

[1] Tambov State Technical University, Russia
borisenko@mail.gaps.tstu.ru
[2] University of Muenster, Germany
{michael.haidl|gorlatch}@uni-muenster.de
[3] Cells-in-Motion Cluster of Excellence (EXC 1003 – CiM), Muenster, Germany

Abstract. Parallel implementation of the Branch-and-Bound (B&B) technique for optimization problems is a promising approach to accelerating their solution, but it remains challenging on Graphics Processing Units (GPUs) due to B&B's irregular data structures and poor computation/communication ratio. The contributions of this paper are as follows: 1) we develop two basic implementations (iterative and recursive) of B&B on systems with GPUs for a practical application scenario - optimal design of multi-product batch plants; 2) we propose and implement several optimizations of our CUDA code using both algorithmic techniques of reducing branch divergence and GPU-specific properties of the memory hierarchy; and 3) we evaluate our implementations and optimizations on a modern GPU-based system and we report our experimental results.

Keywords: GPU computing, CUDA, branch-and-bound, combinatorial optimization, multi-product batch plant design

1 Motivation and Related Work

Combinatorial optimization [8] is often very time-consuming due to "combinatorial explosion" – the number of combinations to be examined grows exponentially, such that even the fastest supercomputers would require an intolerable amount of time. A common approach in applications is to formulate a mixed-integer nonlinear programing (MINLP) model [6, 15] and to exploit the Branch-and-bound (B&B) technique for solving it. In B&B, the search space is represented as a tree whose root is the original problem, the internal nodes are partially solved subproblems, and the leaves are the potential solutions. B&B proceeds in several iterations where the best solution found so far (upper bound) is progressively improved: a bounding mechanism is used to eliminate the subproblems that are not likely to lead to optimal solutions and to cut their corresponding sub-trees. This reduces the size of the explored search space, but can be still time-consuming in practice and requires acceleration, for example using parallel computing.

Parallelization of B&B has been extensively studied, recently with a focus on systems comprising Graphics Processing Units (GPUs). Recent approaches

usually address the most time consuming bounding mechanism of B&B [10]. The main difficulty in B&B are irregular data structures not well suited for GPU computing and the low computation/communication ratio. In [2], a hybrid implementation of B&B for the knapsack problem demonstrates that for small problem sizes it is not efficient to launch the B&B computation kernels on GPU. A parallel implementation in [3] with CUDA makes use of data y compression. In [11], a hybrid CPU-GPU implementation is presented, and [16] studies the design of parallel B&B in large-scale heterogeneous compute environments with multiple shared memory cores, multiple distributed CPUs and GPU devices.

In this paper, we parallelize B&B and illustrate it with a practical application – the optimal selection of equipment for multi-product batch plants [12]. We develop and evaluate an implementation of the B&B method on a CPU-GPU system using the CUDA programming environment [13] in two versions – an iterative and a recursive one – and we describe their optimizations, as well as compare them to each other. We report experimental results about the speedup of our GPU-based implementations as compared to the sequential CPU version.

2 Problem Formulation

Our application use case is a *Chemical-Engineering System* (CES) – a set of equipment (reactors, tanks, filters, dryers etc.) which implement the processing stages for manufacturing certain products. CES comprises I processing stages; i-th stage is equipped with units from a finite set X_i, with J_i being the number of equipment variants in X_i. All equipment variants of a CES are described as $X_i = \{x_{i,j}\}, i = \overline{1, I}, j = \overline{1, J_i}$, where $x_{i,j}$ is the main size j (working volume) of the unit suitable for stage i. A variant $\Omega_e, e = \overline{1, E}$ of a CES, where $E = \prod_{i=1}^{I} J_i$ is the number of all possible system variants, is an ordered set of equipment unit variants, selected from the respective sets. Each variant Ω_e of a system must be in an operable condition (*compatibility constraint*) i.e., it must satisfy the conditions of a joint action for all its processing stages: $S(\Omega_e) = 0$ if compatibility constraint is satisfied. An operable variant of a CES must run at a given production rate in a given period of time (*processing time constraint*), such that it satisfies the restrictions for the duration of its operating period $T(\Omega_e) \leq T_{max}$.

Thus, designing an optimal CES can be formulated as the following optimization problem [9]: to find a variant $\Omega^* \in \Omega_e, e = \overline{1, E}$ of a CES, where the optimality criterion – equipment costs $Cost(\Omega_e)$ – reaches a minimum, and both compatibility constraint and processing time constraint are satisfied:

$$\Omega^* = argmin\ Cost(\Omega_e), e = \overline{1, E} \tag{1}$$

$$\Omega_e = \{x_{1,j_1}, x_{2,j_2}, \ldots, x_{I,j_I} | j_i = \overline{1, J_i}, i = \overline{1, I}\}, e = \overline{1, E} \tag{2}$$

$$x_{i,j} \in X_i,\ i = \overline{1, I}, j = \overline{1, J_i} \tag{3}$$

$$S(\Omega_e) = 0, e = \overline{1, E} \tag{4}$$

$$T(\Omega_e) \leq T_{max}, e = \overline{1, E} \tag{5}$$

Figure 1 shows the search space as a tree: all possible variants of a CES with I stages are represented by a tree of height I (see Figure 1). Each tree level corresponds to one processing stage of the CES, each edge corresponds to a selected equipment variant taken from X_i, where X_i is the set of possible variants at stage i of the CES. For example, the edges from level 0 correspond to elements of X_1. Each node $n_{i,k}$ at the tree layer $N_i = \{n_{i,1}, n_{i,2}, \ldots, n_{i,k}\}$, $i = \overline{1,I}, k = \overline{1,K_i}, K_i = \prod_{l=1}^{i}(J_l)$ corresponds to a variant of a beginning part of the CES, composed of equipment units for stages 1 to i. Each path from the root to one of the leaves thus represents a complete variant of the CES. To enumerate all possible variants of a CES, a depth-first traversal of the tree is performed as in Figure 1: starting at level 0, all device variants of the CES at a given level are enumerated and appended to the valid beginning parts of the CES obtained at previous levels, starting with an empty beginning part at level 0. This process continues recursively for all valid beginning parts resulting from appending device variants of the current level to the valid beginning parts from previous levels. When a leaf is reached, the recursive process stops and the new solution is compared to the current optimal solution, possibly replacing it.

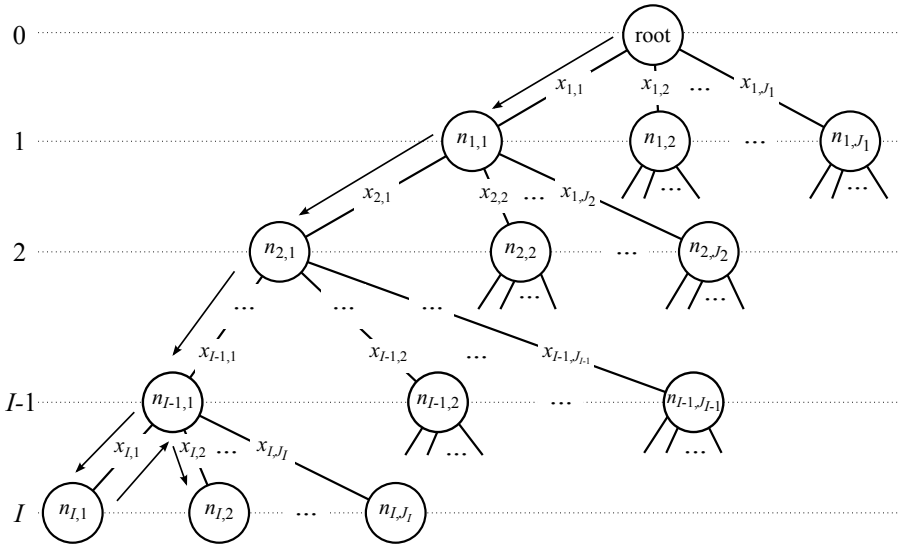

Fig. 1. Tree traversal in depth-first search.

A complete tree traversal (selecting a device on each edge traversal) and checking constraints (Equations 4 and 5)would result in a considerable computational effort. E.g., for a CES consisting of 16 stages where each process stage can be equipped with devices of 5 to 12 standard sizes [9], the number of choices is 5^{16}–12^{16} (approximately 10^{11}–10^{17}). Hence, performing an exhaustive search (pure brute-force solution) for finding a global optimum is usually impractical.

3 Parallelization for GPU

Figure 2 illustrates our strategy for dividing the initial search tree into subtrees for parallels processing: the sequential *host* process on the CPU dispatches computations to multiple *device* threads on the GPU and then gathers the results from these threads. The tree-like organization of B&B provides a potential for parallelization, as all branches of the tree can be processed simultaneously.

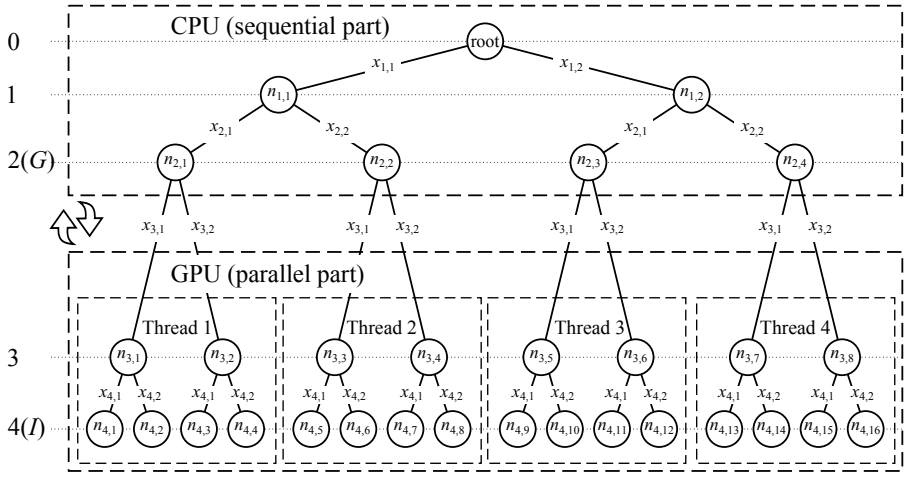

Fig. 2. Dividing the search tree into subtrees for parallel processing.

All nodes $N_i = \{n_{i,1}, n_{i,2}, \ldots, n_{i,k}\}$, $i = \overline{G+1, I}$, $k = \overline{1, K_i}$, $K_i = \prod_{l=1}^{i} J_l$ at each layer i below G are traversed in an independent *thread*. The total number of threads is $N_{threads} = \sum_{i=1}^{G} K_i$, $1 \le G \le I$. The *granularity* parameter G limits the number of threads: each subtree below the granularity level will be processed as one thread on the GPU. E.g., Figure 2 shows a CES consisting of 4 stages ($I = 4$) where each stage can be equipped with 2 devices ($J_1 = J_2 = J_3 = J_4 = 2$); the number of all possible system variants is $2^4 = 16$. We use granularity $G = 2$, so the threads number is $2^2 = 4$. All three nodes at layers from 0 to 2 are processed on CPU, then partial solutions are transferred to GPU, and all other nodes at layers from 3 to 4 are processed in parallel on GPU.

Host Code We use the CUDA Runtime API [5,14]. The host (see Listing 1) begins its work by loading *input data* from file by calling `ReadInputData()` (line 2). The `inData` is a pointer to the structure whose fields store all necessary input data for the calculation. The values of these data are not changed during calculations and are the same for all threads, i.e. constant. Based on the input data `inData` and *granularity* parameter `G`, the required number of threads `numThreads` is computed by `ThreadsNumber()` (line 4) and used in `PrepOperationalData()`

for preparing *operational data* (line 6). The `oprData` is a pointer to the structure whose values are changed in the calculations independently by each thread with *thread index* `threadID`.

```
1  main() {  /* prepare all necessary data */
2    ReadInputData(inData);
3    /* number of threads */
4    numThreads = ThreadsNumber(inData, G);
5    /* prepare all operational data */
6    PrepOperationalData(oprData, inData, numThreads);
7    /* start tree traversal for dividing tree into */
8    /* subtrees and creating beginning parts of the CES */
9    EnumerateHost(0, 0);
10   /* send all necessary data to device */
11   cudaMemcpyHtoD(inData,oprData,W, G);
12   /* define parameters for kernel launch */
13   blocksPerGrid = numThreads / MAX_THREADS_PER_BLOCK + 1;
14   threadsPerBlock = MAX_THREADS_PER_BLOCK;
15   /* starting kernel function on device */
16   FindSolution<<<blocksPerGrid, threadsPerBlock>>>(G);
17   /* synchronize device */
18   cudaDeviceSynchronize();
19   /* copy the results from device to host */
20   cudaMemcpyDtoH(W, Cost, minCost);
21   /* find global optimal solution */
22   for (n=1; n<=numThreads; n++) {
23     if(Cost[n] < minCost) { Wopt = W[n];
24       minCost = Cost[n]; }}}
25 EnumerateHost(threadID, level) {
26   for (j=1; j <= J[level]; j++) {
27     Wloc[level] = X[level, j];
28     if (level < G) { /* check granularity */
29       EnumerateHost(threadID, level + 1); }
30     else{ W[threadID] = Wloc;
31       threadID++; }}}
```

Listing 1. The host pseudo-code.

Both `ReadInputData()` and `PrepOperationalData()` make all necessary memory allocations and variables initialization. Then host performs a depth-first traversal of the tree using recursive function `EnumerateHost()` to the level defined by `G` (line 9). Here `W` is a two-dimensional array, represented as an array of length `numThreads` each element of this is a vector of length `I` specifying the device variant at each stage of the solution. In lines 25 – 31, the host creates beginning parts of CES at levels from `0` to `G` and saves them in `W`. The host neither checks constraints (Equations 4 and 5) nor evaluates upper and lower bounds of the objective function (Equations 1); this will be performed on the GPU. After the host sends all necessary data to the device (line 11), it defines

the number of blocks (line 13) and threads (line 14) in one block and then starts the kernel function `FindSolution()` (line 16).

A CUDA kernel launch is asynchronous and returns control to the CPU immediately after starting the GPU process. By calling `cudaDeviceSynchronize()` (line 18), the CPU is forced to idle until the GPU work has completed and the host receives the results from the device (line 20). Here `Cost` is an array of size `numThreads`; its elements are the local optimal costs of CES which were found by each thread. The `minCost` is the global minimal cost of CES, whose value is used to seek for the best optimal solution. The host compares local minimal costs with global minimum and searches for the best CES-variant (lines 22 – 24). Here, vector `Wopt` specifies the unit variant at each stage of the optimal solution.

Kernel Code We have developed a recursive (Listing 2) and an iterative (Listing 3) tree traversal implementations. The recursive approach can be used on NVIDIA GPU devices of Compute Capability 2.0 and higher. On the older NVIDIA devices, the iterative approach should be used.

```
1  __global__ void FindSolution(G)
2  { /* obtaining thread identifier */
3    threadID = blockDim.x * blockIdx.x + threadIdx.x;
4    /* if threadID  not greater maximal thread numbers */
5    if(threadID <= numThreads) {
6      /* start subtree traversal */
7      EnumerateDevice(threadID, G + 1); }}
8  __device__ EnumerateDevice(threadID, level)
9  { for (j=1; j <= J[level]; j++) {
10     /* append device variant to beginning part */
11     Wloc[level] = X[level, j];
12     if (level < I) {
13       /* check compatibility constraint and upper bound */
14       if (S(Wloc) == 0 && DefineCost(Wloc, level) < minCost){
15         /* search recursively */
16         EnumerateDevice(threadID, level + 1); }}
17     else {/* leaf node */
18       /* check optimality criterion */
19       if (Cost (Wloc) < minCost)) {
20       /* check processing time constraint */
21         if (T(Wloc) <= Tmax) {
22           /* make solution new (local) optimal solution */
23           W[threadID] = Wloc;
24           Cost[threadID] = DefineCost(Wloc, I);
25           atomicMinDbl(minCost, Cost[threadID]); }}}}}
```

Listing 2. The kernel pseudo-code using recursive approach.

The kernel function `FindSolution()` (Listing 2) calculates a global thread identifier `threadID` (line 3) which is compared with the required number of threads `numThreads`(line 5). This is necessary because of rounding of kernel

launch parameters (see Listing 1, line 13): the actual number of running threads can be greater than `numThreads`, but all memory allocations for `oprData` are done for `numThreads`. For all threads with valid `threadID`, the kernel function calls `EnumerateDevice()` at level `G+1` (line 7). Within this function, the device traverses the remaining sub-trees at levels from `G+1` to `I` of the received CES' beginning parts to find solutions. All unit variants of the CES at a given level are enumerated and appended to the beginning parts of the CES. Valid beginning parts are obtained at previous levels, starting at level `G+1`. This process continues recursively for all valid beginning parts that result from appending unit variants of the current level to the valid beginning parts from previous levels.

When a leaf node is reached, the recursive process stops and the current solution is compared to the current optimal solution, possibly replacing it. When traversing the tree, the compatibility constraint (Equation 4, function `S()`) is checked for the corresponding part of the CES. We also compare the cost for the current beginning part of the CES, consisting of the first level stages (function `DefineCost()`) with a global upper bound (variable `minCost`) (line 14). The initialization of the upper bound is done with the sum of all maximum units costs for each production stage. If the current beginning part of the CES fulfills the compatibility constraint and its costs do not exceed the global upper bound, we recursively continue tree traversal to the next level (line 16). If a leaf node of the tree is reached (17), a new full solution has been found and its costs (Equation 1, function `Cost())` are compared to the cost of the previous optimal solution `minCost` (line 19). If a better solution is obtained, the processing time constraint (Equation 5, function `T()`) is checked for the corresponding CES (line 21). If this constraint is fulfilled, a new valid optimal solution replaces the previous optimal solution (line 23), saves it value (line 24) and its costs are taken as the new upper bound and saved with atomic function `atomicMinDbl` (line 25).

The iterative stack-based traversal algorithm is presented in Listing 3. The kernel function is the same like in Listing 2 (lines 1 – 7), so we omit it. In our implementation, stack, which must be not smaller than the number of tree levels, is a one-dimensional array `stack` of size `MAX_STACK_SIZE` (line 10). The stack is accessed for adding and removing data elements through its *top*, variable `stackTop` (line 11). The standard elementary stack operations are `push` (line 13), `pop` (line 14), and `empty` (line 15). At first stack is empty and therefore the first node (units variant for level 1) of the tree is added (line 17). While the stack is not empty, we pop the stack, find all possible choices after the previous one and push these choices onto the stack (line 19). At each loop iteration, the last item on the stack is popped (line 20). If at the current level there are no other nodes, we return to the previous level (line 22). Otherwise we push the next node at the current level into the stack (line 24). If the current beginning part of the CES fulfills the compatibility constraint and its costs do not exceed the global upper bound (line 26), we append unit variants at the current level to the valid beginning parts of CES from previous levels (line 28). Otherwise we discard deeper levels of the tree and go to the next loop iteration. If the last level is not reached (line 29), we continue tree traversal to the next level (line 30) and

push the first node at the next level into the stack (line 32). If a leaf of the tree is reached (line 33), the algorithm works like recursive version (see Listing 2). The solution cost is compared to the global cost of the previous optimal solution (line 35). If a better solution is obtained, then processing time constraint is checked for the corresponding CES (line 37). If this constraint is fulfilled, this mean that a new optimal solution has been found which replaces the previous optimal solution and its cost is taken as the new upper bound (lines 39 – 41).

```
7  ...
8  __device__ EnumerateDevice(threadID, level)
9  { /* declaration of standard last-in, first-out stack */
10   stack[MAX_STACK_SIZE];
11   stackTop = 0;
12   /* defining standard stack operations */
13 #define push(x) stack[stackTop++] = (x)
14 #define pop() stack[--stackTop]
15 #define empty (stackTop == 0)
16   /* add to stack first item on level 1 */
17   push(1);
18   /* loop while stack is not empty */
19   while (!empty) {  /* retrieve the top node from stack */
20     j = pop();
21     /* if on this level are not other nodes */
22     if(j == J[level] + 1) {level--;}
23     else {  /* add to stack next trees node on current level*/
24       push(j + 1);
25       /* check compatibility constraint and upper bound */
26       if (S(Wloc) == 0 && DefineCost(Wloc, level) < minCost){
27         /* append device variant to beginning part */
28         Wloc[level] = X[level, j];
29         if (level < I) { /* go to next trees level */
30           level++;
31           /* add to stack first node on current level */
32           push(1); }
33         else {/* leaf node */
34           /* check optimality criterion */
35           if (Cost (Wloc) < minCost)) {
36             /* check processing time constraint */
37             if (T(Wloc) <= Tmax) {
38             /* make solution new (local) optimal solution */
39               W[threadID] = Wloc;
40               Cost[threadID] = DefineCost(Wloc, I);
41               atomicMinDbl(minCost, Cost[threadID]); }}}}}}}
```

Listing 3. The kernel pseudo-code using iterative approach.

4 Optimizations

In our previous work [1] we have used Standard Template Library (STL), in particular container class *std::vector*, for implementing the mathematical model of CES. Since there is no implementation of STL containers in CUDA, we revised the program code, using many (about 50) multidimensional (from 1- to 5-dimensional) arrays. To keep the arrays' contents contiguous, we simulate a multidimensional array with a one-dimensional array which guarantees that all array elements are in a flat chunk of memory. This is convenient for data transfer with standard `memcpy`-like functions and faster for the memory access as compared to fragmented memory (e.g., there are fewer cache misses and better performance). On the test platform presented in Section 5, this new sequential CPU version without STL vectors is approximately 8 – 9 times faster than the STL-based sequential version. Our experiments in the sequel are carried out for this new CPU version of the program.

We perform and evaluate two optimizations: a) shared memory utilization, and b)reducing branching in the kernel function.

Our target architecture (GPU) comprises *Streaming Multiprocessors* (SMs) contains some *Streaming Processors* (SP) (since Fermi microarchitecture NVIDIA changes the name SP to *CUDA cores*). The DRAM or *Global memory* is the biggest memory region (several gigabytes) on the graphic device. It is off-SM, therefore relatively slow and can be accessed both by host and device. In addition, every SM has a small on-chip memory that can be configured as *Shared memory* and as *L1 cache*. In addition to the *L1 cache*, Kepler introduces a *Read-only data cache* for data for the duration of the function. The *L2 cache* is the primary point of data unification between the SM units, servicing all load and store requests and providing data sharing across the GPU.

```
2  ...
3    threadID = blockDim.x * blockIdx.x + threadIdx.x;
4    __shared__ sharedMemory[];
5    /* send data by shared memory pointer */
6    __device__ SendDataByMemoryPtr(Wloc, sharedMemory);
7  #ifdef ITERATIVE
8    __device__ SendDataByMemoryPtr(stack, sharedMemory);
9  #endif
10   /* if threadID  not greater maximal thread numbers */
11   if(threadID <= numThreads){
12     /* start subtree traversal */
13     EnumerateDevice(threadID, G + 1);}
14 ...
```

Listing 4. The kernel pseudo-code with optimization *O1*.

Listing 4 shows our first memory optimization: we move the local array `Wloc` (line 6) from global memory to shared memory using device function `SendDataByMemoryPtr()` and for the iterative approach we move the `stack` too (line 8) (optimization *O1*). Moreover we move `inData` from the constant to the shared memory, as shown in Listing 5 line 6 (optimization *O2*), using function `SendDataByMemoryPtr()` which is a device function for sending data to the specified memory address.

The SMs of an NVIDIA GPU only get one instruction at a time and all CUDA cores execute the same instruction. Threads within a *warp* (a group of 32 thread, which are used in the hardware implementation to coalesce memory access and instruction dispatch) execute the same instruction in each cycle, disabling threads that are not on the same path of control-flow; this is also known as *thread or branch divergence* [4, 7]. The most common code construct that can cause thread divergence is branching for conditionals in an *if-then-else* statement. Branch divergence can hurt performance due to lower utilization of the processing elements, which cannot be compensated for via increased amount of parallelism [7].

```
2  ...
3    threadID = blockDim.x * blockIdx.x + threadIdx.x;
4    __shared__ sharedMemory[];
5    /* send data by shared memory pointer */
6    SendDataByMemoryPtr(inData, sharedMemory);
7    /* if threadID  not greater maximal thread numbers */
8    if(threadID <= numThreads){
9      /* start subtree traversal */
10     EnumerateDevice(threadID, G + 1);}
11 ...
```

Listing 5. The kernel pseudo-code with optimization *O2*.

In addition to shared memory utilization, we reduce the branch divergence by removing the checking of the compatibility constraint and the upper bound (Listing 2 line 14, Listing 3 line 26). In this case all restrictions are checked only at the last tree level, such that code paths have fewer branches (this effectively transforms B&B into an exhaustive search). The implementations optimized this way get suffix 'm' (e.g., Recursion *O1m*, Iteration *O2m*) in our evaluation.

As an additional advantage, this optimization also reduces the stall time of GPU threads which finish their work earlier and remain idle while waiting for the last thread of the same warp to finish. The stall times of threads could alternatively be minimized by a more complex work distribution across threads. However this would arguably introduce more thread divergence, and also require additional information exchanges across threads, as well as higher load on the registers and shared memory, with more atomic operations, which together would negatively affect the kernel's runtime.

5 Experimental Results

Our experiments are conducted on a hybrid system comprising: 1) CPU: Intel Xeon E5-1620 v2, 4 cores with Hyper-Threading, 3.7 GHz with 16 GB RAM, and 2) GPU: NVIDIA Tesla K20c with 13 SMs, each with 192 CUDA Cores (total 2496 CUDA Cores), 5GB of global memory and up to 48KB of shared memory per SM. We use Ubuntu 14.04.1, NVIDIA Driver version 340.29, CUDA version 6.5 and GNU C++ Compiler version 4.8.2. We study the design of a CES consisting of 16 processing stages with 3 variants of devices at every stage as test case (total $3^{16} = 43\,046\,721$ CES variants).

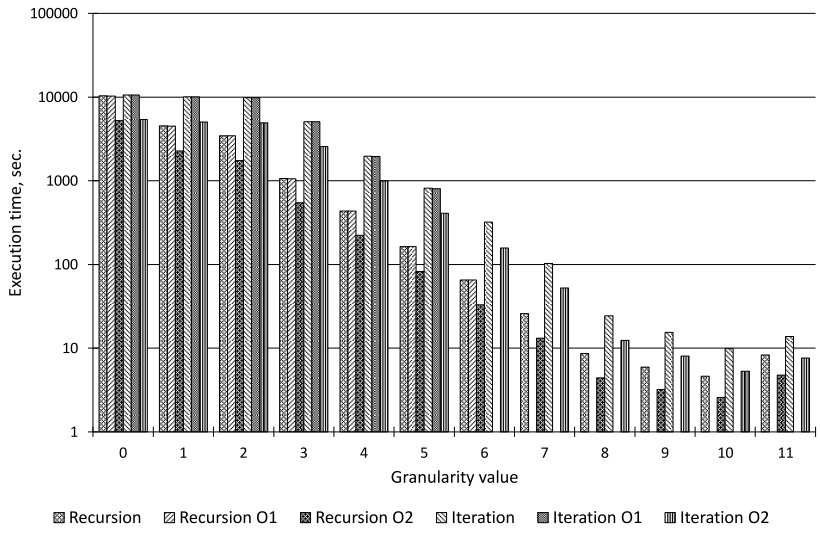

Fig. 3. Results of memory optimization. Run-time depending on granularity.

Our first experiment (see Figure 3)concerns selecting a suitable granularity value which sets the level of parallelism and, therefore, is important for parallel program performance. On the one hand, increasing the number of threads increases the performance of the parallel program on the GPU, but on the other hand, it needs more memory: each thread needs own memory in `oprData`. We run our CUDA-based implementation setting granularity values from 1 to 11. We observe in Figure 3 that the runtime is reduced with increasing granularity, but for larger granularity values, too many threads are created and too much memory is required for operational data `oprData`, such that the implementation runs out of global memory for certain versions, and for granularity greater than 11 it runs out of memory for all versions.

The results of our memory optimization are also shown in Figure 3 (the vertical axis has a logarithmic scale). For optimization *O1*, we use shared memory for

array `Wloc` (recursive approach) and for `Wloc` and `stack` (iterative approach). For granularity greater than 6 for recursive and greater than 5 for iterative approach, we run out of shared memory. The total runtime of our program consists of the runtimes of the CPU-part and the GPU-part. Our measurements show that the CPU runtime affects insignificantly the total runtime (CPU takes about 0.1% of the total runtime). While optimization *O1* does not have a significant impact on the runtime, optimization *O2* significantly reduces the execution time, on average by 48%. We observe that the recursive implementation (Listing 2) is faster than the iterative one (Listing 3) on average by about 2.2 times.

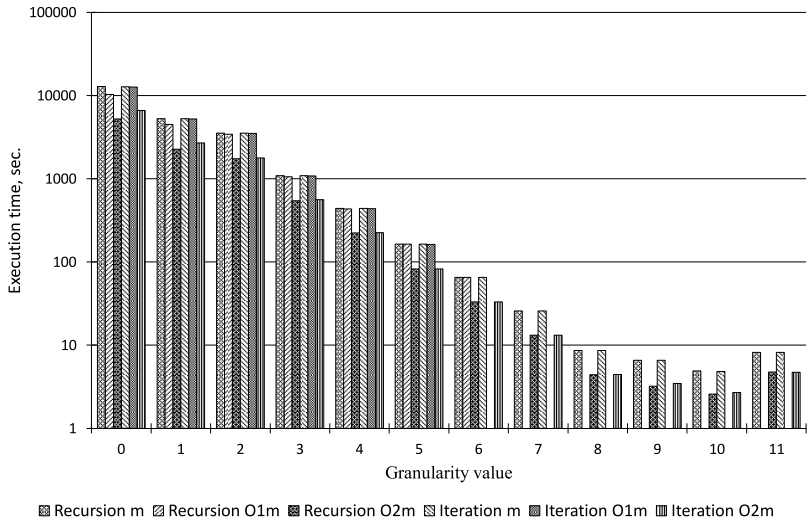

Fig. 4. Results of branch optimization. Run-time depending on granularity.

The results of branch optimization are presented in Figure 4: in addition to the shared memory utilization, we remove the checking of the compatibility constraint and the upper bound (Listing 2 line 14, Listing 3 line 26). We observe in Figure 4 that the branch optimization has significantly different effects for recursion and iteration: for recursion the runtime was slightly increased within 5%; for iteration the runtime was decreased on average by 60%. Based on the experimental data, we conclude that for the recursive version the fastest is the *O2*-optimization, and for the iterative version the *O2m*-optimization is the fastest. The fastest recursive version for our test example is faster than the fastest iterative version by about 5%.

In Figure 5, the speedups of our parallel GPU versions for Recursion *O2* and Iteration *O2m* tree traversal implementations are presented as compared to the sequential CPU-program.

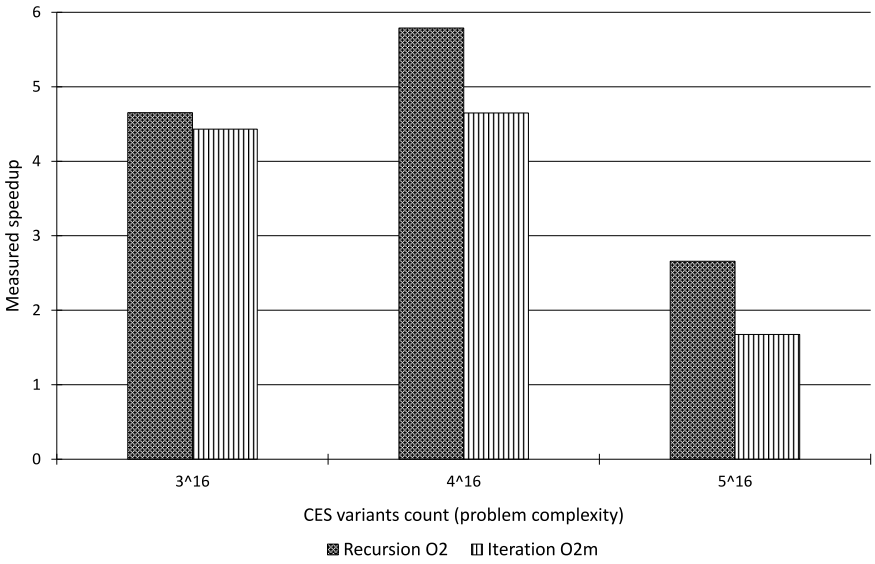

Fig. 5. Experimental results. Measured speedup.

We have run several test cases for problems of larger dimension and we measured the speedup of the parallel GPU-based implementation vs. sequential CPU-program. Our experiments were carried out for the above example (16 processing stages) but with 4 (total $16^4 = 4\,294\,967\,296$ CES variants) and 5 (total $16^5 = 152\,587\,890\,625$ CES variants) units, correspondingly.

We observe in Figure 5 that the speedup value for the recursive version is between 2.66 and 5.79, and for the iterative version it is between 1.67 and 4.43.

6 Conclusion

We proposed two parallelization approaches to implement a parallel B&B algorithm for solving the optimization problem for multi-product batch plants on a CPU-GPU platform.

Two basic implementations – based on recursive and iterative approaches to the tree traversal – have been presented. We analyzed the impact of the degree of parallelism controlled by the granularity parameter, and conducted GPU-specific optimizations of our programs: using shared memory and reducing branch divergence. Our results show that the recursion-based implementation in general is faster than the iteration-based on our test platform, and that our optimizations significantly reduce the total runtime. Our future work will extend the optimization space using the most modern features of the GPU programming approaches.

Acknowledgement

This work was partially supported by the Deutsche Forschungsgemeinschaft (DFG), Cells-in-Motion Cluster of Excellence (EXC 1003 – CiM), University of Muenster, Germany. Andrey Borisenko was supported by the DAAD (German Academic Exchange Service) and by the Ministry of Education and Science of the Russian Federation under the "Mikhail Lomonosov II"-Programme.

References

1. Borisenko, A., Kegel, P., Gorlatch, S.: Optimal design of multi-product batch plants using a parallel branch-and-bound method. In: Parallel Computing Technologies, Lecture Notes in Computer Science, vol. 6873, pp. 417–430. Springer (2011)
2. Boukedjar, A., Lalami, M.E., El Baz, D.: Parallel branch and bound on a CPU-GPU system. In: PDP. pp. 392–398. Citeseer (2012)
3. Boyer, V., El Baz, D., Elkihel, M.: Solving knapsack problems on GPU. Computers & Operations Research 39(1), 42–47 (2012)
4. Chakroun, I., Mezmaz, M., Melab, N., Bendjoudi, A.: Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm. Concurrency and Computation: Practice and Experience 25(8), 1121–1136 (2013)
5. Farber, R.: CUDA application design and development. Elsevier (2011)
6. Fumero, Y., Corsano, G., Montagna, J.M.: A mixed integer linear programming model for simultaneous design and scheduling of flowshop plants. Applied Mathematical Modelling 37(4), 1652–1664 (2013)
7. Han, T.D., Abdelrahman, T.S.: Reducing branch divergence in GPU programs. In: Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units. p. 3. ACM (2011)
8. Hoffman, K., Padberg, M.: Combinatorial and integer optimization. In: Encyclopedia of Operations Research and Management Science, pp. 94–102. Springer (2001)
9. Malygin, E., Karpushkin, S., Borisenko, A.: A mathematical model of the functioning of multiproduct chemical engineering systems. Theoretical foundations of chemical engineering 39(4), 429–439 (2005)
10. Melab, N., Chakroun, I., Mezmaz, M., Tuyttens, D.: A GPU-accelerated branch-and-bound algorithm for the flow-shop scheduling problem. In: Cluster Computing (CLUSTER), 2012 IEEE International Conference on. pp. 10–17. IEEE (2012)
11. Meyer, X., Chopard, B., Albuquerque, P.: A branch-and-bound algorithm using multiple GPU-based LP solvers. In: High Performance Computing (HiPC), 2013 20th International Conference on. pp. 129–138. IEEE (2013)
12. Mokeddem, D., Khellaf, A.: Optimal solutions of multiproduct batch chemical process using multiobjective genetic algorithm with expert decision system. Journal of Analytical Methods in Chemistry 2009 (2009)
13. NVIDIA Corporation: CUDA C programming guide 6.5 (August 2014), `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`
14. Sanders, J., Kandrot, E.: CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional (2010)
15. Terrazas-Moreno, S., Grossmann, I.E., Wassick, J.M.: A mixed-integer linear programming model for optimizing the scheduling and assignment of tank farm operations. Industrial & Engineering Chemistry Research 51(18), 6441–6454 (2012)
16. Vu, T., Derbel, B.: Parallel branch-and-bound in multi-core multi-CPU multi-GPU heterogeneous environments (2014), `https://hal.inria.fr/hal-01067662`