RN/11/18
19 September 2011

Research
Note

Non-Recursive Beam Search on GPU for Formal Concept Analysis

W. B. Langdon, Shin Yoo and Mark Harman

Fax: +44 (0)20 7679 1397

Abstract

We document a parallel non-recursive beam search GPGPU FCA algorithm written in nVidia CUDA C. We run it on benchmarks and to analyse software module dependency. Despite `kernel_sort` removing repeated calculations, 32 bit packing and optimising GPU data structures and kernels, we do not yet see major speed ups. Instead GeForce 295 GTX and Tesla C2050 report 141 072 concepts (maximal rectangles, clusters) in about one second. Future improvements in graphics hardware may make GPU implementations of Galois lattices competitive.

Keywords: FCA, GPU, GPGPU, CUDA C, SBSE, software module clustering, module dependency graphs (MDG), close-by-one CbO, extent object row, intent attribute column, itemset, artificial intelligence, arithmetic intensity

Department of Computer Science
University College London
Gower Street
London WC1E 6BT, UK

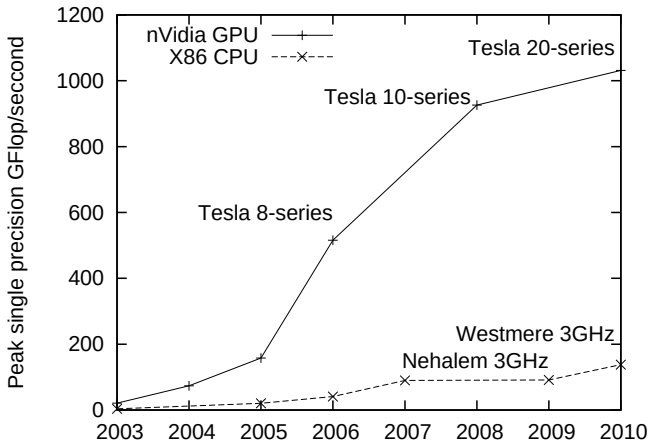

Figure 1: Comparison of increase in speed of graphics cards (+ GPU) and CPU (× X86). (Data supplied by nVidia.)

1 Introduction

We expand on [Langdon *et al.*, 2011] giving many additional implmentation details.

Formal Concept Analysis [Ganter and Wille, 1999] is a well known technique for grouping objects by the attributes they have in common. It can be thought of as a discrete data clustering technique. In general the number of conceptual clusters grows exponentially. However there are a number of specialised algorithms which render FCA manageable, even on quite large problems provided the object-attribute table is sparse [Krajca *et al.*, 2010]. [Krajca *et al.*, 2010] report considerable improvement in FCA algorithms in the last two decades. These successful algorithms treat the problem of finding all the conceptual clusters in an object-attribute table as a depth first tree search.

Computer graphics gaming cards (GPUs) are relatively cheap. Usually they cost less than the computer in which they are mounted and yet offer far more computing power than the computer's CPU alone (see Figure 1). Indeed, the saturation of single CPU clock speeds at about 3 GHz, means that significant increases in computing power will only come in future from parallel computing. There are hundreds of millions of computers fitted with graphics hardware which might be used for general purpose computing [Del Rizzo, 2008]. Even a humble laptop's GPU might be used to run Formal Concept Analysis.

For several years engineers and computer scientists have recognised the benefits of general purpose computing on GPUs and have used GPGPU in an increasing range of applications [Owens *et al.*, 2008]. Indeed the very top end GPUs are now dedicated supercomputers in their own right (rather than simply driving the computer's display) and tend to be priced accordingly. A consumer gaming card such as the 295 GTX contains 480 fully functioning processors and yet costs only a few hundred pounds. Whereas a state of the art Tesla C2050 costs a few thousand pounds. To get the best performance from a GPU it is necessary to divide the work load up into many thousands of independent computing threads [Langdon, 2011a]. nVidia's CUDA is probably the best of the tools available to help GPGPU programming, however OpenCL may become more popular.

[Krajca and Vychodil, 2009b] and [Krajca *et al.*, 2010] report using a distributed computer to overcome the "major drawback [of FCA's] computational complexity". They report their parallel algorithm PCbO gives near linear speed increase with number of computing nodes in a network of up to 15 PCs. In other work [Krajca and Vychodil, 2009a] they conclude that there is no universal best FCA data structure. Instead they suggest that the optimum performance will depend upon the application. In earlier work, Huaiguo Fu had created a parallel implementation of NextClosure but it was limited to 50 attributes [Fu and Nguifo, 2004] but this was subsequently greatly extended [Fu and O'Foghlu, 2008]. However, like [Krajca and Vychodil, 2009b] and [Krajca *et al.*, 2010],

both [Fu and Nguifo, 2004] and [Fu and O'Foghlu, 2008] approaches use conventional distributed computers composed of a few CPUs rather than hundreds of GPU processing elements. Similarly [Djoufak Kengue *et al.*, 2007]'s ParCIM implementation used a conventional network of 8 computers connected in a star fashion with MPI. Ours is the first FCA implementation to run in parallel on computer graphics cards (GPUs).

Borza, Sabou and Sacarea [Borza *et al.*, 2010] describe an interactive tool for FCA data mining. They use Andrews' In-Close algorithm which [Andrews, 2009] reports as some 20 times faster than "Krajca" [Krajca *et al.*, 2008; Vychodil, 2008]. Their conexplore tool is designed for exploring a few levels of the search tree. Like "Krajca", In-Close is a sequential recursive tree searching algorithm. In-Close's speed up appears to come from the exploitation of clever ways to prune the recursion.

2 CUDA FCA Implementation

We initially implemented the Krajca sequential algorithm [Krajca *et al.*, 2008] in Python. This was followed by a version in C, where `ComputeClosure` is implemented in parallel on the GPU using nVidia's CUDA framework. In several places we explicitly refer to the procedures and data structures defined in [Krajca *et al.*, 2008] and so it may help to read the following conjunction with [Krajca *et al.*, 2008].

Krajca's two recursive routines `ComputeClosure` and `GenerateFrom` essentially form a depth first search algorithm which builds and navigates a tree of formal concepts from a binary 0/1 matrix describing which object has which property. The search starts from the top of the tree with the empty concept: no object having no attributes.[1] Krajca *et al.*'s parallel algorithm [Krajca *et al.*, 2008] essentially partitions the tree into large recursive sub-searches, suitable for parallel operation on conventional networks of personal computers. The trick is to choose partitions so each computer has about the same work to do. Since the search is recursive and operates on one point in the search tree at one time, such search is fundamentally unsuitable for parallel operation on graphics cards. Indeed recursion has only recently been supported by nVidia's CUDA GPU frame work. Our graphics card parallel version retains the tree but changes the way and order in which it is searched. Consequently the concepts are printed in a different order. The search is a variant of beam search.

Instead of proceeding to the first leaf of the tree, recursively backing up and then going forward to the next leaf and so on, in beam search, we also start from the top of the tree and then proceed along every branch to the next level. This requires saving information on the beam for every node at that level. Beam search next expands the search again to cover everything at the next level and so on until all the leafs of the tree have been reached. Notice instead of working on a single point in the tree the beam covers many points which can be worked on in parallel. Indeed within a couple of levels we can get a beam containing tens of thousands of individual search points which can be processed independently. This suits the GPU architecture which needs literally thousands of independent processing threads for it to deliver its best performance [Langdon, 2011a]. You will have spotted that in an exponential problem, like FCA, beam search quickly runs out of memory.

Even for quite modest tree depths the beam width is limited by the available space in the GPU card. (We have a configuration limit of 1.8 million simultaneous parallel operations.) When a beam search exceeds this limit, only the first 1.8 million or so search are loaded onto the GPU and the rest of the beam is queued on the host PC. Typically PC memory is cheaper than graphics memory and it is usual for the host PC to have more memory than its GPU. (Although we have not done this, in multi-GPU systems it would be possible to split the beam between the GPUs, allocating up to 1.8 million to each GPU.) The GPU only searches to the next level. It returns the concepts found by the searches and the newly discovered branches which remain to be searched. The concepts are printed by the host PC and the new branches are added to the end of the beam to await their turn. Effectively the beam becomes a queue of points in the tree waiting to be searched. The number of parallel searches is mostly limited by the need to have space on the

[1] All rows of the object-attribute matrix are required to have at least one non-zero entry.

GPU for all the potential new branches. This depends upon the tree's fan out which is problem dependent. Nonetheless the GPU can manage modest real software engineering examples (e.g. dependence clustering of the Linux kernel). Notice the beam will contain a mixture of pending search points at different depths in the tree.

3 CUDA FCA Kernels

In nVidia's CUDA a kernel is a special function (typically a few tens of lines of C code) which runs on nVidia's graphics cards. Whereas the "Linux kernel" is tens of millions of lines of code which form the main part of the Linux operating system. CUDA gets is power from being being able to run each kernel on tens or even hundreds of thousands of data items in parallel.

In our first implementation of [Krajca *et al.*, 2008]'s `ComputeClosure` was implemented in parallel by a single CUDA kernel. However to better suite the GPU architecture it has been split into five kernels (see Figure 2). These are invoked by the host PC in order. At present, no attempt is made to overlap GPU operations.

3.1 GPU Data Structures and `kernel_init`

In the parallel version of `ComputeClosure` the four fixed length vectors: A, B, C and D each become two dimensional arrays. The additional index is needed to identify individual nodes in the tree as it is being searched in parallel. As with [Krajca *et al.*, 2008], the original index refers to the extent or intent (depending upon array type). For efficiency, in both the host and GPU code, 32 objects (or attributes) are packed into an actual `unsigned int` array element. (Although nVidia GPUs can move wider data items and perform long addresses and double precision floating point calculations, they essentially deal with 32 bit integer data.)

On the GPU the order in which array indices (i.e. column first or row first) can make a considerable difference to performance. One of `kernel_init`'s tasks is the reverse the order of the indices in array d_A. (By convention data stored in the GPU's global memory is named with a prefix d_.)

Most of the large arrays on the GPU do not require initialisation. However two arrays (d_MIIndex and d_More) are cleared each time a new set of closures are to be calculated. Surprisingly it turns out to be more efficient for the programmer to write special kernel code to do this, rather than to use the CUDA library routines.

3.2 constant `table`

[Krajca *et al.*, 2008]'s `ComputeClosure` makes heavy read access to the matrix holding the object-attribute table. The GPU architecture provides a small but very fast area of read only memory. It was decided early on to place the object-attribute table in this area of "constant" memory. Since problems of interest are typically sparse, a sparse memory data structure was devised to maximise the size of the arrays that could be processed. To maximise the data compression, the sparse array indexes inside `table` are packed together as 16 bit unsigned numbers.

Unfortunately "constant" memory proved tricky to get best of [Langdon, 2011c, Sect. 4.4]. Nonetheless, with care, it can provide very rapid access to important data. For example, it turns out to be much faster if all members of a group of threads (known as a warp) read the same part of `table` at the same time. This was the initial motivation for sorting the data fed into `kernel_computeClosure` (to be described in Section 3.4) by i. This led to the realisation that `kernel_computeClosure` was being asked to do the same calculations many times over. This in turn lead to the introduction of `kernel_sort` (next section) which not only groups data by i but also keeps track of these many duplicates.

The object-attribute table is loaded onto the GPU's constant memory at the start of the run and remains there throughout the run.

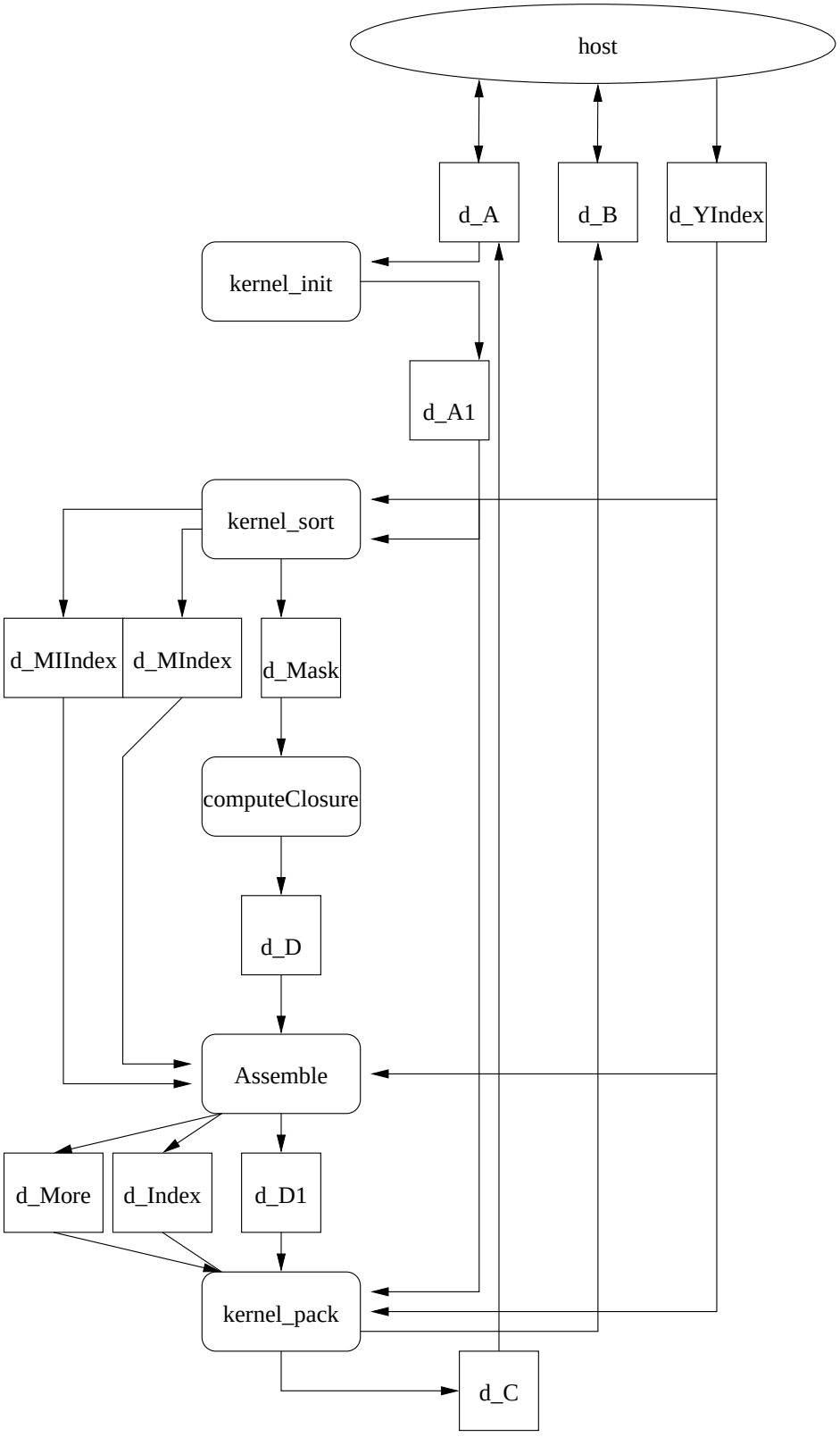

Figure 2: Main FCA global data (rectangles) and FCA program kernels (rectangles with rounded corners). The arrows show the data flows amongst the kernels and between the graphics card and the host PC (oval).

Even with sparse coding, the use of "constant" memory limits the number object–attribute pairs to at very most half a million. It would have to be replaced if this limit was even approached. More recent nVidia GPU include more caches. Even so, replacing constant memory by global memory would have to be done with care to use data locality to ensure each multiprocessor's cache was not overwhelmed.

3.3 `kernel_sort`

What is not clear from [Krajca *et al.*, 2008, page 73] is that in a typical problem large parts of calculations performed by `ComputeClosure` are identical. The early versions of `kernel_computeClosure` simply did them all and as a consequence total run time was dominated by the time the GPU spent in `kernel_computeClosure`.

Typically d_C and d_D have different dimensions and so `kernel_computeClosure` was split with the calculation of d_C being done in an pre-processing kernel and the results (d_Mask) passed to the new simpler `kernel_computeClosure`. All that `kernel_computeClosure` needs is a list of columns (i) for which there is work to be done. Since the work is identical each column need be passed to the simplified `kernel_computeClosure` only once. Typically this hugely reduces `kernel_computeClosure`'s work load and no further optimisation of it is needed. However the pre-processing kernel (now called `kernel_sort`) needs to both filter out the duplicate columns and remember where they came from and allocate space on the GPU to where `kernel_computeClosure`'s answers can be saved.

All the other kernels make one pass through their principle data source. This means mostly that they read global data once (they may have to read ancillary data more than once), process it, then write their output once. This is the usual way of working on GPU and it suits their parallel architecture. `kernel_sort` is the only one of our kernels which makes two passes through its input data.

The first pass uses a GPU shared function calculate_C to implement line 10 of [Krajca *et al.*, 2008, `ComputeClosure`, page 73]). To speed up calculate_C `kernel_sort` calls unpack_matrixw to unpack the sparse form of `table` each time it is used. calculate_C is coded to operate in parallel and take advantage of the fact that data are packed 32 to a word.

Each multi-processor on the GPU has a limited supply of fast memory that is shared between all the processing threads running on it. (Each half of the 295 GTX has 30 multi-processors.) To sort the data being produced in parallel by different threads it must be saved in way that the other threads can read it. To avoid both synchronisation problems and the delays associated with multiple accesses to global memory, `kernel_sort` is written to save data in shared memory.

To make the best use of the very limited shared data each multi-processor works exclusively on one column at a time. On a small problem, with less than 30 columns, this has the drawback that some multi-processors will be unused. However since we need only spot duplicates for the same column it has the effect of magnifying the available shared memory by the number of columns in the problem.

The GPU hardware provides synchronisation between all 32 threads in the same warp (warps were described in Section 3.2). Therefore up to 32 answers produced calculate_C for each warp are saved in a per warp hash table. (For each multi-processor we use 5 warps on the 295 GTX and 24 on the C2050). At the end of the first pass, only the first 32 threads are used to combine all the per warp hash tables into one. Both during initial construction and during combination our function insert() discards duplicates. Thus before starting the second pass we know exactly how many unique answers calculate_C will produce and we can pass them onto `kernel_computeClosure` (next section) via d_Mask. All this is done without explicit atomic operations and limited use of __syncthreads.

In the second pass we repeat the calculation of the first pass. (It is often more efficient to repeat calculations on the GPU than try and save them somewhere and read them back.) However we no longer need to look for duplicates. The results of the first pass tell us exactly what to expect. (Since the second pass only reads the combined hash table there are no synchronisation problems between any of the threads accessing it in

parallel.) Whenever calculate_C produces a duplicate we simply record in d_MIndex to which closure it belongs. Unfortunately this does require atomic operations.

Creating d_MIndex and d_MIIndex, as well as d_Mask, allows `kernel_computeClosure` to do its calculation once (using d_Mask) subsequently `kernel_Assemble` will use d_MIndex and d_MIIndex to ensure `kernel_computeClosure`'s results are propagated to multiple places.

`kernel_sort` is by far the largest kernel and consumes the largest proportion of the total run time (see $\star$ in Figure 3).

3.4 `kernel_computeClosure`

`kernel_computeClosure` processes all of the i in d_Mask in parallel. For each, unless `table`[i,j] is non-zero, it clears the corresponding D[j] in its output.

The implementation needs to take advantage of the fact that `table` is sparse, i.e. most of it is zero, and so care is needed to use the sparse coding (Section 3.2) to avoid clearing too much of D[j]. Secondly (as mentioned in Section 3.1) both `table`[i,j] and D[j] have 32 bit data packed into `unsigned int`.

Each processing thread assembles a complete 32 bit value (in local register notDw) and then writes it to global memory. Thus `kernel_computeClosure` only reads its input from global data once and writes its output to global data (d_D) once.

3.5 `kernel_Assemble`

`kernel_Assemble` is essentially post processing for `kernel_computeClosure`. It does two things. 1) it assembles the part answers (d_D). 2) Given j, B and, the newly calculated D[j], it decides if the recursion is complete.

`kernel_Assemble` sets all the bits in its local version of d_D1 (thus implementing lines 4–6 of [Krajca *et al.*, 2008, `ComputeClosure`, page 73]). The calculation of the conditional clearing of d_D1 (set d[j] to 0; line 11) has already been done by `kernel_computeClosure` (previous section). However to avoid repeating the calculation `kernel_computeClosure` does it only once and saves each partial answer in a location chosen by `kernel_sort`. The two arrays, d_MIIndex and d_MIndex respectively tell `kernel_Assemble` how many fragments it must assemble and where they are located. They are assembled simply by ANDing them with the local copy of d_D1. Once this is complete the local copy can be written to d_D1. Notice again global memory (d_D1) is only written to once. This is not only more efficient but also avoids potential problems of synchronisation between parallel processing threads.

If the recursion must be continued, `kernel_Assemble` sets a flag in d_More to say so. (d_More will be copied back to the host PC.) `kernel_init` (Section 3.1) has already cleared d_More, so that if the recursion is complete `kernel_Assemble` does not have to write zero into it.

3.6 `kernel_pack`

In a pure serial recursive implementation of `ComputeClosure` and `GenerateFrom` the four one dimensional arrays, A, B, C and D, are repeatedly passed between the recursive calls. In a parallel GPU implementation each array may have data for thousands of concepts being worked on simultaneously. Consequently they can be very big. Since passing data between the host PC and its graphics card is time consuming, we want to minimise the transfers of these four arrays.

Some of these data need to be passed back to the host PC however a copy still remains on the GPU. `kernel_pack` avoids the same data being passed from the host to the GPU at the next iteration by doing the same transfer internally within the GPU. Internal transfers are much faster than copying data between the PC and the GPU.

Table 1: Performance on random module dependency graphs (seconds). (For $\frac{1}{2}$ 295 GTX and Tesla C2050 the total time on the GPU is given.)

Dataset	Size	Density	Concepts	FCbO	Python	295 GTX	C2050
krajca	5×7	54%	16	0.00	0.11	0.01	0.01
wiki	10×5	44%	14	0.00	0.03	0.00	0.00
random	10×10	20%	16	0.00	0.04	0.00	0.00
random	100×100	2%	137	0.00	0.40	0.02	0.01
random	200×200	2%	420	0.00	4.33	0.00	0.01
random	500×500	2%	2861	0.01	162.60	0.02	0.02

4 Results

FCbO (version 2010/10/05) was downloaded from fcalgs.sourceforge.net and compiled without changes on a 2.66 GHz PC with 3 Gigabytes of RAM running 64 bit CentOS 5.0. The performance of FCbO, our Python code and our CUDA code on two types of GPU are given in Tables 1 and 2. Figure 3 plots the time taken by our individual CUDA kernels on the largest example.

Tables 1 and 2 show performance on: two bench mark problems, a selection of randomly generated symmetric object-attribute pairings and software module dependency graphs of the real world example programs.[2] Except in one case, all the implementations rapidly complete. It is only the Python prototype of Krajca's FCbO algorithm [Krajca *et al.*, 2010] which is troubled by the kernel of the Linux operating system.

The 295 GTX is actually two GPUs in one. As yet the implementation only allows one GPU to be used. It is expected that in a multiple GPU implementation the time taken by the 295 GTX would be approximately halved.

Tables 1 and 2 give the elapsed times taken by the CPU implementations. In the case of the 295 GTX and Tesla C2050 we have given the time taken by the GPU. This excludes an unfortunate start up overhead. Excluding the delay it causes gives a fairer indication of what the GPUs can do.

It is clear that FCbO is very fast on these examples. It is unclear why the parallel CUDA implementation does not exceed it.

5 Discussion

The Source Forge C implementation of FCbO is very fast. We expect it to be fast compared to our prototype Python implementation, since that does not use bit packing and Python is an interpreted language. We would expect a linear speed advantage for FCbO from both using 64 bit operations and from using compiled rather than interpreted code. However on sizable examples, the ratio between the speed of FCbO and that of our Python code is huge. This hints that FCbO has some algorithmic advantage. Although the Python code directly implements the algorithm given in [Krajca *et al.*, 2008], for implementation reasons it includes a sorting operation. This may be the cause of the non-linear slowdown of our Python code.

We had hoped it would be straight forward to run FCbO under CUDA and perhaps also try porting [Andrews, 2009]'s In-close. However it proved very hard to get FCbO running efficiently on the 295 GTX [Langdon, 2011c; Langdon, 2011b]. Additionally [Andrews, 2009]'s In-Close, like FCbO, uses recursive search and he says it requires "exponential memory". It also maintains a global current object which all searches refer to. Each of these may make it difficult to map In-Close onto a GPU architecture.

[2] The module dependency graphs are undirected and hence their corresponding object-attribute tables are symmetric. None of the algorithms have been altered to take advantage of this symmetry.

Table 2: Performance on Software Engineering datasets [Harman *et al.*, 2005]. Time given in seconds, except for the longest Python run which is hours:mins:secs. (For $\frac{1}{2}$ 295 GTX and Tesla C2050 the total time on the GPU is given.)

Dataset	Size	Density	Concepts	FCbO	Python	295 GTX	C2050
bison	37×37	24%	692	0.00	0.32	0.00	0.01
compiler	33×33	6%	24	0.00	0.05	0.00	0.00
dot	42×42	28%	1302	0.00	0.71	0.00	0.01
grappa	86×86	7%	850	0.00	2.54	0.01	0.01
incl	172×172	2%	238	0.00	1.84	0.00	0.01
ispell	24×24	34%	432	0.00	0.15	0.01	0.01
linuxConverted	955×955	2%	141072	0.73	15:42:51	1.79	0.93
mtunis	20×20	29%	110	0.00	0.05	0.00	0.01
rcs	29×29	37%	1074	0.00	0.46	0.01	0.02
swing	413×413	2%	3654	0.01	208.71	0.03	0.02

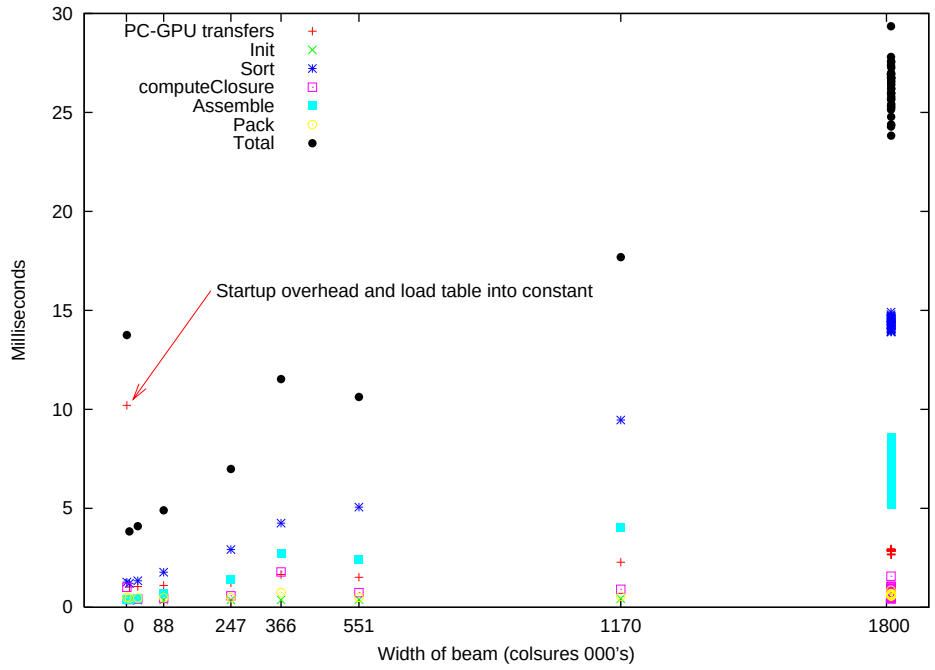

Figure 3: Time taken by the five CUDA kernels and on host–C2050 transfers for the 44 beam searches needed to process the largest example (linuxConverted). The clustering of data at the right hand side stems from the searches being limited by the GPU beam width ($\approx 1\,800\,000$).

Calculations on GPUs are often limited by the time taken to move data rather than the time taken to perform the calculations. One measure of this is "arithmetic intensity", which is the ratio of calculations per data item. Typical arithmetic intensity for common numeric three dimensional kernels vary from 4 floating point operations per transferred data element (4 FLOP/TDE) for a Laplacian stencil kernel up to 63.6 FLOP/TDE for tricubic interpolation [Christen *et al.*, 2011, p206]. We can estimate the arithmetic intensity for [Krajca *et al.*, 2008]'s algorithm by considering the given pseudo code (i.e. the procedures `ComputeClosure` and `GenerateFrom`) and concentrating on their inner loops (which are normally responsible for most of the computation). It appears that mostly `ComputeClosure` compares `table`[i,j] to zero (1 arithmetic operation per transferred data item) whilst `GenerateFrom` reads two data items and compares them (1 arithmetic operation/2 TDE). This suggests [Krajca *et al.*, 2008]'s algorithm has an arithmetic intensity of less than 1.0. Notice also, that these are integer or logic operations, which are usually faster than floating point operations. Thus a potential problem might be there is simply is not enough computation required by FCA compared to the volume of data. (Although arithmetic intensity is known to be an issue with GPUs, current CPUs also have the same problem of moving data to where computation is done. Usually this is ignored but some codes explicitly pre-load data into the CPU's cache memory in order to improve performance.)

Newer versions of CUDA have make it easier to overlap GPU operations. However our implementation does not do this at present. Therefore the next kernel is not started until all the multi-processors have finished processing the previous kernel. This causes some multi-processors to be idle at some point. Since the work is spread across the multi-processors, we suspect that idle time is not a major problem.

6 Conclusions

There are many problems (e.g. in artificial intelligence) which are traditionally solved by depth first search. On a modern serial computer this can be efficiently implemented recursively. However this may not suit low cost computer graphics GPU hardware. An alternative is beam search.

We have implemented a form of beam search and demonstrated it on several existing FCA benchmarks and ten software engineering dependence clustering problems [Harman *et al.*, 2005]. GPU beam search may also be more widely applicable.

Acknowledgements

I am grateful for the assistance of Gernot Ziegler of nVidia. Steve Worley, Sarnath Kannan, Stephen Swift, Stan Seibert and Yuanyuan Zhang. The software engineering module dependency graphs (MDGs) were supplied by Spiros Mancoridis. Tesla donated by nVidia. Funded by EPSRC grant EP/G060525/2.

References

[Andrews, 2009] Simon J. Andrews. In-close, a fast algorithm for computing formal concepts. In *Conceptual Structures Tools Interoperability Workshop at the 17th International Conference on Conceptual Structures*, Moscow, 26-31 July 2009.

[Borza *et al.*, 2010] Paul Valentine Borza, Ovidiu Sabou, and Christian Sacarea. OpenFCA, an open source formal concept analysis toolbox. In *IEEE International Conference on Automation Quality and Testing Robotics (AQTR 2010)*, volume 3, pages 1–5, 28-30 May 2010.

[Christen *et al.*, 2011] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Automatic code generation and tuning for stencil kernels on modern shared memory architectures. *Computer Science - Research and Development*, 26(3):205–210, 2011.

[Del Rizzo, 2008] Bryan Del Rizzo. Dice puts faith in nvidia PhysX technology for Mirror's Edge. NVIDIA Corporation press release, Nov 19 2008.

[Djoufak Kengue *et al.*, 2007] Jean Djoufak Kengue, Petko Valtchev, and Clementin Tayou Djamegni. Parallel computation of closed itemsets and implication rule bases. In Ivan Stojmenovic, Ruppa Thulasiram, Laurence Yang, Weijia Jia, Minyi Guo, and Rodrigo de Mello, editors, *Parallel and Distributed Processing and Applications*, volume 4742 of *LNCS*, pages 359–370. Springer, 2007.

[Fu and Nguifo, 2004] Huaiguo Fu and Engelbert Nguifo. A parallel algorithm to generate formal concepts for large data. In Peter Eklund, editor, *ICFCA*, volume 2961 of *LNAI*, pages 141–142. Springer, 2004.

[Fu and O'Foghlu, 2008] Huaiguo Fu and Micheal O'Foghlu. A distributed algorithm of density-based subspace frequent closed itemset mining. In *HPCC*, pages 750–755. IEEE, 2008.

[Ganter and Wille, 1999] Bernard Ganter and Rudolf Wille. *Formal Concept Analysis*. Springer, 1999.

[Harman *et al.*, 2005] Mark Harman, Stephen Swift, and Kiarash Mahdavi. An empirical study of the robustness of two module clustering fitness functions. In Hans-Georg Beyer, *et al.*, editors, *GECCO 2005*, pages 1029–1036, Washington DC, USA, 25-29 June 2005. ACM Press.

[Krajca and Vychodil, 2009a] Petr Krajca and Vilem Vychodil. Comparison of data structures for computing formal concepts. In Vicenç Torra, Yasuo Narukawa, and Masahiro Inuiguchi, editors, *6th International Conference on Modeling Decisions for Artificial Intelligence, MDAI 2009*, volume 5861 of *LNCS*, pages 114–125, Awaji Island, Japan, November 30-December 2 2009. Springer.

[Krajca and Vychodil, 2009b] Petr Krajca and Vilem Vychodil. Distributed algorithm for computing formal concepts using map-reduce framework. In Niall M. Adams, Céline Robardet, Arno Siebes, and Jean-François Boulicaut, editors, *8th International Symposium on Intelligent Data Analysis, IDA 2009*, volume 5772 of *LNCS*, pages 333–344, Lyon, France, August 31 - September 2 2009. Springer.

[Krajca *et al.*, 2008] Petr Krajca, Jan Outrata, and Vilem Vychodil. Parallel recursive algorithm for FCA. In Radim Belohlavek and Sergei O. Kuznetsov, editors, *The Sixth International Conference Concept Lattices and Their Applications, CLA 2008*, Olomouc, Czech Republic, October 21–23 2008.

[Krajca *et al.*, 2010] Petr Krajca, Jan Outrata, and Vilem Vychodil. Parallel algorithm for computing fixpoints of Galois connections. *Annals of Mathematics and Artificial Intelligence*, 59:257–272, 2010.

[Langdon *et al.*, 2011] W. B. Langdon, Shin Yoo, and M. Harmam. Formal concept analysis on graphics hardware. In Amedeo Napoli and Vilem Vychodil, editors, *The Eighth International Conference on Concept Lattices and Their Applications*, Nancy, France, October 17-21 2011.

[Langdon, 2011a] W. B. Langdon. Graphics processing units and genetic programming: An overview. *Soft Computing*, 15:1657–1669, August 2011.

[Langdon, 2011b] W. B. Langdon. Performing with CUDA. In Simon Harding, *et al.*, editors, *GECCO 2011 Computational intelligence on consumer games and graphics hardware (CIGPU)*, pages 423–430, Dublin, 13 July 2011. ACM.

[Langdon, 2011c] William B. Langdon. Debugging CUDA. In Simon Harding, *et al.*, editors, *GECCO 2011 Computational intelligence on consumer games and graphics hardware (CIGPU)*, pages 415–422, Dublin, 13 July 2011. ACM.

[Owens *et al.*, 2008] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008. Invited paper.

[Vychodil, 2008] Vilem Vychodil. A new algorithm for computing formal concepts. In Robert Trappl, editor, *19th Meeting on Cybernetics and Systems Research (EMCSR 2008)*, Vienna, March 25-28 2008.