Parallel Algorithms on Tree Decompositions

Glenn Stewart

Supervisors: prof. dr. H.L. Bodlaender, T.C. van der Zanden MSc

February 20, 2019

Abstract

Several problems including finding the maximum independent set and the minimum size dominating set of a graph are NP-hard problems that can be solved in linear time on graphs given with a tree decomposition whose width is bounded by a constant. So far, about all work on implementation of these algorithms was restricted to sequential algorithms, running on a CPU. A recent study showed that a GPU implementation of algorithms running on a path decomposition of a graph showed a significant speedup when compared to the CPU implementation.

Both the GPU implementation for maximum independent set and minimum dominating set show a significant increase in speed on a subset of all tested tree decompositions, but do have an equal or a decreased performance on other trees. The tree decomposition the algorithm is ran on has a large influence on the performance of the algorithms.

Contents

List of Figures

List of Tables

1 Introduction

Although Graphical Processing Units (GPUs) was originally meant for rendering computer graphics for video games, people found that they could be used for other tasks as well. Doing so is called *General Purpose Computing on Graphical Processing Units* (GPGPU). GPGPU has been used for many tasks, some of which include: Bioinformatics[18, 10], Fuzzy logic[1] and k-nearest neighbor algorithms[6].

The main feature that differentiates the GPU from the CPU (central processing unit) is the amount of tasks that can be computed in parallel. Solving a problem in parallel, when applied to the right problem, often results in a speedup of the original algorithm as you can solve parts of the problem simultaneously instead of sequentially. On a CPU this can be done to a certain extent with most CPUs having 4 cores, which can work on one thread per core (or two if the CPU has hyper-threading). In comparison a current generation high end consumer GPU has 57344 threads (based on the NVIDIA 1080TI). This means that if a portion of an algorithm can be ran in parallel in can be done effectively on the GPU. For both maximum independent set and dominating set the calculations are only dependent on previous values, not on values that are being calculated in the current step. This means all the calculations in a single step can be executed in parallel as they do not have to wait for other values to be calculated.

Tree decompositions have been used for a long time to solve certain NP-hard graph problems in polynomial time. The problems that we are solving on tree decompositions are problems that can have practical applications. Maximum independent set is a problem that has many real world applications. Examples of applications are code theory [4], bioinformatics [14], map labeling [19], computer vision [5], routing in road networks [8] and many others. Dominating set is also problem that needs to be solved in multiple applications. Scheduling [17], facility location problem [11] and routing [20] to name a few. When applied in practical environments having faster and more efficient algorithms are of course welcomed.

There are several other problems that can also be solved faster on a tree decomposition, if a treewidth of the tree is given, than on the graph. For the following problems treewidth is replaced by tw. A few of these problems are:

- Steiner Tree in $O(tw^{O(tw)})$, as described in chapter 7.3.3. in of Parameterised algorithms[2]
- k-Coloring can be solved in $O(k^{tw}tw^2|VertexofGraph|)$[13]
- Hamiltonian Cycle in $O((2^{tw}tw^{tw/2})^2tw^3|VertexofGraph|)$[13]

The objective of this thesis is to create a CPU and GPU implementation for both solving maximum independent set and minimum dominating set with a dynamic programming algorithm on tree decomposition. The CPU and GPU implementation will be compared to see if there is a significant difference in the computation time. The algorithms for solving maximum independent set and minimum dominating set both use a technique called dynamic programming. With this algorithm you solve subproblems which eventually lead to you solving the complete problem. Nobody has ever solved these problems with dynamic programming algorithms on tree decompositions implemented on the GPU. Plagmeijer[15] solved these problems on the GPU [15] using a path decompositions instead of tree decompositions. Plagmeijers implementation was a starting point for this project[1]. Path decompositions are a special case of tree decompositions; to lift algorithms that work on path decompositions to the tree decomposition case, additional work has to be done, in particular, code that handles join nodes has to be provided.

The biggest changes compared to the path decomposition implementation in both the CPU and GPU implementation are the addition of the code for the join bag and the changes made to the code where the bag specific function calls are made. In the case of path decomposition only two arrays are needed, one to keep the new values in and one for old values. With a tree decomposition you need at least two arrays, but join bags make it so you need to keep an additional array in memory. The code to delete old arrays, add new arrays, and make the next bag look at the correct array as its children is different than the path decomposition implementation.

1.1 Architectural GPU differences

The GPU code that has been written for this thesis uses C++ and the CUDA toolkit. The CUDA toolkit is developed by NVIDIA to let programmers use GPU accelerated computing. To be able to execute our code you need a CUDA-capable GPU. GPU's that are capable of executing CUDA code are GPU's made by NVIDIA, GPU's made by AMD will not be able to run the code.

Unfortunately, to compute the result of a problem on the GPU you cannot simple have CPU code and add: "Compute on GPU in parallel". The difference in thread count is due to the design philosophy of the GPU, which is different than that of the CPU. The CPU is optimized for low-latency while the GPU is optimized for throughput. Figure 1 shows this difference. The CPU has one large control unit (the component that directs the operation

[1]The author would like to thank R. Plagmeijer for sharing his implementation.

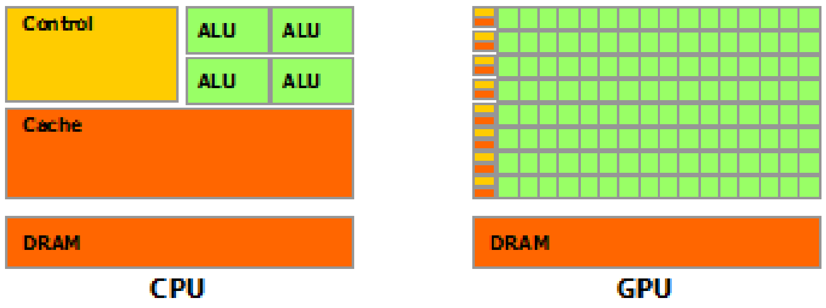


Figure 1: CPU vs GPU architecture[12]

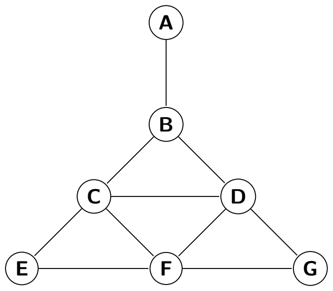
of the processor), a large cache (component that stores data so that future requests for that data can be served faster) and a small amount of complex ALUs (arithmetic logic units, the component used to perform arithmetic and logic operations). The GPU does not have a large cache and control component. Instead, 32 ALUs together share a single control unit, and 4 times 32 ALU's are bundles into a Streaming Multiprocessors which has a single cache. Although these ALUs are simple, by applying a parallel programming model and having many them they are able to generate a significant amount of power.

The programming model used by modern day GPU's is known as SIMT (Single Instruction, Multiple Threads). Each thread that is started must execute the same set of instructions but can be given different memory addresses to run these instructions on.

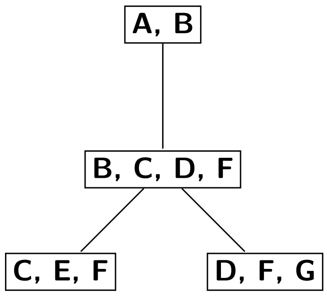
The phenomenon that occurs when two thread take different paths within a kernel is called branch divergence. This can happen at an if-condition when one thread fulfills the condition and another one does not. Branch divergence negatively affects the performance of the program. Because the threads in the GPU execute the same instructions at the same time, threads need to wait for code that is not part of their path to be computed.

Luckily not all threads have to execute the same code in NVIDIA GPU's. All threads are divided into blocks which are further divided into warps. Each block has shared memory between the threads and can contain a total of 1024 threads. Each warp will contain a subset of 32 threads of the total amount of threads in a block. The warps in each of the block exhibit SIMT execution. So instead of all threads having to execute the same instruction, only 32 threads have to. This lowers the chance of branch divergence as there are less threads that can diverge.

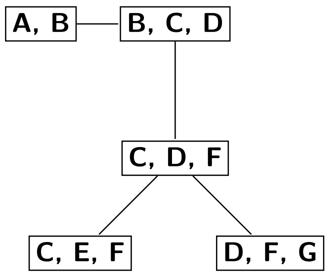
CUDA introduced two concepts to the SIMT model to support conditional execution and branching. The first concept is *predicated execution*. It means



(a) Input graph



(b) Tree Decomposition



(c) Tree Decomposition

that the result of a conditional instruction can be used to mask threads from execution subsequent instruction without branching. The other concept is Instruction replay/serialization. The idea behind this technique is the same, it is only executed differently. All threads execute all the branches of the conditionally executed code by replaying instructions, for instance replaying the if statement (true and then false). If a thread should execute the path for FALSE but is currently executing the path for TRUE, the thread is masked and will not write results, evaluate memory addresses and read operands.

1.2 Tree Decompositions

The algorithms that are implemented are not implemented directly onto the given graph $G = (V, E)$, but on a graphs given with a tree decomposition (TD). A tree decomposition of graph $G = (V, E)$ consists of a tree T and a subset $B_t \subseteq V$ associated with each node $t \in T$. Each of these subsets B_t bags of the tree decomposition. There are only a few conditions that need to hold for it to be a correct tree decomposition of a given graph [16]. The tree decomposition must satisfy the following properties:

1. $\bigcup_{i \in TD} B_i = V$. Each vertex from the original graph is contained in at least one bag

2. $\forall (u, v) \in E, \exists i \in TD : u, v \in B_i$. If vertices are adjacent in the graph, then they have a bag in common

3. if $v \in B_i$ and $v \in B_j$, then $v \in B_k$ for all k on the path from i to j in TD. The bags that contain v form a connected subset of TD

There can be several different tree decompositions which represent the same graph; for example a tree decomposition that consists of one single bag containing all vertices is a valid tree decomposition.

Although a graph can have many different correct tree decompositions, there is a way to restrict the complete set of tree decompositions to a subset of this set. This can be done by specifying a maximum treewidth. The treewidth of a tree decomposition is the size of the biggest bag $B_i - 1$.

1.3 Nice Tree decompositions

In a nice tree decomposition, instead of instantly adding/removing multiple nodes to/from a bag, we limit the amount transitions between bags. A nice tree decomposition is still a valid tree decomposition. The definition of nice tree decomposition that is used in in this paper was introduced by Cygan et al. [3]. The concept of a nice tree decompositions was introduced by Kloks [9]. A tree decomposition is nice if each of the bags is one of the following forms, as described in [2]:

- Leaf bag: An empty bag $B_l = \emptyset$ a leaf of the tree
- Root bag: An empty bag $B_r = \emptyset$ at the root of the tree
- Introduce bag: a bag with one child B_j such that $B_i = B_j \cup v$ for a vertex $v \notin X_j$. We say that v is *introduced* in B_j
- IntroduceEdge bag: a bag B_i with one child B_j such that $B_i = B_j$ and labeled with an edge $uv \in E(G)$ such that $u, v =\in B_i$. The edge uv is *introduced* in B_i
- Forget bag: a bag B_i with one child B_j such that $B_i = B_j \setminus \{v\}$ for a vertex $v \in B_j$. v is *forgotten* in B_i
- Join bag: a bag B_i with two children B_{i1} and B_{i2} such that $Bi = B_{i1} = B_{i2}$

For a tree decomposition to be nice it is not necessary to use the introduce edge bag, but using it has its benefits. When you do not use the introduce edge bag, every time a new vertex is introduced to the tree decomposition you have to simultaneously introduce all the other edges that connected it to vertices in B_i. By adding the edges one by one the description of the algorithm is often simplified. An additional requirement when using an introduce edge bag, is that every edge of E(G) can be introduced exactly one.

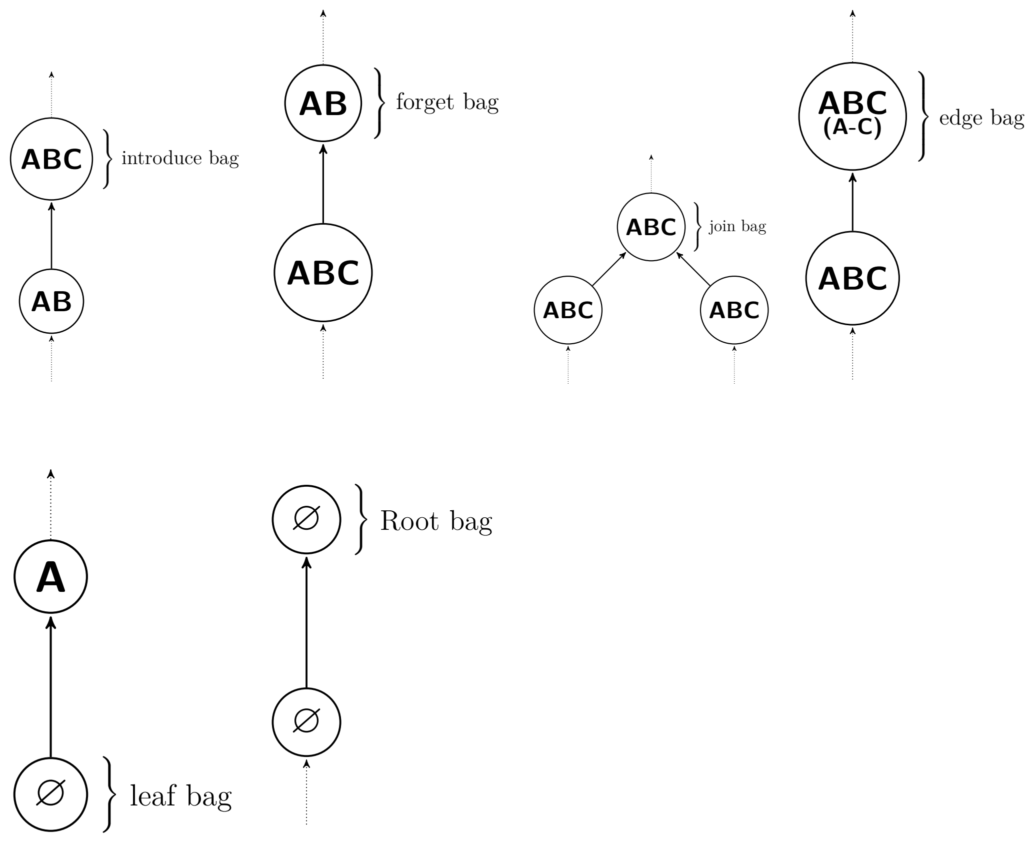


Figure 3: The different bag types

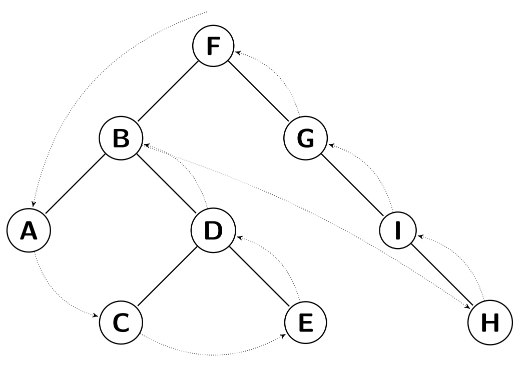


Figure 4: Post-order: A, C, E, D, B, H, I, G, F.

1.4 Algorithms

When solving problems using a (nice) tree decomposition, a dynamic programming algorithm is often the technique that is used. The dynamic programming algorithm will start at the leaves of the nice TD and traverse the tree until it reaches the root. The traversal of the tree decomposition will be done in a post-order traversal. This way the values of the subsets in both child bags of a join bag are known went we want to join them. An example of a post-order tree traversal can be seen in Figure 4. The arrows on the dotted line can be followed to see the path the post-order traversal takes.

Each bag in the tree decomposition contains a table of partial solutions. This table contains all the different subsets (partial solutions) and the value for each of these subset. To calculate the values for the table of the parent node the table of the child node is used (or two children if it is a join bag). At a leaf bag, the table is empty. Whilst traversing the tree to the root new vertices will introduced, resulting in a larger table than the child table, forgotten, resulting in a decrease in table size, and combined in the join bag where the table stays the same size. In the root node of the tree the table will contain one value as the optimal result, if it exists. In our implementation the bags only contain the value of the solutions, not the optimal solution itself. So in the root of the tree we have one value for the optimal solution instead of the optimal solution.

1.4.1 Maximum independent set

The first problem that will be solved is the maximum independent set. An independent set is a set of vertices in a graph in which no two vertices from the set are adjacent. A maximum independent set in a graph G is an independent set of the largest possible size in G. The maximum independent set problem asks for a given graph G, to determine what is the size of a maximum independent set. This problem is NP-hard [7]. The algorithm used to solve the maximum independent set in this paper has a complexity of $O(2^k k^{O(1)} n$, where k is the treewidth + 1 and n the number of nodes. The difference between this algorithm and finding the solution via brute force is that the exponential is different. The exponential when using brute force is n, which grows when the graph the tree decomposition is based on grows, whereas k does not grow when the graph gets bigger. This means that when you have a graph with bounded treewidth, this algorithm is a good alternative to brute force.

In the dynamic programming algorithm for maximum independent set, for each subset in the child bag of an introduce bag the newly added vertex of the introduce bag is either added to the subset or discarded. Both of these actions are possible which results in the doubling of the partial solutions in the child bag. Simply put, the table in each bag is size $2^{|B_i|}$. The calculations the algorithm performs for each subset in a bag is given below. A subset of bag B_i (a child bag) is indicated by S_i. B_{i+1} refers to the value of the bag in question, v and w indicates the vertex or edge that is used by the bag. The values calculated for bags when solving maximum independent set are defined as follows:

- Leaf bag: $B_{i+1}(S) = 0$
- Introduce bag (v): $B_{i+1}(S) = \begin{cases} B_i(S) & \text{if } w \notin S \\ B_i(S - \{v\}) + 1 & \text{otherwise} \end{cases}$
- Introduce edge bag (v, w): $B_{i+1}(S) = \begin{cases} -\infty & \text{if } v, w \in S \\ B_i(S) & \text{otherwise} \end{cases}$
- Forget bag (v): $B_{i+1}(S) = max\{B_i(S),\, B_i(S \cup \{v\})\}$
- Join bag: $B_{i+1}(S) = B_{j1}(S) + B_{j2}(S) - |S|$

We have made a slight change to this standard way of finding the maximum independent set. These are the following changes:

- Introduce bag (v): $B_{i+1}(S) = \begin{cases} B_i(S) & \text{if } w \notin S \\ B_I(S - \{v\}) & \text{otherwise} \end{cases}$
- Forget bag (v): $B_{i+1}(S) = max\{B_i(S),\, (B_i(S \cup \{v\}) + 1)\}$
- Join bag: $B_{i+1}(S) = B_{j1}(S) + B_{j2}(S)$

We now add one to the independent set at the forget bag instead of the introduce bag. By doing so we do not have to compute the size of the current independent set in the join bag. With the way we represent an independent set in a bag it would require quite a bit of computation to get the size of the independent set.

While weighted independent set is not the same as maximum independent set, only a few small changes have to be made to change it into maximum independent set. More information about the weighted independent set algorithm can be found in chapter 7.3.1 of Parameterized Algorithms[2].

1.4.2 Minimum dominating set

A minimum dominating set for a graph $G = (V, E)$ is a subset S of V such that every vertex that is not in S is the neighbour of at least one vertex that is in S and is the smallest size dominating set for the graph. This problem, just like the maximum independent set, is NP-hard.

For minimum dominating set the situation is more complicated than for maximum independent set. We have to choose between a vertex being in the dominating set, not in the dominating set and already dominated, and not in the dominating set and not yet dominated. To give this distinction we will give each bag B_i a coloring. A coloring is a mapping $f : B_i \to \{0, \hat{0}, 1\}$ which assigns one of these three colours to the vertices in the bag.

- White (0): if a vertex is coloured white it means that the vertex is not contained in the partial solution (dominating set), and must be dominated by a vertex of the final minimum dominating set
- Grey ($\hat{0}$): if a vertex is coloured grey it means that the vertex is not contained in the partial solution (dominating set), but it does not have to be dominated by a vertex in the current dominating set
- Black (1): if a vertex is coloured black it means that it is contained in the partial solution (dominating set)

This algorithm would not work if we just used white and black. We need the possibility to colour a vertex grey. The reason is that a vertex could be dominated by another vertex or edge that has not been introduced at that point in the tree decomposition. Therefore, solutions in which some vertices of the bag are not required to be dominated have to be considered, since these subproblems could be essential for constructing the best solution. Because there are three colours the amount of subsets of B_i is equal to $3^{|B_i|}$. The calculations the algorithm performs for each subset in a bag is given below.

$f|_S$ is a coloring f of B_i but restricted to the nodes in subset S. $f_{v\to a}$ is a colouring f where its color for vertex v is changed to colour a. Minimum dominating set:

- Leaf bag: $B_{i+1}(f) = 0$
- Introduce bag (v): $B_{i+1}(f) = \begin{cases} +\infty & \text{when } f(v) = 0 \\ B_i(f|_{S_i}) & \text{when } f(v) = \hat{0} \\ B_i(f|_{S_i}) + 1 & f(v) = 1 \end{cases}$
- IntroduceEdge bag (v, w): $B_{i+1}(f) = \begin{cases} B_i(f_{w\to\hat{0}}) & \text{if}(f(v), f(w)) = (1, 0) \\ B_i(f_{v\to\hat{0}}) & \text{if}(f(v), f(w)) = (0, 1) \\ B_i(f) & \text{otherwise} \end{cases}$
- Forget bag (v): $B_{i+1}(f) = min\{B_i(f_{v\to1}), B_i(f_{v\to0})\}$
- Join bag: $B_{i+1}(f) = \min_{f1,f2}\{B_{j1}(f_1) + B_{j2}(f_2) - |f^{-1}(1)|\}$

In the join bag we take the colouring with the minimum value of both the left and right child. These values have to be consistent with the coloring f in the join node as described in chapter 7.3.2 of Parameterized Algorithms[2]. The two child colourings are consistent with the coloring f if the following conditions hold:

- $f(v) = 1$ if and only if $f_1(v) = f_2(v) = 1$
- $f(v) = 0$ if and only if $(f_1(v), f_2(v)) \in \{(\hat{0}, 0), (0, \hat{0})\}$
- $f(v) = \hat{0}$ if and only if $f_1(v) = f_2(v) = \hat{0}$

The fact that it has to stay consistent makes the join bag for the dominating set algorithm costly to compute if the colouring in question contains a lot of zeroes. Each zero in the current colouring doubles the set of possible colourings that could have been used to compute the current colouring. So if the colouring contains ten zeroes, the amount of colourings to check are $2^{10} = 1024$ different colourings from the two bags before the join bag to know the value of one colouring in the join bag.

The same change to the maximum independent set has been made to minimum dominating set, changing the descriptions of the following bags:

- Introduce bag (v): $B_{i+1}(f) = \begin{cases} +\infty & \text{when } f(v) = 0 \\ B_i(f|_{S_i}) & \text{when } f(v) = \hat{0} \\ B_i(f|_{S_i}) & f(v) = 1 \end{cases}$

- Forget bag (v): $B_{i+1}(f) = min\{(B_i(f_{v\to 1}) + 1), B_i(f_{v\to 0})\}$
- Join bag: $B_{i+1}(f) = \min_{f1,f2}\{B_{j1}(f_1) + B_{j2}(f_2)\}$

2 Methods

In this thesis two different dynamic programming algorithms will be implemented, two on the CPU and two on the GPU, with variations for the GPU implementations. They will solve the maximum independent set and minimum dominating set problems on tree decompositions. The goal of the research is to get data on the duration these GPU implemented algorithms need for solving certain problems and compare these with the same algorithms implemented on the CPU solving the same problems. The implementation that is used to conduct these experiments is written in C++, and CUDA 9.1 was used for the code that was executed on the GPU.

2.1 Data

To make other researchers able of comparing their results against the results that were obtained with this implementation a data set is needed that is available for general use. This way implementations can be compared against one another on the same graph/tree decomposition. This is why the algorithms will be tested on data from the 2018 PACE challenge (The Parameterized Algorithms and Computational Experiments Challenge). The data can be found using the following link. The data for the challenge consists of a graph section, a terminals section and a tree decomposition section. For our application we will only use the graph and tree decomposition section of the data. A small grid was also constructed as an extra check whether the values in each type of bag is calculated correctly.

2.2 Tree decomposition construction

To be able to run the algorithm, a tree decomposition has to be constructed. First we construct the graph from the PACE data. Once this graph is created we have the node and edge information that is needed to construct the bags in the tree decomposition. If the tree decomposition was constructed without making the original graph, there would be no information about which nodes had edges in between them which is needed when running the algorithms.

Next up the information from the tree decomposition section of the PACE data is parsed. In the tree decomposition section of the data there is enough information to be able to fill each bag in the tree decomposition with the correct Node objects and connect the bags that are connected to each other. After this parsing we have a correct tree decomposition but not a nice tree decomposition.

A C# implementation for the construction of a nice tree decomposition was given to was obtained from Tom van der Zanden[2]. The code performs a preprocessing step on the tree decomposition data, which changes the tree decomposition into a nice tree decomposition. There is one method in the given code that is used to increase performance of algorithms on the tree decomposition, and that is not necessary for the construction of a correct nice tree decomposition. "ForgetBeforeIntroduce" adds a forget bag before any introduce. If a parent bag and child bag contain different nodes, you first want to forget each node in the child bag not in the parent bag before adding the nodes which were not in the child bag. If several introduce bags are put in the tree before the forget bags, the amount of computation that would have to be done would increase significantly as the size of the bags would become much larger than needed. The tree decomposition is then traversed in post-order. For each bag there are four options:

1. The current bag has no children: a leaf bag is added to the traversal

2. The current bag has one child:

   - If the child has a node in the bag that is not in the current bag, add a forget bag to the traversal
   - If the child does not contain a node that is in the current bag, add a introduce bag to the traversal

3. The current bag has two children: check if the left or right child needs nodes to be introduced to contain the same nodes as the join bag. If this is correct, a join bag will be added to the traversal

4. The current bag has more than two children: Throw an error, because "DeconstructJoins" was not done successfully

Once this traversal has been computed recursively the algorithms can be run on it.

2.3 Running the algorithm

Both the implementation for the GPU and CPU algorithm have a section that is executed on the CPU. While the computation for a single bag is done

[2]The author would like to thank Tom van der Zanden for publishing is implementation online. The code can be found here

on either the CPU or GPU, everything else in the algorithm is done on the CPU.

2.3.1 CPU Implementation

A for loop is used to execute every bag in the traversal. Each bag is represented as an array. The values in the array are the values for the subsets, the indices of the array are used to represent the different subsets. Say an array has length four, it means that there are two nodes in the bag, with indices: 0, 1, 2 and 3. The bit representation of these indices are used to represent each subset as follows:

- 00 (00000000) = no nodes in the subset
- 01 (00000001) = the first node of the bag is in the subset
- 10 (00000010) = the second node of the bag is in the subset
- 11 (00000011) = both nodes are in the subset

For each bag we compute the amount of subsets in a bag based on the amount of nodes in the bag. Next the computations are called by looking at the type of bag (Leaf, Introduce, IntroduceEdge, Forget or Join). A loop is needed to look at all the different possible subsets in the bag. This is done by using a for loop and using the bits in the iterator of the loop as the representation of the subset. For instance, if the value of the iterator is 7 and the bag has 5 nodes in it, in binary this would be "00111". It means that the first 3 nodes are in the set (either the independent set or dominating set) and the other 2 nodes are not.

The actual calculations for Leaf, Introduce, Introduce Edge and forget bag of the MIS and DS implementation are self-explanatory based on the description given in the introduction. The join bag of MS is simply adding two values, but the Join bag of DS is a lot more complicated. In the join bag for the minimum dominating set calculation we have can have multiple colourings from the children being consistent with the one colouring in the join bag. If the colouring for a bag is 0000 there are 16 different combinations of colourings in the child bags that are consistent with the colouring in the join bag.

One way of implementing the join bag is to create two lists of colourings in which the consistent combinations are saved. When all the combinations are created, the lists are used to check the index combination in the children of the join bag to find the best value. The problem with this approach is the size of the two lists, which are 2^{zeroes}. Because each thread only has a small amount of memory, creating to big arrays will be too large to put in thread local memory. The second approach is a binary counter, which is used in our approach. In this version of the join bag only a small array of size at most treewidth is needed, which can be done on the GPU. We create two bitmasks, bitmask A, B and C. Bitmask A masks the positions that contain a $\hat{0}$ or 1, bitmask B is a mask for the positions that contains a 0 and bitmask C is empty. A loop is ran from 0 till $(2^K) - 1$, if i^{th} bit of the iterator is 1, then the i^{th} 0 will go to the right child.

The left child is the original value bitmasked with $(A + B)$ and the right child is the original value bitmasked with $(A + C)$. If a bit in the iterator changes from a 0 to a 1, the 1 on the i^{th} position in bitmask B is changed to a 0 and the i^{th} position in bitmask C is changed to a 1. With this loop we iterate over all different possibilities. After the actual bag calculations the values of the previous bag(s) deleted if they are not needed in the future. If this is not the case, they will remain stored.

2.3.2 GPU Implementation

The structure of the code that calls the methods that do the calculations on the bags is the same as the CPU implementation, but extra steps must be taken for a successful execution. In the CPU implementation we could create new vectors and throw them away without explicitly allocating and deallocating them. With the GPU you have to allocate the space a bag will use on the GPU and deallocate the space, so it can be used again, when you are done with the bag.

A function that computes information on the GPU is called a kernel. Calling a kernel is also different than calling the bag computation function on the CPU. On the CPU you could call the function and it would return the new values in the bag. When a kernel is called a few parameters have to be known. The number of blocks and number of threads must be calculated before the kernel is called. The block size that is used in this research is 1024, so a single block contains 1024 threads. The amount of threads are decided by using the amount of tasks that need to be calculated and the amount of tasks one thread must calculate. For example, if there are 1024 items in the bag and each thread completes four tasks, we need 256 threads.

Because memory in the GPU cannot be used without allocating it first you cannot use growing vectors inside a kernel. The size of the vector must be given at the launch of the kernel so that the needed amount of memory for the vector can be allocated.

While the actual implementation of each kernel is different from the CPU, because they must be executed differently, the way the algorithm is structured stays mostly the same. Parameters and other processes are simply added or changed to make it possible to run the algorithm on the GPU.

2.4 Output

The most important data in the output is the time data. In order to accurately time the application we use the system clock which can be called in C++. The timer starts when the algorithm is called and stops when the result has been calculated. It means that for the GPU something like memory allocation also included in the total execution time. This was done because I believe that the setup time one implementation needs should added to the total execution time. If you would compare two algorithms one where the computation took 1 second and one where the actual computation took 0.001 second but to achieve this acceleration it had to do an extra 20 seconds op precomputation compared to the first algorithm, it should be mentioned or added to the total computation time. The other data generated by the algorithm is the final result for either the size of the maximum independent set or minimum domination set.

Besides the data generated by running the algorithm, data from the tree decomposition and graph that is used by the algorithm is saved with the output. This contains the following data:

- The ID of the PACE graph used
- Whether the data was generated by using the GPU or the CPU
- Whether the data was generated for the MIS or DS problem
- The width of the tree decomposition
- The number of runs executed
- The number of nodes in the graph
- The number of edges in the graph
- The number of bags in the tree decomposition

Each test will be ran 5 times. These 5 runs will then be averaged and used as the final time.

3 Profiling

When installing the CUDA toolkit, one can also install the NVIDIA Visual Profiler. The application can be used to profile CUDA code, and give warnings when the GPU is not used to its full capability. After profiling the written code, multiple warnings were given. These consist of two main warning types:

- Overall GPU usage warning: warnings related to the use of the GPU, for instance the use of memory and concurrency in the GPU
- Warnings of performance of kernels: warnings related to the way kernels are executed and their efficiency

The following warnings were given by the Visual Profiler. The descriptions of the warnings are copied from the Visual Profiler and written in cursive.

Overall GPU usage warning:

1. Low Compute / Memcpy Efficiency: "*The amount of time performing compute is low relative to the amount of time required for Memcpy.*"
   Memcpy is called at two locations in the code. Once when the code is finishes to copy the results back from the GPU memory, and once for each bag to put the ID/IDs and index/indices of the node/node(s) introduced or forgot by the current bag. This warning will not be resolved in this paper. One possible solution would be to count the number of bags in the tree decomposition, multiply this number by four and Memcpy the information for all nodes in one go. This way Memcpy is only called twice when solving a problem.

2. Low Memcpy/Compute Overlap: "*The percentage of time when Memcpy is being performed in parallel with compute is low.*"
   This is a warning that cannot be resolved for the application. The only times Memcpy is called is when the application has solved the problem and to copy the bag instructions to the GPU which is the ID of node 1 and 2, and the index of node 1 and 2 (node 2 is only used when introducing an edge). These operations cannot be performed in parallel.

3. Low Kernel Concurrency: "*The percentage of time when two kernels are being executed in parallel is low.*"

This warning is caused by the fact that we traverse the tree decomposition in post-order. By traversing the tree in this manner, we compute a single bag at a time which only needs one type of kernel. If values of bags were computed in an order in which bags that are not dependend on each other are solved at the same time it would be possible to execute different kernels at the same time.

4. Inefficient Memcpy Size: "*Small memory copies do not enable the GPU to fully use the host to device bandwidth.*"
   Memcpy is only used to copy the node details, which is four ints, and one int, the solution to the problem, so this warning is not unexpected. A possible solution might be the one given at the "Low Compute / Memcpy Efficiency" warning.

5. Low Memcpy Throughput: "*The memory copies are not fully using the available host to device bandwidth.*"
   Memcpy is only used to copy the node details, which is four ints, and one int, the solution to the problem, so this warning is not unexpected. A possible solution might be the one given at the "Low Compute / Memcpy Efficiency" warning.

6. Low Compute Utilization: "*The multiprocessors of one or more GPU's are most idle.*"
   The importance of this warning differs per tree decomposition, and is not something that can be easily solved. In this paper a post-order traversal of the tree decomposition is used. This means the current bag has to be solved to compute the values for the next bag. If the bag is small, a large part of the GPU will stay idle as other bags are not computed when using a post order traversal. If the tree decomposition has a small width, every bag will be small compared to the capability of the GPU, in which case most of the GPU will be idle while solving the problem. A possible solution for this could be to change the order in which the tree is traversed. If the tree decomposition has a tree structure multiple bags that are not dependent on the values of the other bags can be solved at the same time. If for instance, a tree has a single join bag, the left and right branch of the join bag can be solved completely separately. The only prerequisite to solving the join bag is that the values of both children are known. This way the idle multiprocessors could be computing other bags at the same time.

Warnings of performance of kernels

1. Low Global Memory Load Efficiency: "*Global Load efficiency indicates how well the application's global loads are using device memory bandwidth. The efficiency is the number of bytes requested divided by the number of bytes that were transferred from the device memory to satisfy those requests. Because device memory transfers bytes in blocks, the alignment and access pattern of a given store determines how many blocks must be transferred and thus determines the efficiency of that load. Low efficiency indicated that one or more global memory loads have a poor access pattern or alignment.*"
   Grouping threads into warps is not only useful for the speed of the computation but also for the global memory accesses. The GPU combines global memory loads by threads from a warp into as few calls as possible to minimize DRAM bandwidth as possible. When the memory accesses are next to one another, the hardware might be able to combine them into one call to memory. On the other hand, if multiple threads need to access global memory, but the memory locations are far apart in the physical memory, the GPU will not be able to combine the accesses. Unfortunately in the kernels for independent set and dominating set, the calls to load data from memory can be far apart. The lookups to the values in the previous bags can have a space between them, causing the effective bandwidth of the global memory to be poor.

2. Low Global Memory Store Efficiency: "*Global store efficiency indicates how well the application's global stores are using device memory bandwidth. The efficiency is the number of bytes stored divided by the number of bytes that were transferred to device memory to perform those stores. Because device memory transfers bytes in blocks, the alignment and access pattern of a given store determines how many blocks must be transferred and thus determines the efficiency of that store. Low efficiency indicated that one or more global memory stores have a poor access pattern or alignment.*"
   Storing the values in the global memory has the same problems as loading the values from memory.

3. Low Warp Execution Efficiency: "*Warp execution efficiency is the average percentage of active threads in each executed warp. Increased warp execution efficiency will increase utilization of the GPU's compute resources. These kernel's warp execution efficiency is less than 100% due to divergent branches and predicated instructions.*"
   Branch divergence can have a big impact on the performance of kernels. For introduceEdgeBag and introduceBag of both maximum indepen-

dent set and minimum dominating set, there are IF-statements that can be looked at removing or can be called in a different manner so that each thread has the same value at a IF-statement. If threads take the same route through a kernel, there will be no branch divergence no matter the amount of IF's. The join bag of dominating set takes a lot of time to compute, by the fact that it has multiple for loops and if statements, making changes in the join bag could result in big changes to Warp execution efficiency as each thread would have less different loop iterations.

The NVIDIA Visual Profiler also gives priorities for the kernel optimization. These priorities are based on execution time and achieved occupancy. The optimization of higher ranked kernels are more likely to improve the performance compared to lower ranked kernels. Of course this is different for each problem, so multiple graphs from PACE data are used that are different from one another to give a good indication of the priorities. For maximum independent set the following graphs are used:

- Graph 79 = width: 10, #nodes: 36415, #edges: 145635, #bags: 25210
- Graph 191 = width = 25, #nodes: 5096, #edges: 8105, #bags: 711
- Graph 193 = width: 26, #nodes: 1848, #edges: 3286, #bags: 1528

These graphs were chosen for different reasons:

- Graph 79 = large amount of edges and bags with a small width
- Graph 191 = graph with the largest amount of edges with large treewidth
- Graph 193 = graph with the largest treewidth of which the maximum independent set can be found by the implementation.

In the results of the *Kernel Optimization Priorities* kernels are split into multiple instances, there is no overview of each individual kernel, rather a large list of kernel instances for each kernel. For instance: "Rank 100, 54 instances, IntroduceEdgeBagKernelMIS". While the list contains more instances of IntroduceEdgeBagKernelMIS, these 54 are given the highest priority. Because using the data with instances gives a very large list, the ranks of the instances are combined to give an overall priority ranking.

For graph 79 the following priorities are given:

1. introduceEdgeBagKernelMIS
2. introducebagKernelMIS

3. forgetBagKernelMIS
4. joinBagKernelMIS
5. leafBagKernelMIS

For graph 191 the following priorities are given:

1. introduceEdgeBagKernelMIS
2. introducebagKernelMIS
3. forgetBagKernelMIS
4. joinBagKernelMIS
5. leafBagKernelMIS

For graph 193 the following priorities are given:

1. introducebagKernelMIS
2. introduceEdgeBagKernelMIS
3. joinBagKernelMIS
4. forgetBagKernelMIS
5. leafBagKernelMIS

4 Improvements

As is shown in Profiling, multiple warnings are given by the profiler. Although some of these warnings are inherent to the way this problem is solved, meaning the warnings cannot be resolved, others can be looked at for a possible increase in performance. Of the aforementioned warnings, the warning about "Low Warp Execution Efficiency" is looked at in detail to see if it can be improved.

Two changes will be looked at. One change to make the join bag of the dominating set more efficient and the second to the execution of multiple tasks per thread.

It might seem odd that the introduceBagKernel and the introduceEdgeBagKernel are not changed to increase performance. Both kernels are given high priority by the profiler. The reason that these kernels are not changed is the fact that there are no changes that can be made to the code to increase the efficiency of the code in the kernel, for instance decreasing the chance of branch divergence.

The introduceBagKernel contains two IF-statements that each contain an IF-statement. Although this might sound like a source of unnecessary branch divergence, this is not actually the case. The first IF is for whether the new node is added as the last element to the bag or not. This value stays the same for the whole bag, so no change for branch divergence. The reason this distinction is made because the computation for adding an as the last element to the bag costs a few less computation steps. Whether or not the element is added in last, does not change the fact that we have to check whether or not the current index has the element in the independent set or dominating set. This has to be checked using an IF statement, so there are no unnecessary IF statements in IntroduceBagKernel.

To improve the performance of the introduceEdgeBagKernel the way the kernel is called would have to be changed. The kernel has to have two IF-statements to check whether none, one, or both nodes are in the independent set or in the dominating set. This means that there are no IF-statements that can be removed to decrease the chance of divergence. If the kernel was only called when both nodes were in the set for the given index, there would be no divergence. Currently the kernel is always called on all subsets in the bag, but this would have to change to only the subsets in which the nodes to be connected are both in the set.

4.1 joinBagKernelDS

In the current implementation of the join bag kernel for the dominating set, the kernel firstly contains one for loop to find the location of all the zeroes for the given index. Once the zero locations are found the values for the different consistent combinations are all checked. To check all possible combinations, a nested for loop is used. The first loop loops over the zero positions in the index while the nested loop loops over the bag size. Using this technique all different consistent combinations are checked for the best value.

The downside of this technique is that we have for-loops in which divergence can occur. The first loop, that is used to find the positions of the zeroes in the index, is be traversed in a different way for different indices. The for loop itself costs a lot of time to traverse for a kernel, and the fact that branch divergence can occur in the loop makes it even more costly. The outer part of the second loop, the nested loop, loops over the different combinations of zeroes in the index. The inner loop then loops over the each value of the index to make the left and right consistent index combination. The same downside applies to this for loop, but because this is a nested loop it takes an significant amount of time to compute, while also having the possibility for divergence.

By precomputing the zero positions for each index and having the different combinations saved in memory, the first for loop can be removed as well as the inner loop of the second for loop. The removal of these for loops does cause the kernel to have more Memory loads, so the results will show if the decrease in computation time outweighs the increase in memory loads.

4.2 Thread tasks

The last change that will be made to the execution of the kernels is the use of multiple tasks per thread. A task in this case is the amount of times a thread executes a kernel. If a thread has 4 tasks, it would execute the kernel for index 0 till 4. Each thread has a "thread setup cost" to launch it. When using one thread per element, this setup cost has to be paid more often than the case where you have multiple tasks per thread, which is obvious as less threads are started. Of course the optimal amount of threads is using all the available threads in the GPU. When there are more tasks than threads in the GPU you want the threads to have more tasks as there will be less setup cost in that case. When there are less tasks than available threads, you would want to give each thread less tasks, as it would cause the GPU to only use a portion of the available threads. In this paper the amount of tasks per thread is a static amount for each problem, so the amount of tasks per thread is not changed based on the size of the current bag.

5 Results

To be able to see if there is a difference in performance of the CPU and the GPU implementation of independent set and dominating set, and whether the improvements increased the performance, tests need to be run. The following test of variable configurations was tested:

1. maximum independent set CPU implementation
2. maximum independent set GPU implementation, one task per thread
3. maximum independent set GPU implementation, eight tasks per thread
4. maximum independent set GPU implementation, 32 tasks per thread
5. maximum independent set GPU implementation, 128 tasks per thread

6. minimum dominating set CPU implementation
7. minimum dominating set GPU implementation, one task per thread
8. minimum dominating set GPU implementation, eight tasks per thread
9. minimum dominating set GPU implementation, 32 tasks per thread
10. minimum dominating set GPU implementation, 128 tasks per thread
11. minimum dominating set GPU implementation, one task per thread, precomputing join bag

The CPU and GPU implementations of maximum independent set and minimum dominating set are compared the by looking at the CPU implementation and the GPU implementation with one task per thread. The influence of the amount of tasks is decided by comparing the GPU implementations with one, eight, 32 or 128 tasks per thread respectively. Lastly the influence of changing the dominating set join bag is found by comparing the implementation with the altered join bag to the dominating set GPU implementation with a single task per thread.

These tests were ran on a laptop with the following characteristics:

- OS: Windows 10 Home, version 1803
- Processor: Intel Core i7-4710HQ @ 2.50GHz
- RAM: 8GB
- GPU: NVIDIA GeForce GTX 860M
- CUDA cores: 640
- GPU Memory Size: 2048 MB

Showing all the data generated by the runs in dedicated tables would take over 150 pages. That is why this document does not contain an appendix with tables. The complete results can be found in the following repository. The complete codebase for this thesis can be found in the following repository.

5.1 Maximum independent set

5.1.1 CPU vs GPU

The average solving time of each tree decomposition on the CPU and GPU and the speedup of the GPU compared to the CPU. Table 1 shows the data from a few graphs the algorithm was ran on. The first time the GPU has a better performance is at graph 127, with a minor speedup of 1.19×. With the width of the trees increasing, the amount of times the GPU outperforms the CPU also increases. After graph 161 the CPU is only quicker for two graphs. The biggest performance increase is with graphs 191, 193 and 195. of these graphs only 191 had the actual speedup, 22,70×. For graph 193 and 195, the speedup between CPU and GPU are not a fair comparison because the CPU crashed during the execution as it ran out of memory.

The results gotten from running these experiments show a difference between solving these problems on the CPU and GPU, although which tree decomposition the problem is solved on has a major influence. With maximum independent set graph 79 shows that the GPU struggles with a lot of bags of small size, while it is faster for the tree decompositions with the biggest widths.

graph	Avg time CPU (s)	Avg time GPU (s)	Speedup (×)
25	0,16	4,09	0,04
79	4,89	51,22	0,10
127	0,28	0,24	1,19
161	10,23	3,61	2,83
163	0,45	0,43	1,04
165	0,35	0,38	0,93
175	0,62	1,67	0,37
189	6,92	0,58	11,91
191	279,11	12,29	22,70
193	-1,00	2,84	-0,35
195	-1,00	3,12	-0,32

Table 1: Averaged data for maximum independent set on the GPU and CPU compared

5.1.2 Tread tasks

For the first few graphs shown in table 2 using 8 tasks per thread results in a speedup when compared to using 1 task per thread, but this becomes no speedup or a slight decrease in speed a few graphs later, around 0,9× -

1,0×. This decreases further to around a speedup of 0,8× for most of the graphs. With 32 threads the values are close together for most of the graphs. While 32 tasks per thread drops lower at the end to a speedup of 0,68× and 0,65× compared to 0,82× and 0,85× respectively, for most graphs with an id higher than 100 using 32 tasks per thread is slightly faster. Using 128 tasks per thread was only faster for the first graph. After this graph using 128 tasks per thread is always slower with a speedup of is between 0,55× and 0,65×.

Having multiple tasks per threads only lead to a tiny increase in speed for some cases, but most of the time it lead to a decrease in speed. The GPU that was used to run these tests on has 640 CUDA cores. You need 640 indices in a bag to give each thread its own task. If the bag has less than 640 indices a part of the GPU will be idle, while having more than 640 threads will cause the GPU to have to wait for threads to complete before starting the remaining threads. Nothing can be done when you have less than 640 threads, but when there are more than 640 threads, multiple tasks can be given to threads to negate the start-up cost of threads. when a thread is given 8 tasks, 5280 indices are needed, which is more than 2^{12}, and 84480 indices in a bag when each thread has 18 tasks per thread, more than 2^{16}. When the amount of tasks per thread do not change dynamically to be close to 640 a bag with 128 indices, will start one thread if it is told to have 128 tasks. This is around 128× slower than starting 128 threads. If the tree decomposition has bags with $\#indices < 640 \times tasksPerThread$ solving the problem will become slower than the version where 1 tasks per thread is used. For the trees with large width the amount of bags that have enough indices to have multiple tasks per thread increases, which is why a speedup can be seen at graphs 191, 193 and 195 for independent set.

	tasks per thread				Speedup (×)		
graph	1 task	8 tasks	32 tasks	128 tasks	1 vs 8	1 vs 32	1 vs 128
1	0,60	0,22	0,22	0,27	2,68	2,67	2,20
25	4,09	4,17	4,58	6,19	0,98	0,89	0,66
79	51,22	52,82	57,07	79,56	0,97	0,90	0,64
101	1,59	1,86	1,79	2,75	0,86	0,89	0,58
191	12,29	14,42	18,99	26,10	0,85	0,65	0,47
193	2,84	3,45	3,90	5,60	0,82	0,73	0,51
195	3,12	3,79	4,60	6,50	0,82	0,68	0,48

Table 2: Speedup of the maximum independent set GPU implementations with varying tasks per thread

5.2 Minimum dominating set

5.2.1 CPU vs GPU

In table 3 after the first seven graphs, the GPU implementation of the solving the minimum dominating set problem was faster for every other graph. For finding the minimum dominating set it was decided to have a cutoff point if finding the dominating set took longer than one hour. So if after one hour the solution is not found, the graph states the solution took longer than 3600 second to find. Until graph 67 "avg time CPU (s)" shows the actual time. This means that the speedups until graph 67 show the actual speedup, while the speedups afterwards are the minimum speedup. Graph 79 took the GPU 776,73 seconds and because of the cutoff point it looks like the speedup was small. But if the speedup on this graph was comparable to the other graphs around 79, the CPU solution might have taken more than a week to compute.

The maximum actual speedup is 167,38× faster (graph 37). While the minimum speedup was 1873,83× faster (graph 77).

The GPU implementation that solves the minimum dominating set problem becomes faster than the CPU implementation at a treewidth that is a lot smaller than the width where the GPU implementation for the independent set problem became consistently quicker as this was only after treewidth 20. This has to do with the fact that solving the minimum dominating set problem is done in base 3 while the independent set is solved in base 2. When using base 2, the amount of indices in the bag grow a lot slower than when you use base 3. Because of this the GPU can use more compute power for smaller bags. Where a bag with 6 nodes needs $2^6 = 64$ indices for the independent set, for the dominating set $3^6 = 729$ indices are needed. This means that for smaller bags less of the GPU stays idle, resulting in a better performance.

graph	avg time CPU (s)	avg time GPU (s)	Speedup (×)
13	0,17	0,81	0,21
17	3,49	0,14	25,85
25	9,31	4,44	2,10
61	1305,15	7,93	164,68
65	596,13	5,69	104,83
67	>3600	14,45	249,14
77	>3600	1,92	1873,83
79	>3600	776,73	4,63
89	>3600	8,13	442,65
91	>3600	-1	-1

Table 3: Averaged data for minimum dominating set on the GPU and CPU compared

5.2.2 Thread tasks

Table 4 shows the comparison between task amount per thread for the GPU implementations of the minimum dominating set problem. For dominating set other task sizes were compared. 8 and 32 are the same as with independent set, but now 9 and 27 are added. For dominating set each index can contain 3 different values. By using a value that that is not a multiple of three, the chance of having branch divergence is higher. When adding a new node to the front of the bag the value for colouring for the newly adding node will be split into equal parts 0, $\hat{0}$ and 1, so this chance is smaller. But when the node is not added to the front of the bag, the colouring will switch between colours more rapidly. When the amount of tasks per thread are not a multiple of three the tasks of a thread can be out of sync with each other causing one to start at them to start with different colours and maybe have different colours for all the tasks in the thread.

Using more than one thread gives an increase in speed for graph 13 when using eight on nine tasks per thread. For all the other graphs using more than one task per thread leads to a decrease in speed. Up until graph 39, using 9 tasks per thread often results in a slight increase in performance, but still worse than using one task per thread. Using both 27 or 32 tasks per thread leads to a major decrease in performance, going as far as a slowdown of 0,08× for 32 tasks and 0,10× for 27 tasks.

	tasks per thread					speedup (×)			
graph	1 task	8 tasks	9 tasks	27 tasks	32 tasks	1 vs 8	1 vs 9	1 vs 27	1 vs 32
1	0,09	0,28	0,35	0,27	0,28	0,87	0,69	0,91	0,87
25	4,44	5,80	5,01	7,66	8,38	0,77	0,9	0,59	0,54
27	0,03	0,13	0,14	0,37	0,45	0,21	0,2	0,08	0,06
39	1,05	1,88	1,81	3,22	3,70	0,57	0,59	0,33	0,29
51	2,80	8,78	9,39	25,10	30,80	0,32	0,3	0,11	0,09
53	1,20	4,37	4,72	-1	-1	0,29	0,27	0,1	0,08
59	3,96	-1	-1	-1	-1	0,28	0,26	-1	-1
79	776,73	-1	-1	-1	-1	0,28	0,26	-1	-1
91	-1	-1	-1	-1	-1	1	1	1	1

Table 4: Speedup of the minimum dominating set GPU implementations with varying tasks per thread

5.2.3 Join bag

The results for the join bag with precomputation, seen in table 5, does not contain the time it took to precompute the zero positions and combinations. If this was used in a practical application this step would have only been done once, and used forever, this is why is not counted as time for the actual computation.

The results show that for each graph that can be computed with the available memory the join bag with precomputation gives an increase in speed. The lowest speedup is 1,05(×) while the biggest increase is 28,48(×).

When the zero positions and different consistent index combinations are precomputed for the join bag of the minimum dominating set the computation time decreased for each graph. Although the kernel has a lot more calls to memory, which are slow because of the latency, is still outweighs the time it takes to compute every zero position and combination in each kernel. Having a nested loop where branch divergence can occur takes a lot of time, but even for the smaller graphs it is an increase in speed. The GPU might be able to combine optimize one of the calls to memory by making a single warp execute one call to memory instead of each thread making its own call, but other than that saving data to memory and reading it in in a kernel seems to be faster if the kernel otherwise has a lot of computational work.

graph	avg time GPU (s)	avg time GPU join (s)	Speedup (×)
25	4,44	4,16	1,07
27	0,03	0,02	1,78
39	1,05	1,00	1,05
67	14,45	0,52	27,89
69	14,55	0,51	28,48
79	776,73	82,03	9,47
89	8,13	3,07	2,65
91	-1	-1	1

Table 5: Averaged data for minimum dominating set on the GPU and GPU with precomputation for join bag compared

6 Conclusion

The results show a significant difference in speed for solving these problems on the CPU or GPU, with dominating set having the biggest increase in speed. An speed increase of 1873×. Because these results were computed on a mobile GPU, with a more powerful GPU, with more core and memory, the speedup of the GPU compared to the CPU could be even greater as it has more compute power and has enough memory to have bags for larger treewidths in memory.

Furthermore the influence of the tree on the results of the problems is shown. While the GPU get better results at higher treewidths a graph with many small bags, graph 79, can limit the performance of the GPU severely. Tweaking the parameters of the GPU implementation can help it cope better with different trees, but it is difficult to have a speedup with the GPU if the tree does not suit solving Dynamic Programming problems in parallel.

The fact that these significant speedups were achieved on these two problems can incentivize others to create GPU implementations for their Dynamic Programming problems.

7 Further research

While I have looked at different factors that could influence the performance of the GPU, such as the amount of tasks a single thread is given, there are still components that could be tweaked. Examples are: the number of threads in a block, concatenating MemCpy's, dynamic memory management, traversing the tree in a different way so that bags that are independent of each other can be solved at the same time, when they are independent, and dynamic task amount per thread.

During the experiments the size of the blocks were kept at the maximum amount of 1024 threads per block. I do not know whether lowering the thread count per block could be of influence, there might be an optimal amount block size based on the thread count, amount of computation in a thread etc.

As stated in Profiling, concatenating the MemCpy's could lead to an increase in performance, the speedup or slowdown of having one big MemCpy instead of a lot of smaller ones is also useful for other GPU applications, not just for algorithms on tree decompositions.

Reusing memory. The amount of memory allocations can be lowered by reusing memory allocations that are big enough to be used by other bags. This would result in less allocations and deallocations. However it would need some logic to make the most out of the available memory as you do not want to put a new bag of with 8 indices into an allocated memory space of 500MB.

The traversal of the decomposition was done using a post-order traversal, but finding another way of traversing the tree could be beneficial for the performance of the GPU implementation as less of the gpu could be idle when computing the bag values. Instead of using a post-order traversal a traversal could be constructed that takes the GPU the code is running on into account, so it could try to fill the GPU with enough bags that all CUDA cores are used.

As is seen with the results, the tree decomposition the independent set or dominating set is computed for has a big influence on the performance. This research only looked at the impact of changes to the algorithm and parameters used by the algorithm but not at the tree decomposition itself. Because the join bag for the dominating set is so computationally expensive, finding a way of using less join bags or using smaller join bags could be of influence to the execution time of finding the dominating set.

Acknowledgments

I would first like to thank my thesis supervisor prof. dr. Hans Bodlaender of Faculty of Science, Department of information and Computing Sciences, Algorithms and Complexity group at Universiteit Utrecht. The weekly meetings helped me along in the process by having a place to ask questions and speak about my progress. Even if I ran into trouble spots and there was not a lot of progress, he helped me figure out how to get up and going again.

I would also like to thanks Tom van der Zanden MSc, my second thesis supervisor. His advice during the weekly meetings was very useful and helped me progress is my thesis multiple times. He was also so generous to provide me with C# code that he used to generate tree decompositions. Having this code helped me write the C++ code I used for the tree generation.

I also want to acknowledge dr. Johan van Rooij of the Faculty of Science, Department of information and Computing Sciences, Algorithms and Complexity group at Universiteit Utrecht as the second reader of this thesis, I am gratefully indebted to his valuable comments on this thesis

Last but not least, I want to thank Rolf Plagmeijer for allowing me to look at the code he wrote for his research project. He wrote code to solve maximum independent set and minimum dominating set using path decomposition. Some of the code he wrote could be reused for this tree decomposition implementation and a lot could be used for inspiration on how to solve it using tree decomposition.

Finally, I must express my gratitude to my parents for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Author

Glenn Stewart

References

[1] Marco Cococcioni, Raffaele Grasso, and Michel Rixen. Rapid prototyping of high performance fuzzy computing applications using high level GPU programming for maritime operations support. In *Computational Intelligence for Security and Defense Applications (CISDA), 2011 IEEE Symposium on*, pages 17–23. IEEE, 2011.

[2] Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*, volume 4. Springer, 2015.

[3] Marek Cygan, Jesper Nederlof, Marcin Pilipczuk, Michal Pilipczuk, Joham MM van Rooij, and Jakub Onufry Wojtaszczyk. Solving connectivity problems parameterized by treewidth in single exponential time. In *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on*, pages 150–159. IEEE, 2011.

[4] Tuvi Etzion and Patric RJ Ostergard. Greedy and heuristic algorithms for codes and colorings. *IEEE Transactions on Information Theory*, 44(1):382–388, 1998.

[5] Thomas A Feo, Mauricio GC Resende, and Stuart H Smith. A greedy randomized adaptive search procedure for maximum independent set. *Operations Research*, 42(5):860–878, 1994.

[6] Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k nearest neighbor search using GPU. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*, pages 1–6. IEEE, 2008.

[7] K. Viswanathan Iyer. Np-completeness of independent set.

[8] Tim Kieritz, Dennis Luxen, Peter Sanders, and Christian Vetter. Distributed time-dependent contraction hierarchies. In *International Symposium on Experimental Algorithms*, pages 83–93. Springer, 2010.

[9] Ton Kloks. *Treewidth: Computations and Approximations*, volume 842. Springer Science & Business Media, 1994.

[10] Svetlin A Manavski and Giorgio Valle. Cuda compatible GPU cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC bioinformatics*, 9(2):S10, 2008.

[11] Jurij Mihelic and Borut Robic. Facility location and covering problems. In *Proc. of the 7th International Multiconference Information Society*, volume 500, 2004.

[12] NVIDIA. *Comparison of CPU and GPU architecture.* Feb 2010.

[13] Sebastian Ordyniak. Fixed-parameter algorithms, ia166.

[14] Pavel A Pevzner and Sing-Hoi Sze. Combinatorial approaches to finding subtle signals in dna sequences. In *International Society for Computational Biology*, volume 8, pages 269–278, 2000.

[15] Rolf Plagmeijer. Dynamic programming algorithms for graph problems on the GPU. this research looked at implementing maximum independent set and minimum dominating set on path decompositions. 2017.

[16] Neil Robertson and Paul D Seymour. Graph minors. III. planar treewidth. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.

[17] KP Sampoornam and K Rameshwaran. Efficient scheduling scheme using connected dominating set for sensed data aggregators in sensor networks. *Procedia Engineering*, 30:152–158, 2012.

[18] Michael C Schatz, Cole Trapnell, Arthur L Delcher, and Amitabh Varshney. High-throughput sequence alignment using graphics processing units. *BMC bioinformatics*, 8(1):474, 2007.

[19] TW Strijk, AM Verweij, and KI Aardal. Algorithms for maximum independent set applied to map labelling, 2000.

[20] Jie Wu. Extended dominating-set-based routing in ad hoc wireless networks with unidirectional links. *IEEE Transactions on Parallel and Distributed Systems*, 13(9):866–881, 2002.