Compilation of Universal Probabilistic Programs to GPGPUs

DANIEL LUNDÉN, KTH Royal Institute of Technology
JOEY ÖHMAN, KTH Royal Institute of Technology
DAVID BROMAN, KTH Royal Institute of Technology

This work-in-progress paper describes an effort of creating a highly efficient compilation and runtime environment for probabilistic programs. In particular, the work consists of three parts: (i) the formal definition of the core of a universal probabilistic programming language (PPL) from which domain-specific PPLs can be derived, (ii) static analysis and efficient compilation of the core language down to General Purpose Graphical Processing Unit (GPGPU) code, and (iii) an efficient probabilistic inference and runtime engine that executes the program on a GPGPU. The probabilistic core language (called PPLCore) is developed as part of Miking—a framework for constructing domain-specific languages and compilers. The current GPU engine is based on Sequential Monte Carlo (SMC) inference and systematic parallel resampling. The overall toolchain is a work-in-progress effort, where the GPU engine and the formalization of the core language are rather mature, whereas the compilation process is still at a very early stage.

1 INTRODUCTION

Probabilistic programming is gaining increased popularity and interest among several different research fields, including statistics, machine learning, and programming language communities. A key feature of probabilistic programming languages (PPLs)[1] is the separation between the probabilistic model and the inference machinery. In particular, the expressive class of *universal* PPLs makes it possible to express probabilistic programs as Turing complete programs, where the number of random variables is not known statically, before inference. Although such expressive models can make it possible to model complex behavior and potentially open up for new application domains, it is hard to achieve the same performance as hand-tuned implementations where the model and the inference algorithm are combined and implemented together.

There exists a large body of PPLs, where the majority is based on or embedded into an existing programming language. For instance, WebPPL [Goodman and Stuhlmüller 2014] is implemented in JavaScript, Church [Goodman et al. 2008] in Scheme, Figaro [Pfeffer 2009] in Scala, Anglican [Wood et al. 2014] in Clojure, and Gen [Cusumano-Towner et al. 2019] in Julia. Other languages and runtime systems, such as VentureScript [Mansinghka et al. 2014], STAN [Carpenter et al. 2017], and Birch [Murray and Schön 2018] are languages of their own. These PPLs and systems are compiled or interpreted on CPUs, using various inference techniques, such as Metropolis-Hastings, Hamiltonian Monte Carlo, and Sequential Monte Carlo. One of the first PPLs that could be compiled to a GPU was LibBi [Murray 2015]. LibBi is not a universal PPL and only a few universal languages and libraries exist that target GPGPUs. Specifically, Pyro [Bingham et al. 2019] and Probabilistic Torch [Narayanaswamy et al. 2017] make use of PyTorch [Paszke et al. 2017] for GPU acceleration, and Edward [Tran et al. 2017] is based on TensorFlow [Abadi et al. 2016].

In contrast to previous work, the aim of this work-in-progress effort is to develop a highly efficient framework for compiling *universal probabilistic programs* written in a *core PPL* down to *pure CUDA GPGPU code*, without using a runtime environment such as PyTorch or TensorFlow. The objective is twofold: (i) to enable that domain-specific PPLs (for instance in the biological or mechatronic domain) can be formalized and translated into the core language, and (ii) that

[1]See http://probabilistic-programming.org/ for an extensive listening of available PPLs

Authors' addresses: Daniel Lundén, KTH Royal Institute of Technology, Department of Computer Science, dlunde@kth.se; Joey Öhman, KTH Royal Institute of Technology, Department of Computer Science, joeyoh@kth.se; David Broman, KTH Royal Institute of Technology, Department of Computer Science, dbro@kth.se.

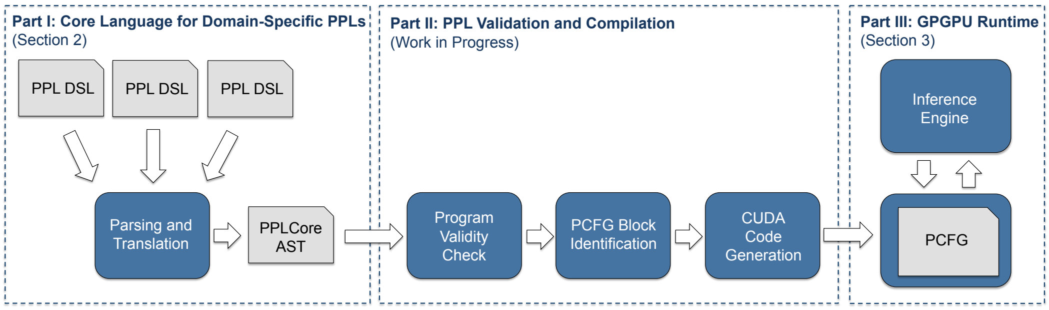


Fig. 1. The figure gives a high-level overview of the overall toolchain. The gray rectangular boxes are artifacts, such as programs, abstract syntax trees (ASTs), or probabilistic control flow graphs (PCFGs). These artifacts are processed by the rounded boxes in blue.

the compiled programs run as efficient on the GPU, as if they were hand-optimized with prior knowledge of the model.

Fig. 1 depicts the overall vision of the toolchain. The toolchain is divided into three parts. In Part I, domain-specific PPLs are parsed and translated into a formal abstract syntax tree (AST) of an intermediate language, called *PPLCore*. The domain-specific PPLs are tailored for specific domains, for instance, DSLs for medical analysis, phylogenetic models, or specialized languages for modeling and analysis of mechatronic systems. We give a brief introduction to PPLCore in Section 2.

The target runtime platform (Part III) is depicted to the right in the figure. The GPGPU runtime system consists of two parts: (i) the inference engine, and (ii) a *probabilstic control flow graph (PCFG)*. Currently, we support SMC inference with parallel systematic resampling [Murray et al. 2013]. The PCFG is a specialized intermediate language in the framework, where each node in the graph consists of C code that can be compiled to the GPU using the CUDA framework. We discuss the GPGPU runtime in Section 3.

Part II—PPL validation and compilation—translates PPLCore programs into a PCFG. This part consists of three main phases: (i) the *program validity check* rules out higher-order programs that cannot be compiled to the GPU (for instance closures or the use of data structures that require garbage collection), (ii) *PCFG block identification* analyzes the PPLCore program and constructs the CFG structure, and (iii) *CUDA code generation* generates the actual C code, which is later compiled using the CUDA compiler. Since the development of this part is at a very early stage, we do not discuss the details of Part II further in this extended abstract.

The overall work-in-progress toolchain is currently being redesigned and developed as part of Miking [Broman 2019], a framework for constructing efficient compilers and language environments. An earlier prototype version of an interpreted PPLCore is available as open source[2].

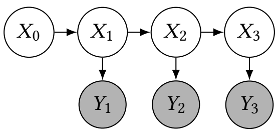
2 A UNIVERSAL CORE LANGUAGE FOR DOMAIN-SPECIFIC PPLS

As part of this work, we develop a new intermediate language called PPLCore: a core PPL based on the lambda calculus, containing a minimal set of probabilistic constructs in order to make it universal. These constructs correspond to the fundamental constructs found in commonly used universal PPLs such as Anglican [Wood et al. 2014], WebPPL [Goodman and Stuhlmüller 2014], and Birch [Murray and Schön 2018]. In particular, PPLCore includes built-in and user-definable probability distributions supporting *sampling*, and a construct for *weighting* executions. The latter

[2]https://github.com/miking-lang/pplcore

```
let observe = lam v. lam dist. weight (logpdf v dist) in

let lgss xprev data = match data with
  | [] -> xprev
  | y:ys ->
    let x = sample (normal (xprev + 2) 1) in
    observe y (normal x 1);
    resample ();
    lgss x ys in

let data = [24.39,25.47,29.62] in
let x0 = sample (normal 0 100) in
lgss x0 data
```

(a) A program written in a concrete PPLCore syntax, used to illustrate the compilation to the SMC GPGPU framework.



(b) A Bayesian network representation of the program in (a).

Fig. 2

construct enables the encoding of Bayesian statistical inference problems, which are widely applied in statistics and machine learning.

In order to demonstrate the idea of compiling a high-level probabilistic program to the GPGPU framework, consider the program in Fig. 2a, written in a concrete syntax for PPLCore. It encodes a rudimentary linear Gaussian state space model, involving an object traveling at a constant speed, on average, in a one-dimensional space. The objective is to approximate the current location of this object, given a set of noisy observations of its current and previous positions. These observations are encoded in the program by using the `weight` construct. Fig. 2b illustrates the corresponding Bayesian network for the program. The current location is X_3, and the observations are Y_1, Y_2, and Y_3.

Note that PPLCore is a small *intermediate language*, aimed as a target language when translating from a domain-specific PPL. Hence, the goal is not to write concrete PPLCore programs, although there exists a concrete syntax (as already exemplified in Fig. 2a). Furthermore, the intermediate language is not only intended as a target language, but also as an intermediate language for performing various optimization phases. For instance, we also plan to incorporate a variant of our previous work on delayed sampling [Murray et al. 2018].

In the next section, we will illustrate how to compile the example from Fig. 2a to our SMC GPGPU framework. In order to do this, we include an explicit `resample` construct in the language (notice its position in Fig. 2a), indicating where resampling should occur when running SMC. The placement of these `resample` statements in arbitrary programs will be done automatically through program analysis and automatic transformation of the AST. Our previous work on automatic alignment [Lundén et al. 2018] is a step in this direction. In particular, the placement has implications for the PCFG Block Identification from Fig. 1—some resampling placements might not be allowed when compiling to the GPGPU due to performance reasons.

3 CODE GENERATION AND AN SMC GPGPU RUNTIME

In order to transform PPLCore programs into efficient CUDA programs, we have developed the novel concept of a probabilistic control flow graph (PCFG). The inference engine (recall Fig.1, Part III) operates during runtime over the PCFG. These blocks are C functions, and a program consists of one or more such blocks. During execution of a block, a *program counter* (PC) is modified, indicating which block to run after the current one has finished. More specifically, this program counter indexes an array of pointers to all available blocks.

Compilation of the PPLCore program in Fig. 2a to a PCFG is done by splitting the program at resample points, and constructing corresponding blocks. Fig. 3a shows the first generated block in the example. In this block, a variable `x` is set (corresponding to line 12 in Fig. 2a) in the *program state*—a data structure available throughout the execution. Both the `PC` and the program state, among other things, are part of the *particle state*, illustrated in Fig. 4a. The list of data from Fig. 2a, line 11, is not initialized in the `init` block. Instead, the data list is initialized elsewhere as a global data structure. In order to keep track of our current location in this list, we use a variable `t` in the program state as an index to it. By doing this, the recursion in the next block, `lgss`, can be written as an iteration. Next, the block increments the `PC`, which in this case means that we will move to the next block, `lgss`. The last line that sets `RESAMPLE` to false informs the SMC inference algorithm that no resample should take place after this block.

The second block, shown in Fig. 3b, corresponds to the recursive code from lines 3 to 9 of the previous example. We do not pass the data list as an argument, since it is stored globally. To access it, we instead use a pointer `dataP` and our index variable `t` in the program state. The `WEIGHT` construct is analogous to `weight` in PPLCore (the `observe` function is inlined). In the end of the block, a check is performed to see whether the last observation has been processed. If this is the case, the `PC` is incremented and points to *null*, terminating execution after finishing the current block. Finally, `RESAMPLE` is set to true, indicating that we should resample after this block. This corresponds to

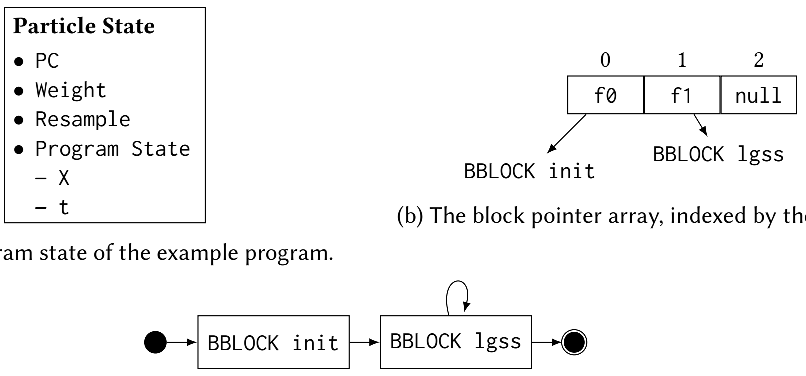
```
BBLOCK(init, progState_t, {
    PSTATE.x = BBLOCK_CALL(sampleNormal, 0, 100);
    PSTATE.t = 0;

    PC++;
    RESAMPLE = false;
})
```

(a) The first basic block of the example program, corresponding to the initialization code.

```
BBLOCK(lgss, progState_t, {
    floating_t* dataP = DATA_POINTER(data);

    PSTATE.x = BBLOCK_CALL(sampleNormal, PSTATE.x + 2.0, 1);
    WEIGHT(logPDFNormal(dataP[PSTATE.t], PSTATE.x, 1.0));

    if(++PSTATE.t == NUM_OBS)
        PC++;

    RESAMPLE = true;
})
```

(b) The second basic block of the example program, corresponding to the recursive code.

Fig. 3



(a) The program state of the example program.

(b) The block pointer array, indexed by the PC.

(c) The PCFG corresponding to the example program.

Fig. 4

the call to `resample` in PPLCore. Fig. 4b shows the array of pointers to the various blocks, indexed by the PC. Fig. 4c shows the PCFG in its entirety.

What remains to be explained is how the PCFG connects to the SMC framework. We assume basic familiarity with SMC inference and terminology (see, e.g., Naesseth et al. [2019] for an introduction). The procedure is as follows:

1. Setup
   - Program specific data setup
   - Move data to GPU
   - Setup basic block function pointers
2. Allocate and initialize particles
3. For each particle, run its current block
   - Perform sampling
   - Update the weight
   - Update PC
   - Specify if resampling should be done
4. Resample, if indicated by the particles
5. Terminate if null-pointer, else go to 3

The first step is to set up data, copy it to the GPU, and to create the block function pointer array. In the second step, execution starts in the SMC framework by allocating and initializing particles. Step three launches CUDA kernels on the GPU, executing the current block for each particle in parallel. The macros used in the code example from Fig. 3 (e.g., `BBLOCK` and `WEIGHT`) expands into inference code that manages particle data on the GPU. Step four resamples the particles, if required (`RESAMPLE = true` for all particles). Finally, if the PC of the particles point to *null*, the SMC algorithm terminates, otherwise, the particles start executing the next block.

4 CONCLUSION AND FUTURE WORK

In this paper, we give a brief overview of a new effort of creating an efficient toolchain for compiling domain-specific PPLs into GPGPU code. In particular, a key aspect of the work is the intermediate language called PPLCore, which is designed to be minimal, but still capture the most common aspects of state-of-the-art PPLs.

One of the main design objectives of the toolchain is to achieve high inference performance. As a consequence, the next step is to conduct a comprehensive empirical evaluation of the overall toolchain. We have so far only conducted some limited experiments with non-trivial state-of-the-art phylogenetic models (inference over evolutionary trees) by encoding the models directly as PCFGs. As for future work, we plan to complete the whole toolchain and to perform the empirical evaluations by comparing real models (such as phylogenetic models and latent dirichlet allocation models) by encoding the same model in several different PPL languages. Such an evaluation between different environments is unfortunately not very common, and our hope is that this can lead to a general benchmark that can drive performance research within the community forward.

REFERENCES

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vol. 16. 265–283.

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research* 20, 28 (2019), 1–6.

David Broman. 2019. A Vision of Miking: Interactive Programmatic Modeling, Sound Language Composition, and Self-Learning Compilation. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering (SLE '19)*. ACM, 55–60.

Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A probabilistic programming language. *Journal of Statistical Software* 76, 1 (2017).

Marco F Cusumano-Towner, Feras A Saad, Alexander K Lew, and Vikash K Mansinghka. 2019. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 221–236.

Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: A Language for Generative Models. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence (UAI'08)*. AUAI Press, Arlington, Virginia, United States, 220–229.

Noah D Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org. Accessed: 2020-1-10.

Daniel Lundén, David Broman, Fredrik Ronquist, and Lawrence M. Murray. 2018. Automatic Alignment of Sequential Monte Carlo Inference in Higher-Order Probabilistic Programs. *arXiv e-prints*, Article arXiv:1812.07439 (Dec. 2018), arXiv:1812.07439 pages. arXiv:cs.PL/1812.07439

Vikash Mansinghka, Daniel Selsam, and Yura Perov. 2014. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099* (2014).

Lawrence Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas Schön. 2018. Delayed Sampling and Automatic Rao-Blackwellization of Probabilistic Programs. In *Proceedings of Machine Learning Research : International Conference on Artificial Intelligence and Statistics (AISTATS)*. PMLR.

Lawrence M. Murray. 2015. Bayesian State-Space Modelling on High-Performance Hardware Using LibBi. *Journal of Statistical Software* 067, i10 (2015).

Lawrence M. Murray, Anthony Lee, and Pierre E. Jacob. 2013. Parallel resampling in the particle filter. *arXiv e-prints*, Article arXiv:1301.4019 (Jan 2013), arXiv:1301.4019 pages. arXiv:stat.CO/1301.4019

Lawrence M Murray and Thomas B Schön. 2018. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control* 46 (2018), 29–43.

Christian A. Naesseth, Fredrik Lindsten, and Thomas B. Schön. 2019. Elements of Sequential Monte Carlo. *Foundations and Trends in Machine Learning* 12, 3 (2019), 307–392.

Siddharth Narayanaswamy, T Brooks Paige, Jan-Willem Van de Meent, Alban Desmaison, Noah Goodman, Pushmeet Kohli, Frank Wood, and Philip Torr. 2017. Learning disentangled representations with semi-supervised deep generative models. In *Advances in Neural Information Processing Systems*. 5925–5935.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

Avi Pfeffer. 2009. *Figaro: An Object-Oriented Probabilistic Programming Language*. Technical Report 137. Charles River Analytics.

Dustin Tran, Matthew D Hoffman, Rif A Saurous, Eugene Brevdo, Kevin Murphy, and David M Blei. 2017. Deep probabilistic programming. *arXiv preprint arXiv:1701.03757* (2017).

Frank Wood, Jan Willem Meent, and Vikash Mansinghka. 2014. A new approach to probabilistic programming inference. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*. 1024–1032.