Towards a GPU-based Implementation of Interaction Nets (extended abstract)

Eugen Jiresch *
jiresch@logic.at

Institute for Computer Languages
Vienna University of Technology

Abstract. We discuss advantages and challenges of our ongoing implementation of a GPU-based, parallel interaction nets evaluator.

1 Introduction

Interaction nets are a model of computation based on graph rewriting. They enjoy several useful properties which makes them a promising candidate for a future functional programming language. In particular, reducible expressions in a net can be evaluated in any order, even in parallel. This makes an implementation of interaction nets on a multicore architecture attractive.

In the recent years, a trend towards using graphics processing units (GPUs) for general purpose computations has emerged. Due to the increasing programmability of GPUs and general purpose APIs (CUDA, OpenCL), the parallel processing power of graphics cards may be used for many kinds of problems, from physics simulations to cracking passwords. In this abstract, we investigate the parallel evaluation of interaction nets using GPUs. While the GPU model of parallelism seems to fit interaction nets well, several factors make an implementation a non-trivial task. We argue that these factors can be overcome to provide efficient evaluation of interaction nets and describe our ongoing prototype implementation.

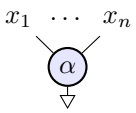
In the remainder of this section, we recall the main notions of interaction nets and the lightweight calculus, which is the basis for our implementation. Section 2 describes the current status of our prototype implementation using the CUDA *Thrust* library. We discuss related work and conclude in Section 3.

1.1 Preliminaries

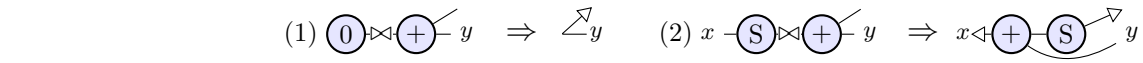
We now recap the main notions of interaction nets and the lightweight interaction calculus. *Interaction nets* were first introduced in [Laf90].

* The author was supported by the Austrian Academy of Sciences (ÖAW) under grant no. 22932 and the Vienna PhD School of Informatics.

A *net* is a graph consisting of *agents* (nodes) and *ports* (edges). Agent labels denote data or function symbols. Computation is modeled by rewriting the graph, which is based on *interaction rules*.



These rules apply to two nodes which are connected by their *principal ports* (indicated by the arrows), forming an *active pair* (or *redex*). For example, the following rules model the addition of natural numbers (encoded by 0 and a successor function S):



This simple system allows for parallel evaluation of programs: active pairs are completely independent of each other. Reducing a pair cannot change, destroy or duplicate another one. Furthermore, any order of evaluation yields the same result (due to *uniform confluence*).

The *lightweight calculus* [HMS10] is a textual representation of interaction nets, providing the basis for our implementation. It handles application of rules as well as rewiring and connecting of ports and agents. It uses the following ingredients:

Symbols Σ representing agents, denoted by α, β, γ.
Names N representing ports, denoted by $x, y, z, x_1, y_1, z_1, \ldots$.
Terms T either a name or a symbol with a number of subterms, corresponding to the agent's arity: $t = x \mid \alpha(t_1, \ldots, t_n)$.
Equations E denoted by $t = s$ where t, s are terms, representing connections in a net. If neither t nor s are a name, then the equation represents an active pair. Δ, Θ denote multisets of equations.
Configurations C representing a net by $\langle \bar{t} \mid \Delta \rangle$. $\bar{t}$ is the interface of the net, i.e., its ports that are not connected.
Rules R denoted by $\alpha(\bar{t_1}) = \beta(\bar{t_2}) \to \Theta$. α, β is the active pair of the left-hand side (LHS) of the rule and the set of equations Θ represents the right-hand side (RHS).

Rewriting a net is modeled by applying four *reduction rules* to a configuration:

Communication: $\langle\ \bar{t} \mid x = t, x = u, \Delta \rangle \xrightarrow{com} \langle\ \bar{t} \mid t = u, \Delta \rangle$
Substitution: $\langle\ \bar{t} \mid x = t, u = s, \Delta \rangle \xrightarrow{sub} \langle\ \bar{t} \mid u[t/x] = s, \Delta \rangle$, where u is not a name.
Collect $\langle\ \bar{t} \mid x = t, \Delta \rangle \xrightarrow{col} \langle\ \bar{t}[t/x] \mid \Delta \rangle$, where x occurs in $\bar{t}$.
Interaction $\langle\ \bar{t} \mid \alpha(\bar{t_1}) = \beta(\bar{t_2}), \Delta \rangle \xrightarrow{int} \langle\ \bar{t} \mid \Theta', \Delta \rangle$, where $\alpha(\bar{s_1}) = \beta(\bar{u_2}) \to \Theta\ \in \mathrm{R}$. Θ' denotes Θ where all bound names in Θ receive fresh names and $\bar{s}, \bar{u}$ are replaced by $\bar{t_1}, \bar{t_2}$.

Intuitively, $\xrightarrow{int}$ replaces an active pair by the RHS of the corresponding rule. The other three reduction rules substitute names, which corresponds to resolving a connection between two agents. It is important to note that only substitutions done by $\xrightarrow{com}$ yield a new active pair. This means that we can reach a normal form of a net (i.e., free of active pairs) by using only $\xrightarrow{int}$ and $\xrightarrow{com}$ rules:

Theorem 1 ([HMS10]) *If $C_1 \longrightarrow^* C_2$ then there is a configuration C such that $C_1 \longrightarrow^* C \xrightarrow{sub}{}^* . \xrightarrow{col}{}^* C_2$, and C_1 is reduced to C applying only communication and interaction rules.*

2 Parallel Evaluation of Interaction Nets in CUDA/Thrust

The general flow of a program using the GPU for data-parallel computation is as follows: an array of input data sets is copied from the main memory (known as *host* memory) to the memory of the GPU (also referred to as *device*). A function (the *kernel*) is executed on the GPU in parallel on each individual data set. Finally, the array of results is copied back to main memory.

In general, implementing an algorithm on a GPU efficiently requires a considerable amount of low-level decisions: factors such as size of data structures, number of threads and thread block size can greatly influence performance. Fortunately, the latest version of CUDA introduces the *Thrust* library [HB10], which features `transform()` and `reduce()` constructs that are similar to Haskell's `map` and `fold`. These functions are a convenient way to write parallel programs without the need for low-level tweaking. Our interaction nets evaluator is completely based on the *Thrust* library.

2.1 Motivation and Challenges

Why attempt a GPU-based implementation of interaction nets in the first place? Several reasons can be given: first, the SIMD (Single Instruction, Multiple Data) model of GPUs is similar to the idea behind interaction nets. Several independent data sets are processed in parallel using the same instruction/program. This is analogous to reducing several active pairs with a common set of interaction rules. Second, the reduction of a single active pair is a fairly small computation, consisting only of a few lines of code. GPUs are optimized for running thousands of threads executing such small programs. Additionally, the number of active pairs existing at the same time may vary greatly through the execution of a program. An interaction nets evaluator should be able to dynamically and transparently scale this potential parallelism to the manycore hardware. Again, GPUs are a promising platform to achieve this.

However, the implementation of interaction nets on a GPU is a non-trivial task. In particular, it poses the following challenges:

Maintaining the net structure While active pairs can be reduced in parallel, they are not completely independent: they are connected in a net, and resolving these connections is needed to generate further active pairs. Unfortunately, the choice of data structures in GPU memory is very limited (essentially just arrays). This means that the net structure needs to be maintained via device memory pointers or an additional table of connections, possibly residing in main memory.

Varying output size of a reduction Reducing one active pair may yield an arbitrarily large net, or any number of equations in the lightweight calculus. GPU-based algorithms usually have a fixed output size for every input.

Solving these challenges is all but completed, as we will discuss in the following subsection.

2.2 Status of the Implementation

Our implementation is still work in progress. We represent agents and variables in a straightforward way:

```
struct Agent
{
  int   id;       // unique id of the agent
  char  name;     // name of the agent
  int   arity;    // number of ports
  device_ptr<Agent>   ports[nPorts]; // connected agents
};
```

The `device_ptr` type is provided by *Thrust* as a pointer to GPU memory. We represent equations as pairs of agents.

So far, we have implemented the *interaction* and *communication* reduction rules. We have prioritized these as the other two rules do not generate or reduce active pairs (see Theorem 1). The control flow of our evaluator is simple:

```
transfer list of equations l to device (GPU) memory
while( l contains active pairs )
{
   apply ⟶^int to all active pairs in l in parallel
   apply ⟶^com where possible
}
transfer l back to host (CPU) memory
```

The *interaction* rule can be parallelized easily: we use a kernel to apply the current set of interaction rules to the list of equations. In general, a rule RHS may contain an arbitrary number of equations. Unfortunately, *Thrust*'s algorithms can only return a fixed number of results per individual input. We have solved this problem in a pragmatic way by setting a maximum number n of equations per RHS. Applying an interaction rule yields between zero and n equations and a number of *dummy equations*, resulting in a fixed result size for each application of the kernel. The dummy equations are filtered in a subsequent parallel computation step.

After the interaction phase above has completed, we apply the *communication* rule to generate new active pairs. Communication cannot be parallelized as easily as *interaction*. The reduction rule requires pairs of equations, and one equation may be part of two different communication steps. This is the case if both sides of the equation are a variable. Therefore, all communication steps are

currently done sequentially. Note that parallel algorithms are still used to filter equations that are eligible for communication.

Performance At its current early stage, the implementation is slower than more mature interaction nets based systems such as *inets* or *amineLight* [ine]. One reason for this is the current inefficient handling of GPU memory (an *array-of-structures* representation of the equation list). This reduces the potential for parallelism. In addition, a high number of transfer operations between CPU and GPU memory slows down the evaluation. We conjecture that once these issues are resolved, a GPU implementation of interaction nets will outperform the sequential systems mentioned above (on examples that have sufficient potential for parallelism).

3 Discussion

Related Work Only few papers (for example, [Pin01]) on parallel evaluation of interaction nets exist. This is surprising, considering their potential for parallelism. To the best of our knowledge, no previous work on a GPU-based implementation exists.

With regard to functional programming in general, several systems based on GPUs have been developed. Two recent examples are *Obsidian* [SCS10] and *Accelerate* [CKL+11], both being extensions to Haskell.

Conclusion In this abstract, we have discussed the potential benefits of a GPU-based evaluation of interaction nets. We have given some information on our ongoing implementation of such an evaluator. We plan to complete and optimize the system in the near future, and present benchmark results comparing our approach to existing interaction nets evaluators. For future work, we are also considering the integration of our results in existing systems such as *inets* [ine].

References

[CKL+11] Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. In Manuel Carro and John H. Reppy, editors, *DAMP*, pages 3–14. ACM, 2011.

[HB10] Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2010. Version 1.3.0.

[HMS10] Abubakar Hassan, Ian Mackie, and Shinya Sato. A lightweight abstract machine for interaction nets. *ECEASST*, 29, 2010.

[ine] The inets project. `http://www.informatics.sussex.ac.uk/research/projects/inets/`.

[Laf90] Yves Lafont. Interaction nets. *Proceedings, 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108, 1990.

[Pin01] Jorge Sousa Pinto. Parallel evaluation of interaction nets with mpine. In Aart Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 353–356. Springer, 2001.

[SCS10] Joel Svensson, Koen Claessen, and Mary Sheeran. Gpgpu kernel implementation and refinement using obsidian. *Procedia CS*, 1(1):2065–2074, 2010.