Volume 71, Issue 6, June 2011, Pages 802-811- Special Issue on Cloud Computing

Accepted Manuscript

Tabu Search with two approaches to parallel flowshop evaluation on CUDA platform

Michał Czapiński, Stuart Barnes

PII: S0743-7315(11)00038-4
DOI: 10.1016/j.jpdc.2011.02.006
Reference: YJPDC 2869

To appear in: *J. Parallel Distrib. Comput.*

Received date: 10 December 2009
Revised date: 24 February 2011
Accepted date: 25 February 2011

Please cite this article as: M. Czapiński, S. Barnes, Tabu Search with two approaches to parallel flowshop evaluation on CUDA platform, *J. Parallel Distrib. Comput.* (2011), doi:10.1016/j.jpdc.2011.02.006

- Two efficient GPU (CUDA) implementations of Tabu Search for Flowshop are proposed.
- Two approaches to parallel evaluation for any population-based algorithm are proposed.
- Tabu Search for Flowshop runs up to 89 times faster on GPU than on CPU.

Tabu Search with two approaches to parallel flowshop evaluation on CUDA platform

Michał Czapiński[a,*] Stuart Barnes[a]

[a] *Applied Mathematics and Computing Group, Cranfield University, College Road, Cranfield, Bedfordshire, MK43 0AL, England*

*Corresponding author.
E-mail address: mczapinski@gmail.com
Phone number: 0044 1234 754662

Abstract

The introduction of NVidia's powerful Tesla GPU hardware and Compute Unified Device Architecture (CUDA) platform enable many-core parallel programming. As a result, existing algorithms implemented on a GPU can run many times faster than on modern CPUs. Relatively little research has been done so far on GPU implementations of discrete optimisation algorithms. In this paper, two approaches to parallel GPU evaluation of Permutation Flowshop Scheduling Problem, with makespan and total flowtime criteria, are proposed. These methods can be employed in most population-based algorithms, e.g. genetic algorithms, Ant Colony Optimisation, Particle Swarm Optimisation, and Tabu Search. Extensive computational experiments, on Tabu Search for Flowshop with both criteria, followed by statistical analysis, confirm great computational capabilities of GPU hardware. A GPU implementation of Tabu Search runs up to 89 times faster than its CPU counterpart.

Keywords: GPGPU, CUDA, Tabu Search, permutation flowshop, parallel algorithms, discrete optimisation

1 Introduction

The introduction of Compute Unified Device Architecture (CUDA), caused rapid increase of interest of scientists and industry, in General Purpose Computation Using Graphics Hardware (GPGPU). Two factors contributed the most to the success of CUDA. First, is that during the last few years, computational capability of Graphics Processing Units (GPU) is growing much faster than capabilities of CPUs. This was mainly due to greatly increasing hardware requirements of modern computer games. Lately, NVidia released a whole line of GPUs, named Tesla, designed for high performance computing, rather than playing games.

Programmable GPU hardware was available even before CUDA. GPGPU started with programmable vector units (pixel and vertex shaders), though possibilities were very limited, due to small set of instructions and restricted code length. As a result, GPGPU-based development was considerably difficult and the benefits were not that impressive, resulting in relatively little interest in GPGPU. On the other hand, CUDA comes in a form of simple extension to C/C++ languages, allowing easy programming of GPU, by hiding almost all GPU-related issues. Of course, in order to maximise performance of CUDA applications, extensive knowledge of hardware features is still required, but many optimisations are done automatically.

Most recent research is mostly focused on how can great computational capabilities of modern GPUs be used to speed up solving computationally demanding problems. Many existing algorithms, and even whole libraries (e.g. Basic Linear Algebra Subprograms — BLAS) are parallelised and implemented to run on GPU, resulting in considerable increase in performance. Relatively little research has been done so far on CUDA implementation of discrete optimisation metaheuristics. Some results in that area were published e.g. by Janiak et al. [12], who propose GPU implementation of Tabu Search for Travelling Salesman Problem (TSP) and Permutation Flowshop Scheduling Problem with makespan criterion. Luong et al. [17] implemented few classic discrete optimisation methods (e.g. Tabu Search, Hill Climbing and Iterated Local Search), to solve Quadratic Assignment Problem (QAP), Permuted Perceptron Problem, and TSP.

Permutation Flowshop Scheduling Problem (PFSP) has been studied since 1954 when it was

first proposed by Johnson [14]. The goal of PFSP is to order n jobs to be processed on m machines. Processing time for each job on each machine is provided and fixed. All jobs are available when processing starts and they must be processed uninterrupted on each machine in the same order. Sequence of processing jobs on each machine is assumed to be the same, and therefore solution to PFSP is a permutation of jobs entering machine 1, which optimises an objective function (criterion). The most common ones are makespan criterion (C_{max}), and total flowtime criterion (C_{sum}). Garey et al. [7] proved that PFSP with C_{max} is $\mathcal{NP}$-complete in the strong sense when $m \geq 3$, and with C_{sum} when $m \geq 2$. Both criteria are considered in research presented in this paper.

A great number of sequential algorithms for solving PFSP were published. This include discrete optimisation methods such as Branch and Bound (e.g. by Chung et al. [4]), simple, yet effective, constructive heuristics (e.g. by Nawaz et al. [18], Liu and Reeves [16], and Li et al. [15]), Tabu Search (e.g. by Nowicki and Smutnicki [19] and by Grabowski and Wodecki [10]), Simulated Annealing (e.g. by Ishubuchi et al. [11]), Genetic Local Search (e.g. by Yamada and Reeves [25]), Ant Colony Optimisation (e.g. by Rajendran and Ziegler [21]), Particle Swarm Optimisation (e.g. by Tasgetiren et al. [23]), Hybrid Genetic Algorithms (e.g. by Tseng and Lin [24] and by Zhang et al. [26]), Iterated Local Search (e.g. by Dong et al. [6]), and finally Estimation of Distribution Algorithm (e.g. by Jarboui et al. [13]).

Recently, some research has been done on parallel algorithms, running on clusters of CPUs. Among published parallel algorithms, based on classic metaheuristics, are e.g. Parallel Genetic Algorithm by Bożejko and Wodecki [2], Parallel Scatter Search by Bożejko and Wodecki [3] and by Bożejko [1], and Parallel Simulated Annealing with Genetic Enhancement by Czapiński [5]. Furthermore, Bożejko [1] provides very detailed analysis of theoretical and empirical speed-ups obtained by parallelisation of PFSP evaluation.

In most metaheuristics for discrete optimisation, the most time consuming part is objective function evaluation. As reported by Janiak et al. [12], this task in algorithms for TSP and PFSP, usually takes over 90% of total computational effort. Therefore, parallelisation should be focused

on this part of the algorithm. Fortunately, in most cases objective function evaluation can be quite easily parallelised. Two very efficient methods, optimised for CUDA platform, are proposed in this paper. They are not designed for any particular algorithm, and can be easily employed in any population-based metaheuristic, including genetic algorithms, Ant Colony Optimisation, Particle Swarm Optimisation, and Tabu Search. The last one is considered in this paper, and applied to PFSP with both C_{max} and C_{sum} criteria.

This paper is organised as follows: in section 2 Permutation Flowshop Scheduling Problem and Tabu Search method are described. Section 3 contains some details regarding CUDA platform, followed by section 4, in which details on CUDA implementation of PFSP evaluation and Tabu Search are presented. Section 5 contains a description and the results of extensive computational experiments. Finally, conclusions are presented in section 6.

2 Tabu Search for PFSP

In this section Tabu Search method for Permutation Flowshop Scheduling Problem (PFSP) is presented. Subsection 2.1 contains definition of PFSP. Then, subsection 2.2 describes moves and neighbourhood used in Tabu Search method, which is presented in subsection 2.3.

2.1 Permutation Flowshop Scheduling Problem

In permutation flowshop scheduling problem, a set of n jobs, is to be processed on a set of m machines. Each job must be processed uninterrupted, in the same order on every machine (starting with machine 1, then machine 2, and so on, until it is processed by machine m). While the sequence of processing jobs on each machine is assumed to be the same, the goal of PFSP is to find a permutation of jobs entering machine 1, which minimises an objective function value.

Let $p_{i,j} \in \mathbb{N}_0$ denote processing time of job j on machine i. Then for every schedule permutation π time of finishing j^{th} job in that schedule, on machine i, denoted as $C_{i,j}(\pi)$, is defined as follows:

$$
\begin{aligned}
C_{1,1}(\pi) &= p_{1,\pi_1}, \\
C_{i,1}(\pi) &= p_{i,\pi_1} + C_{i-1,1}(\pi), \\
C_{1,j}(\pi) &= p_{1,\pi_j} + C_{1,j-1}(\pi), \\
C_{i,j}(\pi) &= p_{i,\pi_j} + \max\{C_{i,j-1}(\pi), C_{i-1,j}(\pi)\},
\end{aligned}
$$

where $i \in \{2, 3, \ldots, m\}$, and $j \in \{2, 3, \ldots, n\}$.

In terms of $C_{i,j}(\pi)$, makespan (C_{max}) and total flowtime (C_{sum}) are defined as:

$$
\begin{aligned}
C_{max}(\pi) &= C_{m,n}, \\
C_{sum}(\pi) &= \sum_{i=1}^{n} C_{m,i}(\pi).
\end{aligned}
$$

Given that, to solve PFSP, is to find permutation π^*, which satisfies

$$
\forall \pi \in \Pi_n \quad F(\pi^*) \leq F(\pi),
$$

where Π_n denotes set of all permutations of length n, and $F \in \{C_{max}, C_{sum}\}$ is an objective function.

2.2 Move and neighbourhood types

Permutation search space can be interpreted as a graph with permutations as vertices. While most metaheuristics are "moving" over search space, edges of this graph can be interpreted as allowed *moves*. Two most common move types are *transposition* and *insertion* moves. Only transposition moves (swapping two elements in the permutation) are considered in this paper. Moves can be interpreted as functions taking permutation as a parameter, and returning permutation as well — $tr_{i,j} : \Pi_n \to \Pi_n$ is a function, which for a given permutation π, returns a permutation π with elements on positions i and j exchanged.

```
best candidate := NULL
for each candidate solution in the neighbourhood
    of current solution do begin
    if F(candidate solution) < F(best candidate) and
        (candidate solution is not on tabu list or
        F(candidate solution) < F(best solution)) then
        best candidate := candidate solution
end
update best solution
update tabu list
current solution := best candidate
```

Listing 1: General scheme for each iteration of Tabu Search method

Having defined $tr_{i,j}$, neighbourhood of π *generated by* transposition moves is defined as:

$$N_t(\pi) = \{tr_{i,j}(\pi) : 1 \leq i, j \leq n, i \neq j\}.$$

N_t will be referred to as *transposition neighbourhood*. It is easy to show that

$$|N_t(\pi)| = \frac{n(n-1)}{2}.$$

2.3 Tabu Search method

Tabu Search, originally proposed by Glover [8, 9], is a neighbourhood-based, deterministic metaheuristic, which can be applied to many discrete optimisation problems. The general idea behind TS, is to iteratively improve some initial solution, by searching its neighbourhood, and moving to the most promising one.

Each iteration of TS starts at some solution, and then all permutations, within its neighbourhood are evaluated. The best solution is chosen for the next iteration. In order to avoid getting stuck in some local minimum, a mechanism called *tabu list* is introduced. It can be implemented as a list of the few most recent moves, which are forbidden in the current iteration unless they provide solution better than the best known one. The general scheme for iteration of Tabu Search method is presented on listing 1. `F(solution)` denotes the objective function value of the solution.

3 Compute Unified Device Architecture

This section provides a brief overview of Compute Unified Device Architecture (CUDA). For a more detailed description, please refer to the CUDA Programming Guide by NVidia [20].

As it was mentioned before, CUDA comes in a form of simple C/C++ language extensions, and hides most GPU-related issues. It leaves three most important tasks to programmer, which are elaborated in the following subsections. These include managing threads hierarchy (subsection 3.1), managing memory access patterns (subsection 3.2), and synchronisation (subsection 3.3).

3.1 Thread hierarchy

To perform computation using CUDA, programmer must define a C function, called the *kernel*. This function is then run using a specified number of lightweight threads. Threads are grouped in *blocks*, and blocks form a *grid*.

This hierarchy is introduced in order to reflect the hardware organisation of GPU. While threads within a block are meant to, and are able, to communicate easily through shared memory, blocks are meant to run independently. This is caused by a fact, that GPU contains number of multiprocessors, and each block can be executed on a different multiprocessor. The order in which blocks are dispatched to multiprocessors is undefined, and managed by the scheduler. There may be a number of blocks being executed on a multiprocessor at the same time, as long, as sufficient resources are available (e.g. shared memory, registers).

Currently, the CUDA specification states that each block may contain up to 512 threads, organised in 1-, 2- or 3-dimensional array. The grid is limited to 2^{32} blocks, organised in 1-, or 2-dimensional array. On devices with compute capability 1.3, each multiprocessor can execute up to 8 blocks, and up to 1024 threads at the same time.

3.2 Memory hierarchy

There are many types of memory, differing in size, visibility, access time and whether it is cached and writeable. Below is the description of each memory type, its capabilities and purpose.

Global memory is used for communication between host and device, therefore it is accessible from kernels (directly), and CPU (through API). It is the largest memory space available on GPU (up to 4 GB), but it is the slowest at the same time. Both read and write operations requires 400–600 clock cycles, and they are not cached.

Shared memory is a very fast on-chip memory, available for both reading and writing, but only accessible by threads within the same block. It is not cached, while accessing it requires only 2 clock cycles. Unfortunately, it is also small — maximum of 16,384 bytes per block. This is also the physical limit of each multiprocessor, therefore if a block is using all 16,384 bytes of memory, it effectively prevents parallel execution of other blocks on the same multiprocessor.

Constant memory is accessible as global memory, but it is cached — in the case of a cache miss, a read operation takes the same time as reading from global memory, otherwise it is much faster (read from constant cache). It is also read-only from the device, and relatively small (65,536 bytes per GPU).

Registers are the fastest on-chip memory used for threads' automatic variables. The number of 32-bit registers on each multiprocessor is limited up to 16,384, therefore if each thread requires too many registers, the number of blocks executed on each multiprocessor is reduced.

Local memory is a local per-thread memory, used for large automatic variables (e.g. arrays or large structures) that will not fit in registers. It is not cached, and is as slow as global memory. Using local memory should, and usually can, be avoided.

3.3 Synchronisation

CUDA provides a simple and efficient mechanism for thread synchronisation within a block. Issuing `__syncthreads()` command in a kernel code, forces all threads to wait, until they all reach this point. There is no mechanism to synchronise the execution of blocks.

The thread synchronisation mechanism is especially useful when using shared memory to exchange data between threads in block. The order, in which threads are accessing shared memory,

is undefined, therefore to avoid race conditions, threads should synchronise after write operations.

4 Tabu Search for PFSP on CUDA

Porting Tabu Search method to CUDA architecture is presented in this section. This includes two methods for neighbourhood evaluation, which can be employed in any population-based metaheuristic, and GPU implementation of tabu list, which can be used in other Tabu Search based algorithms.

As mentioned before, each iteration of the Tabu Search algorithm consists of four main tasks: neighbourhood generation, neighbourhood evaluation, choosing the best solution within a neighbourhood and updating the tabu list, and finally moving to the new solution.

Neighbourhood generation and moving to the next solution can be easily parallelised, provided permutations forming neighbourhood are all stored in memory. Subsection 4.1 provides more details on how this tasks are implemented on CUDA.

Initial profiler reports indicate, that neighbourhood evaluation is the most time consuming part of the whole algorithm (over 90% of total computation time), which complies with observations made by Janiak et al. [12]. Two methods for neighbourhood evaluation are proposed in this paper, and described in detail in subsections 4.2 and 4.3.

Two implementations of move selection, and tabu list management were considered in this study. In the first one, this task is simply done on CPU. The drawback is, it requires data transfer between GPU and CPU in every iteration, which may harm general performance. Therefore, second implementation is considered, in which tabu list, and selection of the next move is handled entirely by GPU. This way no data transfers between GPU and CPU are required during algorithm execution. Implementation details are presented in subsection 4.4.

4.1 Parallel neighbourhood generation

Each transposition move can be described by a pair of integers — positions of elements to be exchanged. While a neighbourhood contains all possible transposition moves, its size is constant

during algorithm's runtime. Moves are numbered, and correspond to consecutive permutations in the neighbourhood.

At the beginning of each iteration all permutations are the same (representing the current solution). This way, a neighbourhood can be generated by running the same number of threads, as permutations in the neighbourhood. Each thread is applying a move (elements exchange, requiring three assignment operations) to the corresponding permutation.

After each iteration, the same method may be applied, first to revert all permutations (after which they are equal to the initial one), and then apply the move chosen during iteration.

4.2 Sequential objective function evaluation

The simplest approach to parallelise computation of the objective function, is to evaluate all permutations in neighbourhood at the same time — one CUDA thread evaluates one permutation.

As stated in subsection 2.1, all $C_{i,j}(\pi)$ values (time of finishing j^{th} task in permutation π on machine number i) must be computed. There are mn $C_{i,j}(\pi)$ values for each permutation, therefore time complexity of sequential evaluation is $O(mn)$.

4.2.1 Global memory access coalescing

One of the bottlenecks in CUDA platform, is very slow access to global memory. Since a neighbourhood can be quite large (up to 250 MB for datasets with 500 tasks), all permutations are stored in global memory. During evaluation, each position of each permutation must be read at least once, resulting in significant amount of read operations.

To overcome the problem of slow global memory access, NVidia introduced a mechanism called *coalescing*. It allows the reading of several data cells in a single operation, but only if certain requirements are met. For details, please refer to Programming Guide by NVidia [20].

During evaluation all threads start with reading first the elements in their respective permutations, then the second element, and so on. In order to meet the above-mentioned requirements for coalescing, permutations must be stored in non-intuitive way — the first elements of all permuta-

tions are stored first, then second elements, and so on. Figure 1 illustrates the intuitive method for how permutations can be stored in global memory, and the one considered in this paper. Only the second approach allows coalesced reads. Full advantage of coalescing mechanism is taken, as long as number of threads per block is divisible by 16.

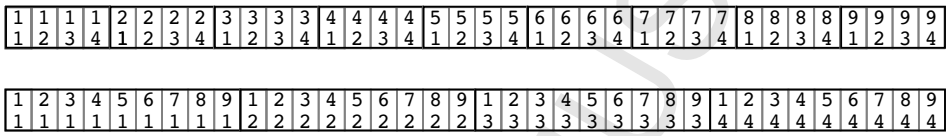

Figure 1: Two ways of storing permutations in 1-D array. First one represents intuitive approach — permutations are stored one after another. In second one, first elements of all permutations are stored first, then second elements, and so on. Top number is permutation number in neighbourhood, and bottom one a number of the element in permutation

4.2.2 Shared memory utilisation

The naive approach to evaluation, would be to create a 2D array of $C_{i,j}(\pi)$ values, and compute each cell in an order determined by data dependencies. This way each permutation would require $O(mn)$ memory, e.g. 60 kB for datasets of size 500×30. Therefore, $C_{i,j}$ array would not fit into shared memory, forcing global memory usage, considerably decreasing performance.

Fortunately, data dependencies allow row- or column-wise evaluation of $C_{i,j}$ array. Therefore, only one row (column) must be stored in memory at a time, resulting in $O(m)$ or $O(n)$ memory complexity. This way there is no need to use global memory for computing $C_{i,j}(\pi)$ values.

Implementation with array of length m is chosen, resulting in up to 120 bytes required for each permutation, for largest datasets with 30 machines. Evaluation starts by filling $C_{i,j}$ array with $C_{i,\pi(1)}$ values, and iterates n times for next elements of the permutation.

4.2.3 Avoiding shared memory bank conflicts

In j^{th} iteration, while computing $C_{i,\pi(j)}$ values, each thread first computes $C_{1,\pi(j)}$, then $C_{2,\pi(j)}$, and so on. Shared memory is organised into 16 identical banks of memory, which can be accessed

simultaneously by threads in the same half-warp. To avoid bank conflicts, shared memory is organised in similar fashion to global memory — $C_{1,j}$ values are stored first, then $C_{2,j}$, and so on.

4.2.4 Constant memory utilisation

During evaluation, processing times ($p_{i,j}$ values) are frequently accessed without any predictable access pattern. Therefore, it would be best to store them in shared memory. While processing times are integers between 1 and 99, one byte is sufficient for each value. Still, for datasets of size 500×30, there are 15,000 $p_{i,j}$ values, requiring 15,000 bytes of memory. If stored in shared memory, there would not be enough space for $C_{i,j}$ arrays.

On the other hand, the processing times matrix can be stored in constant memory space, effectively utilising its cache. There is a drawback though. Even that constant memory space is 64 kB, physical cache is only 8 kB. Therefore, performance for larger datasets (where nm product is greater than 8192) will suffer from constant memory cache misses. This effect is fairly limited though, while all threads are accessing processing times of tasks on the same position, and any two permutations within neighbourhood differ only on two positions.

4.3 Parallel objective function evaluation

Apart from evaluating permutations in parallel, an additional source of parallelism is available. Due to the theoretical properties of PFSP, described e.g. by Bożejko [1], limited parallelisation of evaluation of even just one permutation is possible.

Dependencies between $C_{i,j}(\pi)$ values allow parallel evaluation of diagonals. Figure 2 illustrates possibility of parallel objective function evaluation, for simple PFSP instance with 4 tasks and 3 machines. Arrows show the dependencies between $C_{i,j}(\pi)$ values, and different shades of grey show values which can be evaluated at the same time. Evaluation starts with top left value known (the darkest dot). For makespan criterion, bottom right value must be known (the lightest shade of grey dot), and for total flowtime all bottom values must be known.

This approach reduces time complexity to $O(m + n)$. To achieve this m CUDA threads are

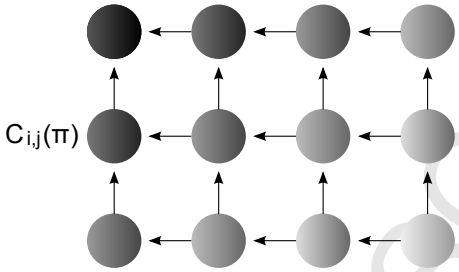

Figure 2: Parallel objective function evaluation for PFSP. Arrows show dependencies between $C_{i,j}(\pi)$ values. Values which can be evaluated in parallel have the same shade of grey

required. Therefore, the total evaluation cost for each permutation is $O(nm + m^2)$. A parallel approach to evaluation requires number of threads equal to a product of neighbourhood size and the number of machines m.

4.3.1 Global memory access coalescing

In parallel evaluation, in order to take full advantage of coalescing mechanism, data must be organised in the same way as in sequential evaluation, but this time the number of permutations evaluated per block must be divisible by 16.

If this requirement is not met, partial coalescing is still possible. Let k denote the number of permutations evaluated in each block. On GPUs with compute capability 1.1, reads for the first $k - (k \bmod 16)$ permutations will be coalesced into groups of 16 values at a time. Remaining $k \bmod 16$ values will be read one at a time. Therefore, each block must evaluate at least 16 permutations, in order to get any coalesced reads on devices with compute capability 1.1.

Similarly, the first $k - (k \bmod 16)$ permutations will be coalesced into groups of 16 values at a time on devices with compute capability 1.3. The main difference is, that the remaining $k \bmod 16$ values will be read in up to two groups of up to 8 values at a time. Therefore, partial coalescing is always achieved on devices with compute capability 1.3.

4.3.2 Shared memory utilisation

Threads working on the same permutation, communicate with each other by accessing the same memory (array containing $C_{i,j}(\pi)$ values on the current diagonal). Therefore, diagonals are stored in *shared memory*. While computing values on diagonal is performed in parallel, in order to prevent write-read conflicts, two arrays are required — one for the previous and another for the current diagonal. As a result, 8 bytes of shared memory is required for each thread.

4.3.3 Avoiding shared memory bank conflicts

Each thread is accessing only two cells (depending on number of corresponding machine) of both arrays alternatingly. All threads in the same block, corresponding to the same machine, access arrays simultaneously. Therefore, cells for machine 1 are at the beginning, followed by cells for machine 2, and so on. To take full advantage of parallel reads from shared memory, the number of permutations evaluated per block, again must be divisible by 16.

4.3.4 Constant memory utilisation

The processing times array is stored in constant memory in the same way as in the sequential evaluation. Though, in the parallel approach, this array is accessed in a much more erratic way, causing many more cache misses for larger datasets ($nm > 8192$). This is due to the fact, that in the sequential evaluation, in all permutations processing times for tasks on the same position are read. In the case of parallel evaluation, processing times for tasks from up to m positions are read at the same time.

4.4 Tabu list and move selection on GPU

Tabu list on GPU is implemented as an array of integers, one for each possible move. Value equal to zero means, that particular move is not on the tabu list. A positive value represents the number of iterations that the corresponding move will stay on the tabu list.

After evaluation, the best possible move is selected. The only constraint is, that the move

is currently not on the tabu list, unless the move leads to solution better than the best known solution. To select the next move using the GPU, the following steps are taken:

1. Values greater or equal to the best known value, corresponding to moves on tabu list, are changed to the maximal value of the integer — this way they will not be picked during search for minimal value. This task can be done in a constant time, by a number of threads equal to neighbourhood size.

2. Index of minimal value in the neighbourhood is found.

3. Tabu list is updated — all values in tabu list array are decreased by one, unless they are already zero. Tabu list value corresponding to the next move, selected in step 2, is set to tabu list length. This task can be done in a constant time, by a number of threads equal to neighbourhood size.

4. If the value found in step 2 is smaller than the best known value, both value and corresponding permutation, are copied as the new best known solution.

The most critical part in the above algorithm, is step 2 — finding the minimum. It is implemented as follows:

1. All the values are divided into blocks of size equal to number of threads per block on GPU.

2. Each block of threads finds minimal value, and its index within the neighbourhood, in the corresponding block. This is done using tournament method of comparison between the values, thus computational complexity is $O(\log M)$, where M is a block size.

3. Each block writes index of the minimal value into new array. If this array has more than one element, go to step 1, using the new array of minimal values, instead of the initial one.

The above algorithm makes $\lceil \log_M n \rceil$ loops, where n is the neighbourhood size.

5 Computational experiments

Extensive computational experiments were conducted in order to measure the performance of the CUDA implementation of Tabu Search, with both evaluation methods, and both methods of tabu list management. In subsection 5.1 the testbed configuration is described in detail. The following subsection (5.2) provides occupancy analysis for both evaluation methods, and two GPU cards considered in this study. Subsection 5.3 contain results of experiments conducted in order to find the optimal number of threads per block for each evaluation method. Then, subsection 5.4 contains a comparison between tabu list management on CPU and on GPU. Experiments on CUDA Tabu Search, on Tesla GPU hardware are presented in subsection 5.5. Finally, subsection 5.6 contain comparison of performance of CUDA Tabu Search, against GPU implementation of Tabu Search by Janiak et al. [12].

5.1 Testbed configuration

CPU implementation of Tabu Search was tested on Intel Xeon 3.0 GHz processor with 2 GB memory. Experiments on CUDA algorithms were carried out on computer equipped with an NVidia Tesla C1060 GPU (4 GB dedicated memory), Intel Xeon 2.8 GHz and 16 GB memory. Additional experiments were conducted on a laptop equipped with an NVidia Quadro FX 570M (256 MB dedicated memory), Intel Core 2 Duo 2.5 GHz and 4 GB memory.

Tesla C1060 contain 30 multiprocessors with 8 scalar processor cores each. It has compute capability 1.3. The peak performance for single precision operations is 933 GFlops, and for double precision operations 78 GFlops. Quadro FX 570M contain 4 multiprocessors, also with 8 cores each. It has compute capability 1.1.

The most commonly used benchmark suite for Permutation Flowshop Scheduling Problem, is a set of 120 datasets proposed by Taillard [22]. For the purpose of experiments presented in this article, additional random datasets were generated, using the same method as described by Taillard [22]. As a result, benchmark suite contains 600 datasets — 10 datasets for each combination of number of tasks $n \in \{10, 20, 40, 50, 60, 100, 120, 200, 300, 500\}$, and machines $m \in$

$\{5, 10, 15, 20, 25, 30\}$.

5.2 GPU occupancy analysis

In sequential evaluation each thread evaluates one permutation. This would result in 3072 permutations evaluated in parallel on Quadro FX 570M (30,720 on Tesla C1060). Unfortunately, small shared memory size (16 kB per multiprocessor) does not allow us to reach these numbers. For datasets with only 5 machines, each thread requires 20 bytes of shared memory — this limits number of threads running on each multiprocessor to 819 (768 in practice). The number of threads, which can be executed on each multiprocessor at the same time is decreasing with increasing number of machines. As a result, for datasets with 30 machines, each thread requires 120 bytes, effectively limiting number of threads per multiprocessor to 136 (128 in practice).

Each thread performing sequential evaluation requires 13 registers, for both C_{max} and C_{sum} criteria. This is not a problem on Tesla C1060, but on Quadro FX 570M it reduces maximal number of threads per multiprocessor to 576.

Taking all these factors into consideration, for datasets with $m = 5$, at most 2304 permutations (75% occupancy) can be evaluated on Quadro FX 570M card, and 23,040 (75% occupancy) on Tesla C1060. For datasets with $m = 30$, respective limits are 512 (16.7% occupancy), and 3840 permutations (12.5% occupancy).

In the case of parallel evaluation, each permutation requires m threads. Maximal number of permutations evaluated in parallel varies between 1020 ($m = 30$) and 6124 ($m = 5$), on Tesla C1060, and between 100 ($m = 30$) and 612 ($m = 5$) on Quadro FX 570M.

Fortunately, this time small shared memory size is not an issue, as only 8 bytes of memory is required for each thread (8 kB total when running all 1024 threads). Also, demand on registers is smaller — 10 registers for makespan criterion, and 11 for total flowtime criterion. This limits the maximal number of threads per multiprocessor on Quadro FX 570M, but only for total flowtime criterion (to 640 threads per multiprocessor).

Taking all these factors into consideration, for datasets with 5 machines, at most 612 (512

for C_{sum}) permutations (100% and 83% occupancy, respectively) can be evaluated at the same time on Quadro FX 570M card, and 6124 (100% occupancy) on Tesla C1060. For datasets with 30 machines respective limits are 96 for C_{max}, and 80 for C_{sum} (100% and 83% occupancy, respectively), and 960 permutations (100% occupancy).

5.3 Choosing number of threads per block

Initial experiments showed, that performance of both evaluation methods significantly depends on number of threads per block. Therefore, all reasonable configurations were tested using profiler provided by NVidia in CUDA Toolkit 2.2.

In sequential evaluation, number of threads per block can be any positive integer up to 512 (unless there is not enough shared memory for all threads). Yet, each block is divided into warps of size 32 threads. As a result, number of threads not divisible by 32, would simply waste some computational power. Therefore, only numbers divisible by 32 were considered for sequential evaluation in this experiment.

In the case of parallel evaluation, the number of threads must be divisible by the number of machines, and preferably also divisible by 32. Fortunately, both requirements can be satisfied for all datasets, except those with 25 machines (least common multiple of 25 and 32 is 800). Anyway, both requirements are satisfied for only a few block sizes, therefore all multiples of number of machines, greater than 32, were considered for parallel evaluation in this experiment.

This experiment was conducted on all 600 datasets on Tesla C1060, and on 480 datasets (excluding datasets with 300 and 500 tasks) on Quadro FX 570M. For each dataset, complete neighbourhood was generated and evaluated. Results were checked against sequential method to confirm their validity. While the goal of this experiment, was to choose optimal number of threads per block, only evaluation time was measured. Evaluation for total flowtime criterion, took up to 6% more time than for makespan criterion, but trends remained the same. Thus, only results for C_{max} criterion are presented.

While number of machines is influencing demand on shared memory, which is a bottleneck in

sequential evaluation, results are grouped in respect to this parameter. Times for 10 datasets of each size are averaged, and then averaged among all datasets with the same number of machines. For datasets with 30 machines, maximal number of blocks, divisible by 32, is 128. While for each dataset size, at least one configuration with ≤ 128 threads per block is in best 3 configurations, only these configurations are presented. Figure 3 and 4 show results of this experiment on Tesla C1060 and Quadro FX 570M GPUs, respectively.

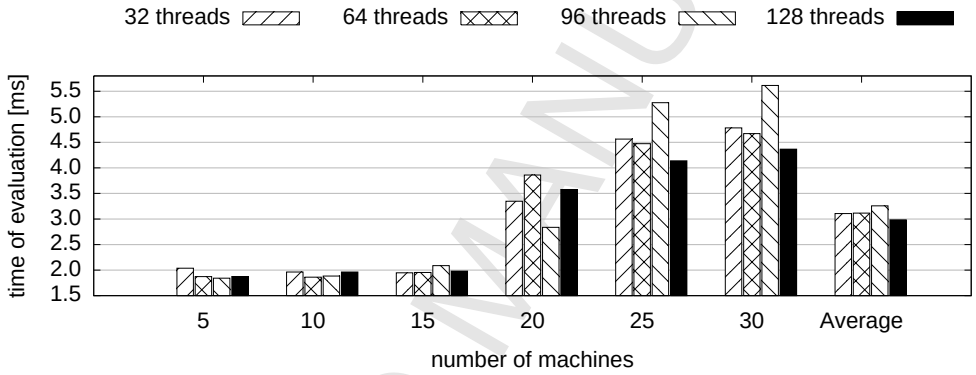

Figure 3: Influence of number of threads on sequential evaluation time on Tesla C1060

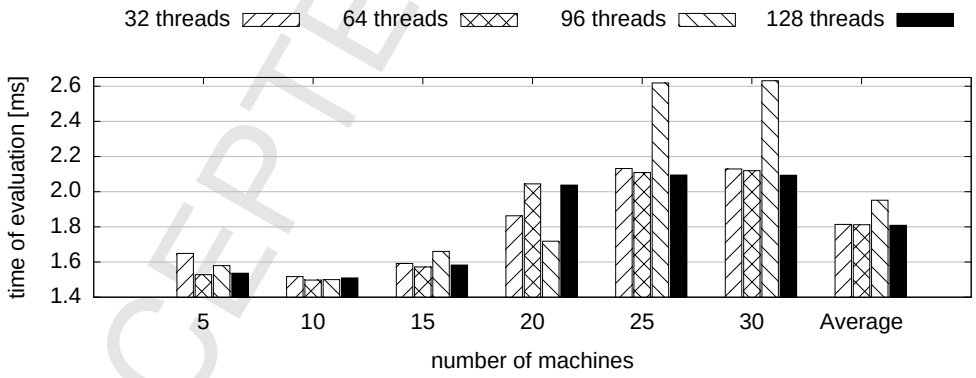

Figure 4: Influence of number of threads on sequential evaluation time on Quadro FX 570M

It can be easily observed, that on both GPUs, evaluation times for 32, 64 and 128 threads are very similar, disregarding number of machines. Results for configuration with 96 threads are more diverse — for 20 machines it is clearly the best configuration, but for 25 and 30 machines the worst one. Occupancy analysis provides explanation to this phenomenon. On Tesla C1060 (Quadro FX

570M), for datasets with 20 machines, respective occupancies are: 15.6% (20.8%) for 32 threads, 12.5% (16.7%) for 64 and 128 threads, and 18.8% (25%) for 96 threads. For datasets with 25 and 30 machines, occupancy for 32, 64 and 128 threads is 12.5% (16.7%), but for 96 threads only 9.4% (12.5%). Nevertheless, on both GPUs, on average the best choice is 128 threads per block.

Also, results for parallel evaluation are grouped with respect to number of machines, while it affects number of permutations evaluated per block, thus total number of blocks required to evaluate whole neighbourhood, as well as possibility of accessing global memory in a coalesced way. Times for 10 datasets of each size are averaged, and then averaged among all datasets with the same number of machines. For each number of machines, the best three configurations are presented. Figure 5 and 6 show results of this experiment on Tesla and Quadro GPUs, respectively.

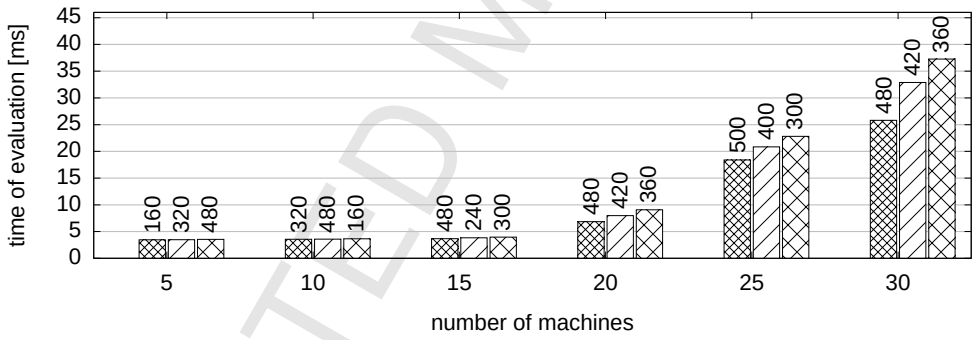

Figure 5: Best configurations (number of threads per block above each bar) of parallel evaluation on NVidia Tesla C1060

Results for parallel evaluation confirm that taking advantage of coalesced global memory accesses is crucial to parallel evaluation performance. As stated in subsection 4.3, full coalescing is possible only when k (number of permutations evaluated in each block) is divisible by 16. This can be especially observed in results for Quadro FX 570M (compute capability 1.1), where number of threads in optimal configuration is always divisible by $16m$. On Tesla, this does not apply to datasets with 25 machines, where benefits from higher k (less cache misses in accessing processing times array), are greater than those from full coalescing.

To summarise, number of threads equal to $16m$ (except for $m = 5$, where it is $32m$) is the best

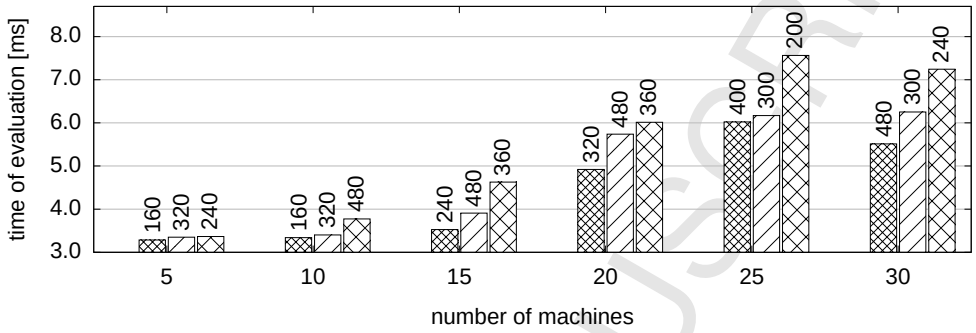

Figure 6: Best configurations (number of threads per block above each bar) of parallel evaluation on NVidia Quadro FX 570M

choice on Quadro FX 570M. On Tesla C1060 it is best to choose $32m$ threads whenever possible ($m \leq 15$). For datasets with 20 and more machines, optimal choice would be $16m$, but only for problems which do not suffer constant memory cache misses (occur when $nm > 8192$). For the largest datasets, benefits from smaller number of cache misses are greater than those from full coalescing.

5.4 Tabu list management and move selection

In order to compare two methods of tabu list management and move selection, described in subsection 4.4, two CUDA implementations of Tabu Search, were run for each dataset on Tesla C1060, and for up to 120 tasks on Quadro FX 570M, because of its limited memory. Algorithm was run with sequential evaluation for 1000 iterations, and 10 runs were made for each dataset.

Complexity of tabu list management and move selection depends only on the number of tasks, therefore execution times were averaged in respect to this parameter. Results are shown on figure 7, in form of relative speed-up gained due to implementation of those tasks on GPU ($\frac{time_{CPU} - time_{GPU}}{time_{CPU}}$). Negative values indicate, that CPU version is faster.

It can be easily seen that Tesla GPU is not fully utilised for less than 120 tasks. Above that number of tasks, speed-up stays positive. Unfortunately, on Quadro 570M parallelisation overheads are greater than gains, resulting in performance worse than CPU version. These results

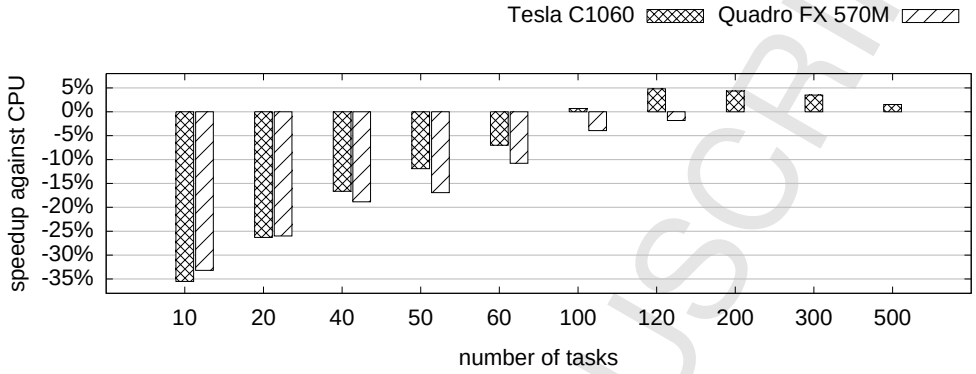

Figure 7: Performance of two implementations of tabu list management and move selection. Value indicate how much faster is implementation with those tasks done on GPU

were confirmed to be statistically significant by sign and Wilcoxon matched pairs tests. As a result, regardless of hardware, CPU version is significantly faster when number of tasks is smaller than 100.

The greatest speed-up (about 5%) is observed for datasets with 120 tasks. Above that number of tasks, speed-up is slowly decreasing, because fraction of computation time spent on neighbourhood evaluation is significantly increasing. Using GPU implementation of tabu list management and move selection results in execution time reduced by up to 5% for larger datasets.

5.5 Performance of CUDA implementation of Tabu Search on Tesla

This subsection presents experiments conducted in order to assess benefits from porting Tabu Search method on GPU. Performance of CUDA implementation of Tabu Search, with GPU tabu list management and move selection, and both evaluation methods, was compared against performance of CPU implementation.

Wall clock execution times were measured using `gettimeofday()` routine. Tabu Search was run for 1000 iterations. For both evaluation methods, number of threads per block was chosen according to guidelines presented in subsection 5.3.

For each out of 600 datasets, 10 runs were conducted. For each problem size, measured times were averaged. Speed-up is defined as a ratio of total execution times on CPU and GPU

($time_{CPU}/time_{GPU}$). Speed-up value greater than 1, indicates that the GPU implementation is faster.

Speed-ups for each problem size are presented on figures 8 (sequential evaluation), and 9 (parallel evaluation). Exact speed-up values, for both evaluation methods, are summarised in table 1. Again execution times for C_{max} and C_{sum} criteria, are almost the same, therefore only results for makespan criterion, are presented.

In order to confirm statistical significance of observations based on those results, appropriate statistical tests were made. Null hypothesis is that there is no significant difference between compared values, whereas the alternative is that the difference is statistically significant. All tests were made at significance level $\alpha = 0.05$.

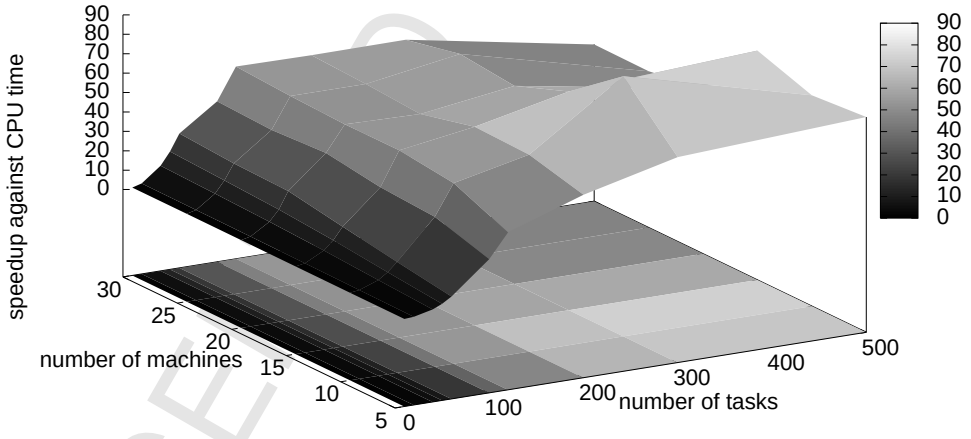

Figure 8: Speed-ups of CUDA implementation of Tabu Search with sequential evaluation

GPU implementation, with both evaluation methods, is slower than CPU, for all datasets with 10 tasks, and datasets of size 20×5 and 20×10. For all other datasets GPU implementation is faster. These observations are confirmed to be significant by sign and Wilcoxon tests.

Disregarding number of machines, speed-ups of CUDA Tabu Search with both evaluation meth-

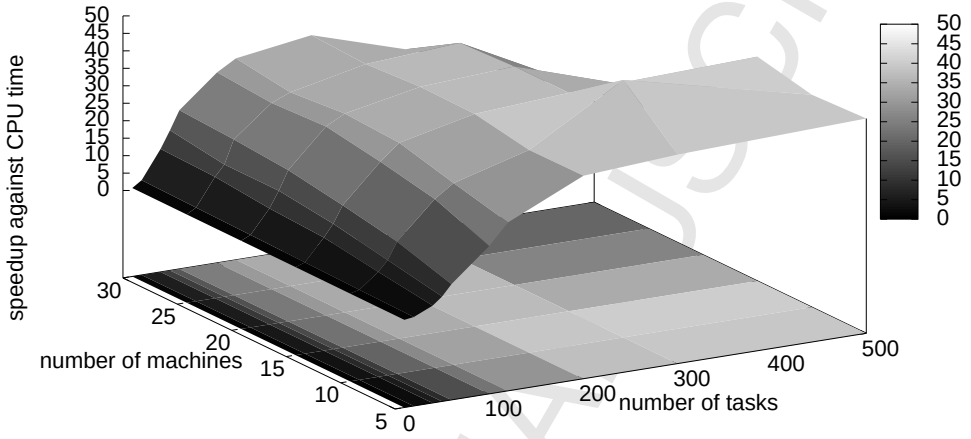

Figure 9: Speed-ups of CUDA implementation of Tabu Search with parallel evaluation

ods are increasing rapidly, with increasing number of tasks, up to 100 tasks, when neighbourhood size exceeds maximum number of permutations evaluated at the same time. After this threshold, speed-ups continue to grow, until nm product exceeds 8192, and constant memory cache misses start to occur. Further increase in number of tasks results in decrease in speed-ups. Degradation of performance is significantly more severe in case of parallel evaluation, due to larger number of cache misses.

number of tasks	number of machines 5		10		15		20		25		30	
	par	seq	par	seq	par	seq	par	seq	par	seq	par	seq
10	0,10	0,10	0,14	0,14	0,18	0,18	0,22	0,22	0,26	0,25	0,31	0,29
20	0,46	0,44	0,77	0,71	1,11	1,02	1,47	1,27	1,77	1,46	2,12	1,64
40	2,87	2,68	5,01	4,28	7,35	6,02	8,17	7,37	9,72	8,35	10,20	9,17
50	5,04	4,81	7,95	7,58	10,85	10,57	12,82	12,86	12,82	14,38	15,11	15,74
60	8,16	7,76	12,80	12,08	15,72	16,74	18,27	21,62	19,55	22,31	20,40	24,24
100	17,33	23,89	23,27	32,18	26,46	33,94	29,05	38,08	28,25	36,45	29,51	37,67
120	23,22	36,12	26,95	48,02	31,01	51,40	31,85	51,26	30,43	52,37	32,65	54,05
200	33,12	49,27	32,99	56,42	35,79	57,29	36,05	50,33	35,05	52,30	35,81	53,65
300	34,84	60,86	48,69	89,01	37,48	67,63	37,56	57,02	36,77	59,96	27,46	53,83
500	36,48	65,92	35,79	63,58	39,17	73,10	20,04	40,18	9,15	35,52	6,37	35,79

Table 1: Speed-ups of Tabu Search with parallel (par) and sequential (seq) evaluation methods. Horizontal line represents threshold, beyond which constant memory cache misses occur

Kruskal-Wallis test confirms, that speed-ups are increasing significantly, with increasing number of tasks, up to 200 for parallel evaluation, and up to 300 for sequential evaluation. Average speed-up for parallel evaluation, for 300 tasks is larger than for 200 tasks, but not significantly ($p = 0.689$). For 500 tasks, speed-ups are significantly smaller — back to average for 200 tasks in case of sequential, and to average for 120 tasks in case of parallel evaluation.

Speed-ups are also increasing, with increasing number of machines, up to 15 machines. Above that number, speed-ups remain stable, until $n \times m$ exceeds 8192. A Kruskal-Wallis test was conducted in order to confirm those observations. Indeed, for parallel evaluation, a statistically significant increase in speed-ups is observed with increasing number of machines, up to 15 machines. Above that speed-ups remain stable — differences are not statistically significant.

For sequential evaluation, speed-ups increase significantly from 5 to 10 machines. For 15 machines speed-ups are slightly better, but not significantly ($p = 0.189$). For 20 machines speed-ups decrease slightly, but significantly ($p = 0.046$). This is due to drop in occupancy from 25% for 15 machines, to 12.5% for 20, 25, and 30 machines. This drop is caused by increasing demand for shared memory, with increasing number of machines. Above 20 machines speed-ups remain stable.

The best speed-ups are observed for datasets with 300 tasks and 10 machines — over 89× for sequential evaluation, and over 48× for parallel evaluation.

Sign and Wilcoxon matched pairs test confirm, that parallel evaluation performs significantly better than sequential, for datasets with up to 50 tasks. For larger datasets, sequential evaluation performs significantly better. This is mainly due to the fact, that sequential evaluation requires more permutations (almost 4000 against 960 for parallel evaluation) to fully utilise all Tesla C1060 multiprocessors.

Advantages of the sequential approach increase, with increasing problem size. This is caused by overheads in parallel evaluation — the cost of evaluating one permutation is $O(nm)$ in the sequential case, and $O(nm + m^2)$ in parallel evaluation. Another reason is, that once the nm product exceeds 8192, there are many more cache misses in parallel evaluation.

5.6 Comparison with existing GPU implementation

In 2008, Janiak et al. proposed another GPU implementation of Tabu Search (it will be referred to as TS-JJL). Results of experiments conducted in order to compare performance of TS-JJL, and two methods proposed in this paper (they will be referred to as TS-CB-S for sequential, and as TS-CB-P for parallel evaluation), are presented in this subsection.

For neighbourhood evaluation TS-JJL follows the same parallelisation scheme as TS-CB-S: one thread is evaluating one solution. Other tasks — tabu list management and move selection — in TS-JJL are done on CPU, thus in the experiments, TS-CB-S and TS-CB-P were run with those tasks performed on CPU as well. The main difference between implementations is the memory management: TS-JJL uses texture memory to store all the data, while both TS-CB algorithms use cached constant memory to store instance data, global memory to store the neighbourhood, and on-chip shared memory for local computation.

TS-JJL was implemented in C# using Microsoft XNA framework. Algorithms presented in this paper were implemented in C++, therefore direct comparison of execution times is not informative. Still, comparison of speed-up values can be reliable, provided testing platforms yield comparable computational power.

TS-JJL was tested on two platforms. First one was equipped with Intel Celeron 2.9 GHz, 1.5 GB RAM, and two GPU cards — NVidia GeForce 7300 GT (8 pixel shader and 4 vertex shader units, denoted as G8), and NVidia GeForce 8600 GT (32 unified shader units, denoted as G32). Second platform consisted of an Intel Xeon 2.4 GHz, 4 GB RAM, and NVidia GeForce 8800 GT (112 unified shader units, denoted as G112). To provide fair comparison with speed-ups for G32, experiments were conducted using NVidia Quadro FX 570M (also equipped with 32 unified shader units). Furthermore, speed-ups of TS-CB algorithms were computed against Intel Xeon 3.0 GHz, which is much faster than Intel Celeron 2.9 GHz.

TS-JJL was tested on a set of random datasets — 10 datasets for every combination of number of tasks $n \in \{10, 40, 60, 100, 120\}$ and number of machines $m \in \{5, 10, 15\}$. TS-CB implementations were tested on datasets of corresponding sizes. All algorithms were stopped after 1000

iterations. Speed-ups averaged for datasets of the same size, for different number of tasks (X axis), and number of machines, are presented on figure 10.

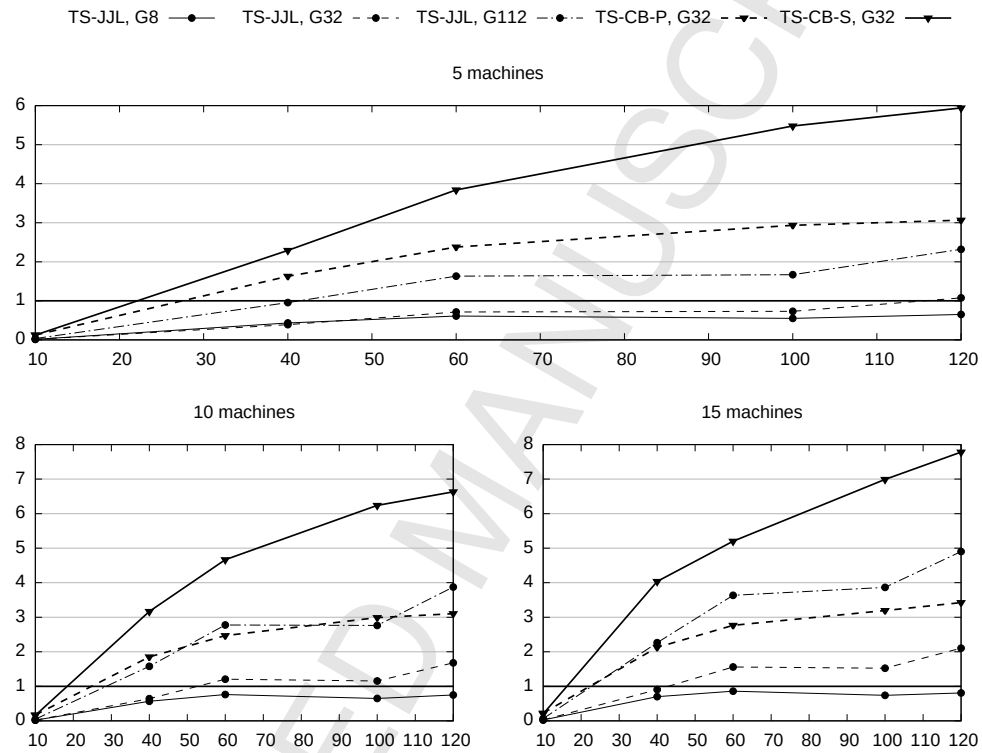

Figure 10: Comparison of speed-ups obtained by Tabu Search implementations on GPU by Janiak et al. [12], and one presented in this paper, with both evaluation methods

For GPUs with 32 cores, it is clear, that for all problem sizes, both TS-CB implementations, significantly outperform implementation by Janiak et al. [12]. Furthermore, TS-CB-S outperforms TS-JJL even when the latter is run on much faster GPU with 112 cores. This is also the case for TS-CB-P for datasets with 5 machines. All the above observations are confirmed to be statistically significant by sign and Wilcoxon matched pairs tests. To give some idea on absolute execution times, for largest datasets (120×15) TS-JJL requires 53.5 seconds on G112, while TS-CB-P and TS-CB-S require 13.7 and only 6 seconds, respectively, on GPU with 32 cores.

As TS-JJL and TS-CB-S follow the same parallelisation scheme for neighbourhood evaluation, superior performance of TS-CB-S is mainly caused by much more efficient use of memory. Also

TS-CB-P benefits from better memory management, but in this case increase in performance over TS-JJL is not as significant, because TS-CB-P suffers from a greater number of uncoalesced global memory reads. In fact, on NVidia Quadro FX 570M TS-CB-P performs worse than TS-CB-S for all problem sizes. This is not consistent with performance on Tesla C1060. This phenomenon is caused by the fact that coalescing on GPUs with compute capability 1.1 is very limited.

6 Conclusions

Two methods for neighbourhood evaluation, for Permutation Flowshop Scheduling Problem with makespan and total flowtime criteria, optimised for CUDA platform, were presented. Both methods can be used in most population-based metaheuristics, to accelerate their most time consuming part — neighbourhood evaluation. Both methods were employed in Tabu Search metaheuristic (TS-CB-S and TS-CB-P). Profiler experiments were conducted in order to find optimal configuration (number of threads per block) for each method. Apart from neighbourhood evaluation, neighbourhood generation was also parallelised on GPU. CPU and GPU implementations of tabu list management and move selection were presented. Both TS-CB-S and TS-CB-P, are faster with CPU version for datasets with less than 100 tasks, but for larger problems, GPU version is faster by up to 5%.

For comprehensive performance evaluation, a benchmark suite comprising 600 random datasets was generated. This suite include datasets with number of tasks between 10 and 500, and number of machines between 5 and 30. Execution time of Tabu Search implementation on both, CPU (Intel Xeon 3.0 GHz) and GPU (NVidia Tesla C1060) were measured and compared. Statistical significance of observations made on the results, was confirmed with sign and Wilcoxon matched pairs tests. Both, TS-CB-S and TS-CB-P, are faster than CPU implementation, for datasets with 20 and more tasks. TS-CB-P is faster than TS-CB-S for datasets with up to 50 tasks. TS-CB-P runs up to 48 times, and TS-CB-S up to 89 times faster than CPU implementation.

Additional experiments were conducted on NVidia Quadro FX 570M GPU (32 cores), in order

to compare TS-CB implementations to the one proposed by Janiak et al. [12] (TS-JJL). Results clearly indicate, that both implementations proposed in this paper are significantly faster than TS-JJL, mostly due to better memory utilisation. Moreover, TS-CB-S is considerably faster, even when TS-JJL was run on much faster GPU, equipped with 112 cores. Statistical significance of those results was confirmed with sign and Wilcoxon tests.

Further research will be done, on how techniques used in implementations, presented in this paper, can be applied to speed-up computation in other discrete optimisation problems, and other population-based metaheuristics. Another interesting problem is, whether implementation at least as efficient as ones presented, can be prepared in OpenCL framework, which would allow computation on AMD hardware.

References

[1] W. Bożejko, Solving the flow shop problem by parallel programming, Journal of Parallel and Distributed Computing 69 (5) (2009) 470–481.

[2] W. Bożejko, M. Wodecki, Parallel genetic algorithm for the flow shop scheduling problem, Lecture Notes in Computer Science 3019 (2004) 566–571.

[3] W. Bożejko, M. Wodecki, Parallel scatter search algorithm for the flow shop sequencing problem, Lecture Notes in Computer Science 4967 (2008) 180–188.

[4] C.-S. Chung, J. Flynn, O. Kirca, A branch and bound algorithm to minimize the total flow time for m-machine permutation flowshop problems, International Journal of Production Economics 79 (3) (2002) 185–196.

[5] M. Czapiński, Parallel Simulated Annealing with Genetic Enhancement for flowshop problem with Csum, Computers & Industrial Engineering 59 (4) (2010) 778–785.

[6] X. Dong, H. Huang, P. Chen, An iterated local search algorithm for the permutation flowshop problem with total flowtime criterion, Computers & Operations Research 36 (5) (2009) 1664–1669.

[7] M. R. Garey, D. Johnson, R. Sethi, The complexity of flowshop and jobshop scheduling, Mathematics of Operations Research 1 (2) (1976) 117–129.

[8] F. Glover, Tabu search — part I, ORSA Journal on Computing 1 (3) (1989) 190–206.

[9] F. Glover, Tabu search — part II, ORSA Journal on Computing 2 (1) (1990) 4–32.

[10] J. Grabowski, M. Wodecki, A very fast tabu search algorithm for the permutation flow shop problem with makespan criterion, Computers & Operations Research 31 (11) (2004) 1891–1909.

[11] M. Ishubuchi, S. Masaki, H. Tanaka, Modified simulated annealing for the flow shop sequencing problems, European Journal of Operational Research 81 (1995) 388–398.

[12] A. Janiak, W. Janiak, M. Lichtenstein, Tabu search on GPU, Journal of Universal Computer Science 14 (14) (2008) 2416–2427.

[13] B. Jarboui, M. Eddaly, P. Siarry, An estimation of distribution algorithm for minimizing the total flowtime in permutation flowshop scheduling problems, Computers & Operations Research 36 (9) (2009) 2638–2646.

[14] S. M. Johnson, Optimal two- and three-stage production schedules with setup times included, Naval Research Logistics Quarterly 1 (1) (1954) 61–68.

[15] X. Li, Q. Wang, C. Wu, Efficient composite heuristics for total flowtime minimization in permutation flow shops, Omega 37 (1) (2009) 155–164.

[16] J. Liu, C. R. Reeves, Constructive and composite heuristic solutions to the $P||\sum C_i$ scheduling problem, European Journal of Operational Research 132 (2) (2001) 439–452.

[17] T. V. Luong, N. Melab, E.-G. Talbi, Parallel local search on GPU, Tech. rep., Centre de recherche INRIA Lille (2009).

[18] M. Nawaz, E. E. Enscore Jr, I. Ham, A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem, Omega 11 (1) (1983) 91–95.

[19] E. Nowicki, C. Smutnicki, A fast tabu search algorithm for the permutation flow-shop problem, European Journal of Operational Research 91 (1) (1996) 160–175.

[20] NVidia, NVidia CUDA^{TM}. Programming guide. Version 2.3, `http://www.nvidia.co.uk/object/cuda_get_uk.html` (2009).

[21] C. Rajendran, H. Ziegler, Two ant-colony algorithms for minimizing total flowtime in permutation flowshops, Computers & Industrial Engineering 48 (4) (2005) 789–797.

[22] E. Taillard, Benchmarks for basic scheduling problems, European Journal of Operational Research 64 (2) (1993) 278–285.

[23] M. F. Tasgetiren, Y.-C. Liang, M. Sevkli, G. Gencyilmaz, A particle swarm optimization algorithm for makespan and total flowtime minimization in the permutation flowshop sequencing problem, European Journal of Operational Research 177 (3) (2007) 1930–1947.

[24] L.-Y. Tseng, Y.-T. Lin, A hybrid genetic local search algorithm for the permutation flowshop scheduling problem, European Journal of Operational Research 198 (1) (2009) 84–92.

[25] T. Yamada, C. R. Reeves, Solving the C_{sum} permutation flowshop scheduling problem by genetic local search, in: Proceedings of 1998 IEEE International Conference on Evolutionary Computation (1998), pp. 230–234.

[26] Y. Zhang, X. Li, Q. Wang, Hybrid genetic algorithm for permutation flowshop scheduling problems with total flowtime minimization, European Journal of Operational Research 196 (3) (2009) 869–876.

Michał Czapiński is currently a PhD Student at Cranfield University. He obtained MSc in computer science at University of Wrocław, Institute of Computer Science in 2008, MSc in Computational & Software Techniques in Engineering at Cranfield University, School of Engineering in 2009, and BSc in mathematics at University of Wrocław, Mathematical Institute in 2010. His research interests include distributed and parallel computing, discrete optimisation, and software engineering.

Stuart Barnes is currently a Research Fellow in the Applied Mathematics and Computing Group at Cranfield University. He received his BSc and MSc degrees in Physics from the University of Kent in 1991 and 1993 respectively. After spending 10 years working in the IT industry he obtained his PhD in Image Processing from Cranfield University and joined as a member of the research staff in 2005. Currently teaching in the fields of software engineering and parallel programming, his research interests include the use of parallel computational architectures to solve problems in machine vision and scientific computing.