Parallel Execution of Constraint Handling Rules on a Graphical Processing Unit

Amira Zaki, Thom Frühwirth, and Ilvar Geller

Faculty of Engineering and Computer Sciences, Ulm University, Germany
`{amira.zaki,thom.fruehwirth,ilvar.geller}@uni-ulm.de`

Abstract. Graphical Processing Units (GPUs) consist of hundreds of small cores, collectively operating to provide massive computation capabilities. The aim of this work is to utilize this technology to execute Constraint Handling Rules (CHR) which are inherently parallel. A translation scheme is defined to transform a subset of CHR rules to C/C++, then to use a GPU to fire the rules on all combinations of constraints. As proof of concept, the scheme was performed on several CHR examples.

Keywords: Constraint Handling Rules, CUDA, GPU, Parallel

1 Introduction

In recent years, graphics hardware has incurred a rapid increase in terms of performance. Its use has evolved from merely rendering graphics to offering a powerful platform for parallel computations. It has facilitated high performance computing to be readily available on a typical desktop, shipped as the common graphics processing units (GPUs). The powerful technology has become abundant at a relatively low price, hence it is tempting for researchers to harness this power for general-purpose computing to tackle intensive computations.

Furthermore, the introduction of CUDA (Compute Unified Device Architecture) by NVIDIA, a leading GPU manufacturer, gave rise to a new era of computing. CUDA allows users to seamlessly run C, C++ and Fortran code on a GPU, without requiring to resort to assembly language. CUDA has helped unleash the power of GPUs to be easily available to wide range of users [6]. Several works have emerged making use of this computing potential, like several number crunching algorithms [1], graph algorithms [4] and various others.

Constraint Handling Rules (CHR) is a committed-choice rule-based programming language having a well-established formal basis. The abstract semantics of CHR is inherently parallel, it involves multi-set rewriting over a multi-set of constraints [2]. CHR rules can be applied in parallel even to overlapping multi-sets of constraints, if they are removed by at most one rule. Thus it supports a very fine-grained form of parallelism.

A first abstract operational semantics for parallel CHR has been proposed by Thom Frühwirth [3]. Early prototypes for parallel execution of CHR have been developed based on shared-transaction memory (STM) by Edward Lam and Martin Sulzmann [5, 7]. Experimental evaluation of these systems revealed a significant boost and often linear speedup over sequential executions. However, these prototypes showed that conflicts occur with the STM-based approach; this results in a slow down of the execution. More recently, Andrea Triossi [8, 9] has developed a framework for compiling CHR to specialized hardware circuits. A code fragment of CHR is compiled into a low level hardware description language, to generate a specialized digital circuit on a Field Programmable Gate Array (FPGA) for each specific CHR code fragment. The hardware blocks then enable a parallel execution model for the compiled CHR fragment.

In this work, we aim to develop a prototype whilst exploiting the power of graphics processing units to simulate the execution of a subset of CHR by experimenting with different potential execution schemes. A translation scheme from CHR to CUDA is defined in such a manner that the output CUDA code is run in parallel, hence investigating the potential speed up of a parallel execution of the CHR rules.

2 CHR Overview

Constraint Handling Rules (CHR) is a high-level, concurrent, committed-choice, constraint logic programming language [2]. It consists of guarded rules that perform conditional transformation of multi-sets of constraints, known as a constraint store, until a fixed point is reached. CHR utilizes built-in constraints which are predefined by the host language, and other user-defined CHR constraints. A CHR constraint is a predicate having a name and a certain number of arguments. A CHR program typically consists of a finite set of rules, which can be generally represented with a simpagation rule as follows:

rule_name `@` *heads_kept* \ *heads_removed* `<=>` *guard* `|` *built_ins, body_constraints.*

The *rule_name* is an optional unique identifier given to a rule. *heads_kept, heads_removed, body_constraints* are a conjunction of one or more CHR constraints, where the constraints are kept, removed or added respectively. The rule operates by matching the *heads_kept* and *heads_removed* with constraints in the constraint store, then checks for the guard validity. If it holds then the *heads_removed* are removed from the store, and replaced with the *built_ins* and the *body_constraints.* Additionally there are propagation and simplification rules, which do not remove and do not keep any constraints respectively.

3 CUDA

CUDA offers a data parallel programming model that is supported on NVIDIA GPUs [10]. In this model, the host program launches a sequence of kernels, where a kernel is a hierarchy of threads. Threads are grouped into blocks, and blocks are grouped into a grid. The sizes of grids, blocks and threads is hardware dependent but a block typically contains 512 threads.

Each thread has a unique local index in its block (`threadIdx`), and each block of dimension (`blockDim`) has a unique index in the grid (`blockIdx`). The three indexes given are built-in 3-component vectors to access their values. Threads in a single block will be executed on a single multiprocessor, sharing the software data cache, and can synchronize and share data with threads in the same block. Threads in different blocks may be assigned to different multiprocessors concurrently, to the same multiprocessor concurrently, or may be assigned to the same or different multiprocessors at different times, depending on how the blocks are scheduled dynamically.

Thus a kernel is executed N times in parallel by N different CUDA threads. A kernel is defined using C/C++ functions and characterized with the `__global__` declaration specifier indicating that it is callable from the host only. The number of threads per block and the number of blocks per grid is specified using the `<<<...>>>>` statement. Other functions which are callable from the device only are indicated with `__device__` specifier.

4 Translation Scheme

The approach presented in this paper involves translating CHR rules into an imperative form, which can then be easily transformed into CUDA code to run on a graphics card. The CUDA code is run in parallel to simulate the parallel firing of the CHR rules. A subset of the CHR language is used, which includes only simplification rules and simpagation rules that do not introduce more constraints than those removed. This subset is a necessity due to the limited memory of the graphics card.

The CUDA code defines a structure for every CHR constraint, to store the information associated with it. The constraint store is then modeled and stored as an array of fixed length consisting of the structures. The dynamic nature of CHR constraints could be captured more clearly with a dynamic data structure such as a list structure, however this would not be practical on the graphics card. The GPU can not allocate memory in kernel calls because it does not contain a memory management unit. Moreover despite developments to support this feature in the future, there would still be an overhead introduced due to synchronization issues. Moreover, the total number of constraints possible in the lifetime of a program has to be known in advance, due to memory limitations of the GPU. For the scope of this work, a compromise was reached by choosing the subset of CHR that ensures an easy prediction of the number of constraints incurred by a program.

4.1 CHR Constraint Representation

Constraints represent data in a program and can be introduced and removed from the constraint store by CHR rules. Every constraint is a distinguished predicate of first order logic, having a name and a number of arguments. With the CHR Prolog implementation, every CHR constraint used has to be declared with a `chr_constraint/1` declaration by the constraint specifier. In its extended form, a constraint specifier is *constraint_name(type*$_1, \ldots,$ *type*$_n$*)*, where *constraint_name* is the constraint's functor, n its arity and the *type*$_i$ are argument specifiers. An argument specifier is a mode, followed by a type. Similar to the work done in [11], for every constraint a C/C++ structure is defined having the same name as the functor and with a listing of the arguments of the constraint using the provided types. Additional meta-data about the constraint can also be stored within the structure. Thus for a CHR constraint *constraint_name(type*$_1, \ldots,$ *type*$_n$*)*, the corresponding C/C++ structure can be defined accordingly:

```
typedef struct {
      type_1 var_1;
      ...
      type_n var_n;
      boolean isRemoved;
} constraint_name;
```

The variables var_i are used to store the arguments of the constraint. Additionally every constraint structure generated should contain a `boolean isRemoved` variable, which indicates the presence of the constraint in the store. It should be changed during the computation if the constraint was removed from the store.

As an example, a CHR constraint to describe a candidate number for the computation of a minimum can be expressed as: `min(+int)`. Using the previously mentioned translation scheme, it can be transformed into the following `MIN` structure:

```
typedef struct {
   int value;
   bool isRemoved;
} MIN;
```

The constraint store which contains N candidate minimum constraints is modeled as an array named `min_store` as follows: `MIN min_store [N]`.

4.2 CHR Rule Representation

The CHR subset chosen, should ensure that the body includes at most as many added constraints as the removed ones. Generic simpagation rules are taken as expressed in section 2. The subset chosen ensures that the number of constraints removed is at most equal to the added body constraints, thus:

$$|heads\_removed| \geq |body\_constraints|$$

A CHR rule can be translated into a function in C/C++, by mapping it to the following form:

```
void rule_name (calling_heads_kept, calling_heads_removed) {
   if(head constraints are not marked as removed
       && matching of variables in heads holds
        && guard holds) {
          equivalent built-ins, setting body constraints
   }
}
```

The name for a rule is optional in CHR but it is needed in C/C++ as a unique identifier for each function. The parameter list contains a listing referencing the structures of the equivalent head constraints. Constraints are fired only if they are actually present in the constraint store, thus first a check must be performed to check that they have not been marked as removed. Then when firing the rule, variables may exist in common between the head constraints and matching is performed. Thus in the translated C/C++ code matching of the variables must be explicitly ensured. Lastly before the rule fires, the guard must be checked if it holds and this must also be performed in the translated code. The guard is a typical condition and contains only built-in constraints which are expressed as straight forward C/C++ built-ins. The body consists of built-in constraints and overwrites existing constraints or deletes them by changing their `isRemoved` status variable.

Added constraints are actually overwritten in the place of removed head constraints. This is done by modifying their respective structure variables, to match the newly produced constraint. Head constraints that are removed and not overwritten, must have their `isRemoved` variable changed.

For example to calculate the minimum of a multi-set of numbers n_i expressed as `min(`n_1`)`,...`min(`n_k`)`, a simpagation rule that takes two `min` candidates and removes the one with the larger value is given as:

```
   minimum @  min(A) \ min(B) <=> A=<B | true.
```

The equivalent C/C++ function using the previously mentioned translation scheme is shown below. No variable matching is done in the rule, however both constraints are first checked for being present in the store. The guard is also checked, if all holds then the constraint with the larger value is removed from the store.

```
void minimum(MIN &a, MIN &b) {
   if(!a.isRemoved && !b.isRemoved && a.value <= b.value)
       b.isRemoved = true;
}
```

4.3 CHR Rule Firing

The translation scheme involves transforming the query constraints into the specified structure format and then placing them in an indexed array. The rule is fired on every possible combination of constraints. This exhaustive method can be optimized and changed according to the problem to be solved.

For the running minimum example, it is sufficient to apply the exhaustive firing. This means that the rule is fired for each pair of constraints; for an array `min_store` of N constraints, N^2 pairs are constructed.

The `MIN` constraints are stored in an indexed array and for each constraint pair the rule-function is called. Using the exact same constraint in the rule does not make any sense since a constraint is only present once in the store and should not be fired against itself unless it is present twice in the store, therefore a further if statement is needed. At the end of the loops the result will be a single non-removed constraint in the array which contains the smallest value. Encapsulating this functionality into a fire function is shown below:

```
void fire_minimum(MIN *min_store, int N) {
  for (int i = 0; i < N; i += 1)
    for (int j = 0; j < N; j += 1)
      if (i != j)
        minimum(min_store[i], min_store[j]);
}
```

4.4 Mapping to CUDA

After translating a CHR program into a C/C++ program it can be mapped with little effort into a CUDA program. Every CHR rule was mapped into a C/C++ function, which is now defined to be called by a thread from a device, and thus is redefined by adding the `__device__` declaration specifier. The function is redefined to the following:

```
__device__ void minimum(MIN &a, MIN &b) {
  if(!a.isRemoved && !b.isRemoved && a.value <= b.value)
   b.isRemoved = true;
}
```

The calls to the rule-firing functions, which were shown in the previous section as nested loops, will now be run in parallel. This straightforward translation with nested for-loops is perfectly suitable for the massive parallelism of CUDA. The outer loop is now considered as a block and each block can be designed to have 512 threads working on its content. With this thread layout a large amount of data can be processed.

An alternative approach to parallelize both loops is possible, but the amount of data has to be significantly smaller and a greater overhead is incurred leading to a slow down. The topic of work distribution between the threads remains a subject of future investigations.

The loop-firing function is now declared with `__global__`. The loops are reduced by one dimension, which now is replaced by the index of the handling thread (calculated from one-dimension of the 3-component indexes). For the minimum example, this now becomes:

```
__global__ void fire_minimum(int *min_store, int N) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  for (int j = 0; j < N; j += 1)
   if(i != j)
    minimum(min_store[i], min_store[j]);
}
```

In the CUDA code's main body, the number of threads is initialized. Assuming we have defined an array, `min_store`, of `N` minimum constraints, and a `block_size` equal to 512, then the initialization of the worker threads and assigning them to the firing function is done as given below:

```
int num_blocks = N / block_size + (N % block_size == 0 ? 0 : 1);
fire_minimum <<< num_blocks, block_size >>> (min_store, N);
```

A CUDA kernel launch is asynchronous and returns immediately. Thus to ensure synchronization between the worker threads, `cudaThreadSynchronize()` should be called to block execution until the device has completed all preceding tasks. This would ensure that all worker threads fire a single rule synchronously, and update the needed constraints before launching another kernel round.

5 Dynamic Detection Enhancement: Floyd-Warshall

As a proof of concept, several different algorithms were investigated and translated using the proposed scheme. These algorithms were the Sieve of Eratosthenes, GCD calculation and Floyd-Warshall. Due to the limited space of this short paper, the latter one will only be presented here. It sheds light on the need for an enhancement to the initial translation scheme to allow for dynamic detection of re-firing of rules due to new constraints that have been added.

The Floyd-Warshall algorithm finds the length of the shortest paths between all pairs of vertexes in a weighted graph. An edge can be represented by a CHR constraint `edge(?int,?int,?int)`, with the first two parameters expressing a connection between two connected integer indexed nodes in a graph and the third parameter describing the weight of the edge. The Floyd-Warshall algorithm can be expressed in a single CHR rule:

```
floydw @ edge(I, K, D1), edge(K, J, D2) \ edge(I , J, D3)
      <=> D3 > D1 + D2
       | D4 is D1 + D2, edge(I, J, D4).
```

The `edge` constraints are stored in an array named `edges_store`; each one is modeled using the following structure:

```
typedef struct {
   int from, to, weight;
   bool isRemoved;
} EDGE;
```

The number of constraints in the program life cycle is equal to the number of input constraints, as the rule only overwrites an existing constraint. The `floydw` rule is transformed into the following CUDA function:

```
__device__ void floydw (EDGE &a, EDGE &b, EDGE &c) {
  if(!a.isRemoved && !b.isRemoved && !c.isRemoved
      && a.from == c.from && a.to == b.from && b.to == c.to
       && c.weight > a.weight + b.weight)
         c.distance = a.distance + b.distance;
}
```

Since the rule tries the matching of three heads, it follows that the rule firings require three nested for-loops. Similar to the previous example this gets reduced to two for-loops, and the resulting in the CUDA code is shown below:

```
__global__ void fire_floydw(EDGE *edges_store, int N) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  for (int j = 0; j < N; j += 1)
   for (int k = 0; k < N; k += 1)
    if (k != j && k != i && j != i)
     floydw(edges_store[i], edges_store[j], edges_store[k]);
}
```

However in this example an existing constraint is overwritten and a new constraint has been introduced into the constraint store. This new constraint must be tried in a potential rule application. Thus it is not sufficient to fire the rule on every triplet combination, rather the firings must be performed exhaustively until no changes have been done.

Thus a boolean flag (`update`) is introduced which detects if a new constraint has been added. The flag is changed inside the body of the if-statement of the `floydw` function. Inside a loop, the CUDA threads are initialized and call the kernel fire function. This takes place several times until no new constraint is added to the store. A simplified CUDA code snippet for this would look like:

```
int update = 1;
while (update) {
   update = 0; ...
   fire_floydw <<< n_blocks, block_size >>> (edges_store, N);
   cudaThreadSynchronize();  ...
}
```

6 Conclusion

Constraint Handling Rules is a declarative multi-headed guarded rule-based programming language, which is parallel by nature. Graphics processing units have gained popularity nowadays, and have emerged as a cheap and powerful computation power for parallel executions; their uses have exceeded the rendering of graphics and have become desirable for various computationally expensive tasks. In this work, we described a means to model the parallel execution of CHR onto a graphics processor. Due to the limited memory of graphical units, a subset of CHR was chosen which ensures that the maximal number of CHR constraints present in the constraint store throughout the course of the program is known beforehand. The scheme translates CHR constraints to C/C++ structures, defines an array of these structures to denote the constraint store and each rule into a function that performs the firing action. The firing of rules is simulated by nested for-loops that fire rules on all combinations of constraints available in the store.

The work presented is still in progress, requiring several extensions, benchmarks and generalizations. Benchmarks to access the value of the gained speed up which the translation incurred is missing. Furthermore, an automatic CHR-to-CUDA translator that produces the output CUDA code would be greatly advantageous. Another criterion which further needs optimization is the rule firings methodology and threads work load distribution. The process used in this work was a naive one which exhaustively tries all combinations of constraint pairs, this could be altered and optimized. Benchmarks for the various options for rule firings would then be an interesting open topic to access.

References

1. Chen, H., Cheng, C., Hung, S., Lin, Z.: Integer number crunching on the cell processor. Proceedings of the 39th International Conference on Parallel Processing, 508–515 (2010).
2. Frühwirth, T.: Constraint handling rules. Cambridge University Press (2009).
3. Frühwirth, T.: Parallelizing union-find in constraint handling rules using confluence analysis. Logic Programming: 21st International Conference, 113–127. Springer (2005).
4. Katz, G., Kider, J.: All-pairs shortest-paths for large graphs on the GPU. In Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, 47–55 (2008).
5. Lam, E., Sulzmann, M.: Concurrent goal-based execution of constraint handling rules. Theory and Practice of Logic Programming, 841–879 (2010).
6. Owens, J., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A., Purcell, T.: A survey of general-purpose computation on graphics hardware. Eurographics 2005, State of the Art Reports, 21–51, (2005).
7. Sulzmann, M., Lam, E.: Parallel execution of multi-set constraint rewrite rules. Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming, 20–31, ACM Press (2008).
8. Triossi, A.: Hardware execution of constraint handling rules. PhD Thesis (2011).
9. Triossi, A., Orlando, S., Raffaetá, A., Frühwirth, T.: Compiling CHR to parallel hardware. Proceedings of the 14th international ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming, ACM Press (2012, forthcoming).
10. NVIDIA Corporation: NVIDIA CUDA C Programming Guide. Version 4.3 (2012).
11. Wuille, P., Schrijvers, T., Demoen, B.: CCHR: the fastest CHR implementation. Proceedings of the 4th Workshop on Constraint Handling Rules, 123-137 (2007).