Synthesizing Code for GPGPUs from Abstract Formal Models

Gabriel Hjort Blindell* ghb@kth.se

Christian Menne* chris.f.menne@gmail.com

Ingo Sander* ingo@kth.se

*Department of Electronic Systems
School of Information and Communication Technology
KTH Royal Institute of Technology, Sweden

***Abstract*—Today multiple frameworks exist for elevating the task of writing programs for GPGPUs, which are massively data-parallel execution platforms. These are needed as writing correct and high-performing applications for GPGPUs is notoriously difficult due to the intricacies of the underlying architecture. However, the existing frameworks lack a formal foundation that makes them difficult to use together with formal verification, testing, and design space exploration. We present in this paper a novel software synthesis tool – called *f2cc* – which is capable of generating efficient GPGPU code from abstract formal models based on the synchronous model of computation. These models can be built using high-level modeling methodologies that hide low-level architecture details from the developer. The correctness of the tool has been experimentally validated on models derived from two applications. The experiments also demonstrate that the synthesized GPGPU code yielded a $28\times$ speedup when executed on a graphics card with 96 cores and compared against a sequential version that uses only the CPU.**

***Keywords*—Analytical models, computational modeling, system-level design, multicore processing**

I. Introduction

We are experiencing a seemingly never-ending improvement in computational processing capacity. The past decades have yielded faster, denser, and more complex chips, and the processing units are increasingly being composed into multicore platforms which require complicated communication and scheduling schemes. This results in an incredible challenge that system developers need to face in managing the growing complexity of systems. To make matters worse, low-level implementation details must be considered in order to produce, not only correct, but efficient systems. This problem is especially notorious for *general purpose graphics processing units* (GPGPUs). GPGPUs are massively parallel execution platforms that have emerged from the graphics card technology whose processing capacity have grown to such an extent that they can be considered affordable small-scale supercomputers. But the underlying architecture exhibits many intricacies, making it difficult to exploit. For instance, in order to reach maximum performance it is paramount that the GPGPUs's registers, on-chip memories, and caches are used efficiently, but optimizing the usage of one resource often has a negative impact on another. Moreover, the convoluted addressing schemes required for distributing data across the threads are mechanical, tedious, and error-prone to manage manually. Hence, to manually write applications that are both correct and efficient when executed on a GPGPU is an extremely challenging and error-prone task.

Although there exist several frameworks for elevating the task of GPGPU programming, they are all based on programming methodologies that hinder the use of automated tools for tasks such as verification, testing, and design space exploration. To mitigate these issues we present in this paper a novel software synthesis tool – called *f2cc*[1] – that generates GPGPU code from applications which are represented as *abstract formal models*. These models have a solid formal foundation based on the theory of models of computation [15] and are devoid of low-level details regarding implementation and target architecture, which raises the level of abstraction for the system developer and enables the use of formal system design tools. In this case we use ForSyDe for modeling the applications. Hence *f2cc* promotes an application design flow that is "correct by construction" [8] by allowing the system developer to focus on *what* the system is meant to do rather than *how*, which lowers the development cost. Most importantly, *f2cc* enables system developers to take advantage of GPGPUs without needing to have extensive and in-depth knowledge about the underlying architecture.

The paper makes the following contributions:

- We present a novel software synthesis tool (*f2cc*) that is capable of generating GPGPU code from abstract formal models based on the synchronous model of computation. Using a formal framework for application design enables the potential to perform verification, testing, and design space exploration in an automated fashion. Other advantages of the tool include:
    - *Modeling framework independence. f2cc* provides a flexible XML + C input format and frontend support which can be extended to support models created using different formal modeling methodologies.
    - *Adaptive and stand-alone code.* The GPGPU code produced by *f2cc* adapts itself to the properties of the graphics card at runtime, and does not depend on any proprietary libraries in order to be compiled or executed.
    - *Flexible data type support. f2cc* allows the developer to use custom-made `struct`s as data types in the models, thus facilitating the application design.

[1]Source code is available at http://forsyde.ict.kth.se/trac/wiki/ForSyDe/f2cc

- We describe the methods and algorithms devised for *f2cc*, including an $O(n)$ algorithm for finding a process schedule for synchronous models containing feedback loops.
- We present experiments that demonstrate the correctness and efficiency of *f2cc* for GPGPU code synthesized for a Mandelbrot generator and an industrial-scale image processor. Compared against the performance of a hand-written CPU version, the GPGPU code generated by *f2cc* yielded a speedup of $28\times$ when executed on a graphics card equipped with 96 cores.

The rest of the paper is organized as follows. Section II briefly describes the GPGPU platform and introduces ForSyDe, the formal modeling methodology currently supported by *f2cc*. Section III explains the software synthesis process, the techniques and methods applied, and its current limitations. Section IV gives the results from the experiments that were performed to validate the tool. Section V covers related work and discusses existing frameworks which elevate the task of GPGPU programming. Lastly, Section VI concludes the paper and lists future work.

II. Background

A. GPGPUs

GPGPUs are enhanced versions of GPUs [10], [18], which are processing units specifically designed for rendering image frames. As image rendering is generally a parallel process where pixels can be generated independent from one another, the GPU quickly evolved into a massively data-parallel platform. Recognizing this vast computational resource, program developers urged the manufacturers to augment the GPU with functionality that would allow execution of applications written in general-purpose programming languages such as C. When adapted to GPGPUs applications have often yielded a significant performance increase, at times reaching orders of magnitudes in speedup [9].

A well-known family of GPGPUs is CUDA [10], [14], [17], [18], which is developed by NVIDIA. The CUDA platform, as illustrated in Figure 1, consists of clusters of *streaming multiprocessors* (SMs) which are connected to a dedicated DRAM commonly referred to as the *global memory*. Each SM contains 8 *streaming processors* (SPs) or *CUDA cores* which share the same fetch/dispatch unit, register file, and instruction cache. The SM also consists of a set of various on-chip memories: a *shared memory*, sometimes denoted as *scratchpad memory*, which is an application-controlled cache; a *constant cache*, which retains constant values; and a *texture memory*, which is used to cache neighboring cells in a 2D data matrix. With the DRAM bandwidth usually being the main performance bottleneck, these caches are used to reduce the amount of traffic to and from the global memory.

After having copied the input data to the global memory, the GPGPU is accessed through a *kernel invocation* which spawns a set of threads to be executed on the GPGPU. These threads are bundled into *thread blocks*, which in turn are allocated onto the SMs. A small set of threads is then randomly selected from each thread block for execution on the SPs. Thread-context switches are made with virtually zero overhead, and provided that there is an abundance of threads the GPGPU can hide long latency operations through continuous thread switching. This makes the GPGPU a *throughput-oriented architecture* [10]. All thread blocks allocated to an SM share the same register file and other resources such as caches. This means that if a thread block uses too much of any resource, the maximum number of residential thread blocks per SM will be reduced. Fewer thread blocks means fewer threads to swap in and out to hide long latency operations, which in turn decreases the performance. Since the caches are very limited – the sizes are in the order of tens of kilobytes – meticulous care must be taken to not claim too much of any cache per thread block, and make optimal use of the allotted slice. Hence the main challenges of exploiting GPGPUs are as follows:

- *Adapting the application to fit the data-parallel execution platform.* Even algorithms that are inherently parallel may need to be redesigned in order to avoid performance-hampering issues such as *thread divergence* [14], which may occur when the code contains branch instructions.
- *Determining how to layout the input data and thread configuration.* The data needs to be packaged in such a way that it can be accessed from a thread using its thread and thread block IDs. Thus, there should be a correlation between the data layout and the thread configuration.
- *Determining which GPGPU resources to use, and how, in order to achieve optimal performance.* The GPGPU

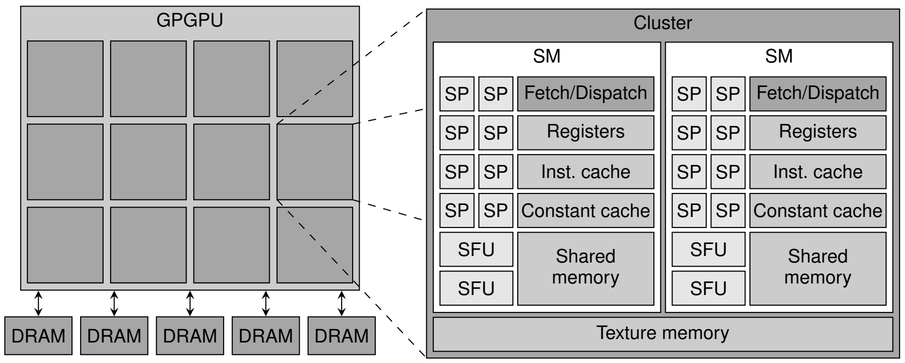


Fig. 1. Overview of the NVIDIA CUDA platform [17], [18]

contains several resources such as caches and on-chip memories that can greatly boost performance. However, it is not always clear how each can be used for a particular application, often forcing new algorithms to be considered.

- *Determining whether utilizing the GPGPU is beneficial.* Even if all performance-inhibiting problems related to the GPGPU itself are dealt with, it is still possible that the code runs *slower* on the GPGPU than on the CPU. For example, the CPU may be relatively more powerful than the GPGPU, or there may not be enough computational complexity in the kernel to sufficiently amortize the GPGPU overhead of moving data between the main RAM and the GPGPU RAM.

B. ForSyDe

ForSyDe (Formal System Design) [19], [1] is a formal design methodology for embedded systems. It consists of a set of libraries, currently available in Haskell and SystemC, that enable modeling of systems at a high level of abstraction where the functionality of a system is detached from its implementation. The libraries support several *models of computation* (MoCs), but in the context of this paper only the *synchronous MoC* is considered. The synchronous MoC is based on the *perfect synchrony hypothesis* [3], which assumes that data propagation and process execution take zero time (i.e. processes produce their output values immediately as their inputs arrive). This assumption leads to a simple and elegant mathematical model that fits nicely with a large class of data flow applications and with the underlying mechanisms of the GPGPU platform. The synchronous MoC is also base for the family of synchronous languages like Esterel [4] and Lustre [11], for which mathematical methods exist for performing verification and testing. Another similar modeling framework is StreamIt [21], where program hierarchy is modeled using predefined structures.

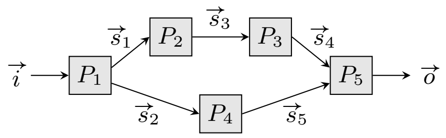


Fig. 2. Example of a ForSyDe model

Systems are modeled in ForSyDe as hierarchical concurrent process networks, where *processes* communicate by means of *signals* (see Figure 2). Processes are created using predefined *process constructors* that take *side effect-free functions* and *values* as arguments. This concept of process constructors leads to a clean separation between *communication* and *computation*: communication and model of computation is expressed by the process constructor; and computation is specified by the arguments of the process constructor. In Figure 3, we see this exemplified using *mooreSY*, a process constructor for a Moore finite state machine that applies the synchronous MoC. As arguments, *mooreSY* takes two functions ns and o and a value s: ns specifies the calculation of the next state; o specifies the calculation of the output value; and s specifies the initial state. *mooreSY* is but one of many process constructors that ForSyDe provides.

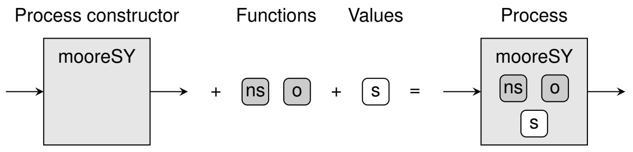


Fig. 3. Process constructor concept

This separation of concerns is exploited when writing the ForSyDe models to text files. Using GraphML – the input format of *f2cc* (see Section III-A) – the hierarchical structure of the process network is expressed in XML, and the computation is given as C code. Hence, the description of the structure is separated from the description of the computation. We want to point out that other formalisms that support the synchronous MoC, and provide a similar separation of communication and computation as ForSyDe, can be used in conjunction with *f2cc* as described in Section III-A.

III. Synthesis process

f2cc operates by first parsing an input file containing the model and converted into an internal model representation. Then a series of semantically-preserving optimizations are applied, and lastly the model is synthesized into code. This process is also illustrated in Figure 4. We will begin by discussing the input format, and then proceed with examining the internals of *f2cc* (more details are available in [13]).

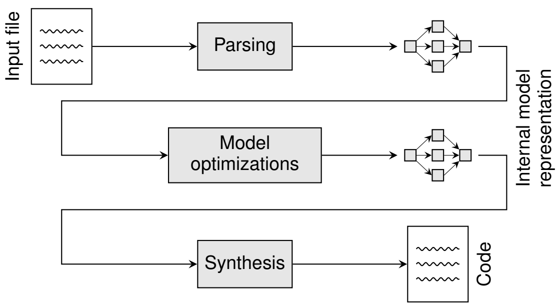


Fig. 4. Overview of the synthesis process

A. Input format

Once a model has been designed, it is passed to *f2cc* in the form of a *GraphML file*. Similar output can be generated from ForSyDe-SystemC using introspection [1], and converting it to GraphML is trivial. GraphML [5] is a standardized format based XML in which graphs can be represented in a formal manner, and allows the process functions to be provided as data annotated to the nodes. The process functions are defined as side effect-free C functions, meaning they must not depend on any external state such as global variables or dynamically allocated memory. An example of such an GraphML file is available in Listing 1, whose model is illustrated in Figure 5. Note that the input file contains no GPGPU-related information, thereby completely hiding any implementation-specific details about the target platform from the developer. Moreover, the input format does not require data types to be specified for signals and processes which do not have a C function as argument. Instead, the data types for these will be automatically inferred by *f2cc* during synthesis (see Section III-D). This makes for a

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml>
 <graph id="test" edgedefault="directed">
  <node id="in">
   <data key="process_type">InPort</data>
   <port name="out" />
  </node>
  <node id="out">
   <data key="process_type">OutPort</data>
   <port name="in" />
  </node>

  <!-- Processes -->
  <node id="inc">
   <data key="process_type">ParallelMapSY</data>
   <data key="procfun_arg">
    <![CDATA[
    int func(const int arg) {
      return arg+1;
    }
    ]]>
   </data>
   <data key="num_processes">3</data>
   <port name="in" />
   <port name="out" />
  </node>

  <!-- Signals -->
  <edge source="in" sourceport="out"
        target="inc" targetport="in" />
  <edge source="inc" sourceport="out"
        target="out" targetport="in" />
 </graph>
</graphml>
```

Listing 1. Example of an input file to *f2cc*

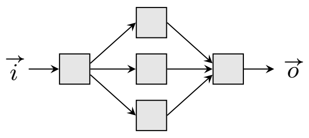


Fig. 5. Illustration of the model declared in Listing 1

very versatile format that allows models to be created using any formal modeling framework, provided the models can be converted into the expected input format and hold the same semantic meaning. Since the format is human-readable the input files can even be written by hand. If desired, *f2cc* can also be extended with additional frontends to support for another input format.

B. Model optimizations

In order to take advantage of the parallel nature of GPGPUs, the model needs to exhibit a certain level of data parallelism which can either be declared *implicitly* or *explicitly*. Implicit data parallelism is declared through a network of processes, known as a *data-parallel component*, while explicit data parallelism is declared via a single processes that semantically entail the functionality of entire data-parallel components.

There are many patterns of data parallelism. One such pattern is a data-parallel component that accepts an input array, applies one or more functions on every element or non-overlapping range of elements, and produces an array as output (see Figure 6a). While simple, it is an important and powerful pattern that allows modeling of many embarrassingly parallel problems. We call this the *split-map-merge* pattern: first, the array is *split* into multiple data sets, then one or more functions are *mapped* onto each data set, and lastly the results are *merged*. We have devised a special process constructor called *parallelMapSY* (see Figure 6b) for explicit declaration of this pattern (which is equivalent to StreamIt's *splitjoin* construct), and support for exploiting it for efficient execution on GPGPUs is already available in *f2cc*. Our tool

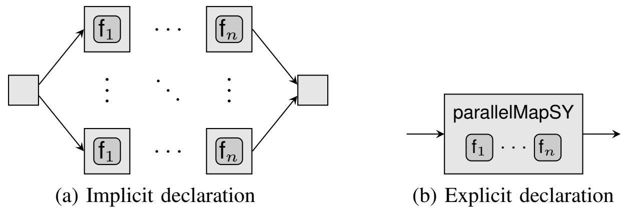


Fig. 6. The split-map-merge pattern

is also capable of combining chains of *map* processes into a single *map* process in order to reduce the amount of function calls, which we refer to as *process coalescing*.

Since discovering explicitly declared data parallelism is trivial (the data-parallel component is contained in a *single* process), the challenge lies in detecting implicitly declared data parallelism where a *cluster* of processes needs to be combined into a data-parallel component. For the split-map-merge pattern, this is done using an $O(n^2)$ depth-first algorithm which searches for pairs of *split* and *merge* processes. For a given pair, it then checks whether the data flow is contained between the two processes, and whether the intermediate processes between the *split* and *merge* processes consist of chains of *map* processes only. Once identified, the implicitly declared data-parallel components are replaced by single processes of the type which corresponds to the explicit declaration of the patterns (e.g. a data-parallel component arranged as the split-map-merge pattern will be replaced by a *parallelMapSY* process). This simplifies the later process schedule and code generation stages as each such process will constitute a complete and separate GPGPU kernel. It is possible to add support for exploitation of explicitly declared patterns of data parallelism while leaving out discovery of implicit declarations. In such instances, models containing implicit declarations will still be synthesized, however the data-parallel component will be executed sequentially on the CPU instead of in parallel on the GPGPU.

C. Process schedule generation

As order of execution has an impact on the final output, a process schedule must adhere to the effects of the perfect synchrony hypothesis (i.e. that process execution and data propagation between processes take zero time). Finding such a schedule for sequential models is straight-forward – one just needs to traverse the model along its signals – but diverging data flows and feedback loops complicates this task.

Listing 2 shows the algorithm which was devised for *f2cc*. It is based on a recursive depth-first search approach: starting from the model outputs, each process P is visited by traversing the model in the reverse data flow direction until no further traversing is possible (if the traversal was done in the forward data flow direction, then no schedule would be generated for models with no inputs). Partial schedules are then built and concatenated until the entire model has been traversed, and a set of visited processes is maintained in order to avoid redundant search and provide termination when feedback loops (i.e. cyclic data flow) is encountered. However, the synchronous MoC does not allow feedback loops without using a kind of delay element, and the placement of the this element within the loop affects the final schedule (as illustrated in Figure 7). In this context, a delay element is a process that for an input sequence $\langle v_1, \ldots, v_n \rangle$

```
function FindSchedule($M$) returns schedule for model $M$
  schedule $\leftarrow$ empty list; queue $\leftarrow$ empty queue
  $visited_G$ $\leftarrow$ empty set
  for each output signal $S$ of $M$ do
    add process of $S$ to head of queue
  while queue is not empty do
    $visited_L$ $\leftarrow$ empty set
    $P$ $\leftarrow$ head of queue; remove head from queue
    {p_schedule, ip} $\leftarrow$ FindPartialSchedule($P$, $visited_G$,
      $visited_L$, queue)
    if ip = "at beginning" then
      insert p_schedule before head in schedule
    else
      insert p_schedule after process ip in schedule
    add $visited_L$ to $visited_G$
  return schedule

function FindPartialSchedule($P$, $visited_G$, $visited_L$, queue)
  if $P \in visited_G$ then
    return {empty list, $P$}
  if $P$ is a delay element then
    add preceding process of $P$ to end of queue
    return {$P$, "at beginning"}
  schedule $\leftarrow$ empty list
  ip $\leftarrow$ "at beginning"
  if $P \notin visited_L$ then
    add $P$ to $visited_L$
    for each preceding process $O$ of $P$ do
      {p_schedule, new_ip} $\leftarrow$ FindPartialSchedule($O$,
        $visited_G$, $visited_L$, queue)
      append p_schedule to schedule
      if new_ip $\neq$ "at beginning" then
        ip $\leftarrow$ new_ip
    append $P$ to schedule
  return {schedule, ip}
```

Listing 2. Process scheduling algorithm

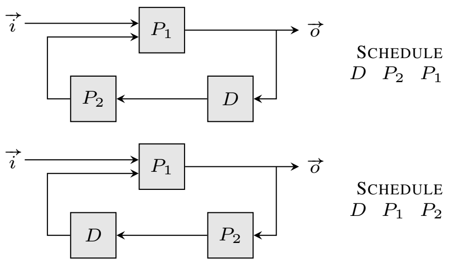


Fig. 7. Examples of two models with corresponding process schedules

shifts the sequence in time by inserting an initial delay value s, thus producing $\langle s, v_1, \ldots, v_n \rangle$ (in ForSyDe this element is implemented using the *delaySY* process constructor). Our scheduling algorithm handles these situations by effectively acting as if the inbound edges to the *delaySY* processes had been removed. Using data structures that can be accessed in constant time, the algorithm finishes in $O(n)$ time.

D. Signal data type inference

Signals are the vessels in the model through which data is propagated from one process to another. It is therefore appropriate to retain the notion of signals by implementing them as data containers in the synthesized code, typically as either global or local C variables. However, the data types of the signals are not immediately available from the formal model as they are only explicitly specified as part of the C functions, which only appear in the *map* processes. Hence the signals connected to other processes such as *delay*, *split*, and *merge*, the data types have to be automatically inferred. In *f2cc* this is done using an algorithm that recursively traverses the model until

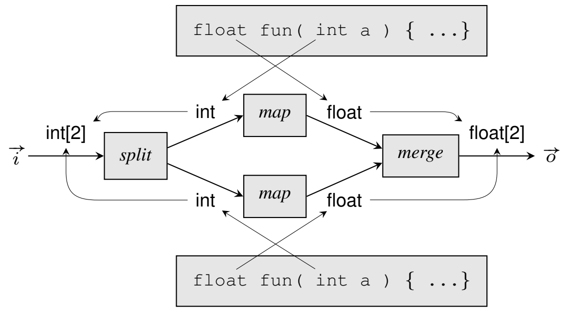


Fig. 8. Example of how the data types propagates along the signals

signal connected to a *map* process is found. This information is then propagated backwards to the original signal, and hence the data types ripple from signal to signal across the model as shown in Figure 8. By caching the data type found for each signal, the algorithm takes $O(n)$ time to find the data types of all signals in a model. Failing to infer the data type for a signal indicates that the model is invalid, which is also reported by *f2cc*.

E. GPGPU code generation

In Listing 3 we provide the CUDA code generated by *f2cc* using the GraphML file given in Listing 1 as input. For each data-parallel component, which at this stage will have been converted into a single-process equivalents, *f2cc* will generate a set of wrapper functions (see Figure 9). The C function that implements the computational part of the data-parallel component – we will from now on call this the *data function* – is wrapped by a *kernel function*. The kernel function is responsible for providing the input data based on the thread block and thread IDs, managing the shared memory, and preventing out-of-bound threads from executing. In the case of the *split-map-merge* pattern, utilizing shared memory is done by first copying all the data required by the data function from global memory to the shared memory, and then passing the appropriate pointer to the data function. The kernel function is then wrapped inside an *invoker* function, which manages memory transfers between the CPU and GPGPU and sets up the thread configuration. The thread configuration is decided at runtime such that the size of the thread blocks is the maximum size supported by the graphics card (since the number of concurrent thread blocks per SM is limited to 8 at a time, it is necessary to use as large thread blocks as possible in order to achieve optimal performance). However, if the generated code makes use of shared memory then the threads may require more shared memory than available, which reduces the number of thread blocks per SM and thus inhibits performance. To prevent this an algorithm is employed which incrementally decreases the thread block size and calculates the amount of unused shared memory for that size. This continues until either the amount reaches zero, or until the number of thread blocks per SM becomes greater than 8 (upon which the configuration with the least waste is selected). Some GPGPU execution environments may also enforce a maximum execution time for each kernel invocation, and *f2cc* embeds additional code for handling such situations when generating the invoker function.

```
__device__ int finc_func1(const int arg) {
  return arg+1;
}

__global__ void finc_kernel(const int* in, int* out, int offset) {
  unsigned int gi = (blockIdx.x * blockDim.x + threadIdx.x) + offset;
  extern __shared__ int in_cached[];
  if (gi < 3) { // Prevents out-of-bound threads from executing
    int in_i = threadIdx.x * 1; int global_in_i = gi * 1;
    in_cached[in_i + 0] = in[global_in_i + 0];
    out[gi] = finc_func1(&in_cached[in_i]);
  }
}

void finc_kernel_wrapper(const int* in, int* out) {
  int* d_in; int* d_out; struct cudaDeviceProp prop;

  // Get GPGPU device information
  cudaGetDeviceProperties(&prop, 0);
  int max_t_per_b = prop.maxThreadsPerBlock;
  int smem_per_sm = (int) prop.sharedMemPerBlock;
  int full_utc = max_t_per_b * prop.multiProcessorCount;

  // Prepare device and transfer input data
  cudaMalloc((void**) &d_in, 3 * sizeof(int));
  cudaMalloc((void**) &d_out, 3 * sizeof(int));
  cudaMemcpy((void*) d_in, (void*) in, 3 * sizeof(int), cudaMemcpyHostToDevice);

  // Execute kernel
  struct KernelConfig c;
  if (prop.kernelExecTimeoutEnabled) {
    int num_t_left = 3; int offset = 0;
    while (num_t_left > 0) {
      int num_t_exec = num_t_left < full_utc ? num_t_left : full_utc;
      c = calculateBestKernelConfig(num_t_exec, max_t_per_b, 1 * sizeof(int),
                                    smem_per_sm);
      finc_kernel<<<c.grid, c.threadBlock, c.sharedMemory>>>(d_in, d_out, offset);
      int num_executed_threads = c.grid.x * c.threadBlock.x;
      num_t_left -= num_executed_threads;
      offset += num_executed_threads;
    }
  }
  else {
    c = calculateBestKernelConfig(3, max_t_per_b, 1 * sizeof(int), smem_per_sm);
    finc_kernel<<<c.grid, c.threadBlock, c.sharedMemory>>>(d_in, d_out, 0);
  }

  // Transfer result back to host and clean up
  cudaMemcpy((void*) out, (void*) d_out, 3 * sizeof(int), cudaMemcpyDeviceToHost);
  cudaFree((void*) d_in);
  cudaFree((void*) d_out);
}

void executeModel(const int* in1, int* out1) {
  // Declare and alias signal array variables with model input/output arrays
  const int* vmodel_in_to_inc_in = in1;
  int* vinc_out_to_model_out = out1;

  // Execute processes
  finc_kernel_wrapper(vmodel_in_to_inc_in, vinc_out_to_model_out);
}
```

Listing 3. CUDA code generated for the input file given in Listing 1. Note that the code has been manually edited and shortened in order to fit this paper

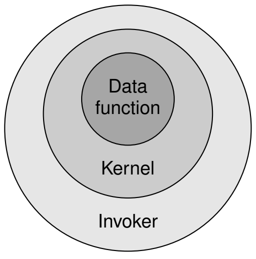


Fig. 9. The function stack used by *f2cc* for executing functions on the GPGPU

F. Process execution and data propagation

Executing the processes is straight-forward: the code simply needs to invoke the processes' C functions (if the process is of such type) with the appropriate parameters according to the generated process schedule. Data propagation is then done via a set of C variables – one for each signal – which are passed as parameters to the C functions. Part of the future work will be to identify and remove redundant signal variables, which will reduce the number of signal-to-signal copying operations and thus increase performance. Delay element values are stored in static C variables as these need to be retained between model invocations. For signals consisting of multiple values, the tool builds the necessary arrays and manages the addressing such that each process gets the correct input value.

G. Limitations

So far we have focused on supporting discovery and exploitation of the split-map-merge pattern. Hence *f2cc* does not yet provide full support for all process types that are available in ForSyDe, but the process type support as well as the recognition and exploitation of additional patterns of data parallelism can be extended by defining new process types, adding recognition of the new process types in the frontends, and extending the backend to synthesize the appropriate C or GPGPU code for each process type.

The synthesized GPGPU code also does not make full use of all available CUDA resources. Currently only the shared memory is considered, but this is simply because the potential resource usage is dependent on the pattern of data parallelism being exploited. In the case of the split-map-merge pattern, there is little or no gain in using the shared memory, or any other resource for that matter. Hence, these resources can be put to better use when additional patterns are available.

Another significant drawback is that no cost analysis is currently performed of whether it is actually beneficial to offload parallel computations onto the GPGPU. This means that, depending on the performance of the GPGPU and CPU, the generated CUDA code may run *slower* than if had been executed sequentially on the CPU.

IV. Experiments

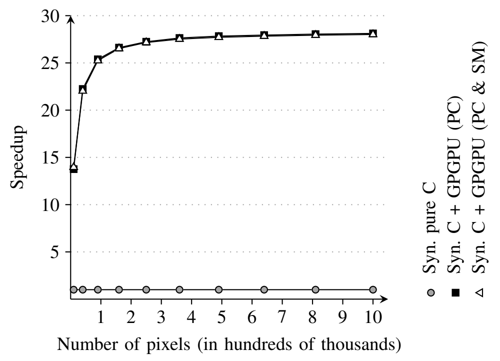
To validate the correctness and efficiency of *f2cc*, the tool was applied on models derived from two applications: a Mandelbrot generator, and an industrial-scale image processor. For each model, a pure C implementation of the final code and multiple implementations where the data-parallel components are executed on the GPGPU were generated and evaluated. The output and performance of the synthesized C code was compared with a hand-written C version which was executed by a single thread on the CPU. The C code and GPGPU code was compiled using *g++* v.4.6.1 and *nvcc* release 3.2 v0.2.1221, respectively, with all optimizations disabled. The test cases were executed on an Intel Core i7-2600 at 3.40 GHz, 16 GB DDR3 RAM at 1333 MHz, and an NVIDIA Quadro 600 with 96 CUDA cores, 1 GB DDR3 RAM. Each test case was run 10 times and then an arithmetic mean average was calculated from the results.

A. Mandelbrot tests

Generating Mandelbrot images is a task exhibiting an abundance of data parallelism. Each pixel coordinate is converted into a corresponding coordinate within a rectangular coordinate window in the complex plane. From the complex coordinate an integer value is computed which determines to whether the coordinate is part of the Mandelbrot set. In these tests, the window was bounded by $(-1/4, -1/4)$ and $(1/4, 1/4)$. Its model consists of a single data-parallel component, and when expressed using *parallelMapSY* the model shrinks to a single process. The performance results of the synthesized C and GPGPU code are given in Figure 10a. We see that the

Problem size (pixels)	Execution time (s)			
	Pure C impl.		C + GPGPU impl.	
	HW	Syn.	PC	PC & SM
10,000	1.33	1.33	0.10	0.10
40,000	5.30	5.30	0.24	0.24
90,000	11.92	11.91	0.47	0.47
160,000	21.19	21.19	0.80	0.80
250,000	33.09	33.10	1.21	1.22
360,000	47.66	47.66	1.72	1.73
490,000	64.87	64.86	2.33	2.34
640,000	84.72	84.72	3.03	3.04
810,000	107.23	107.21	3.82	3.84
1,000,000	132.39	132.42	4.71	4.73

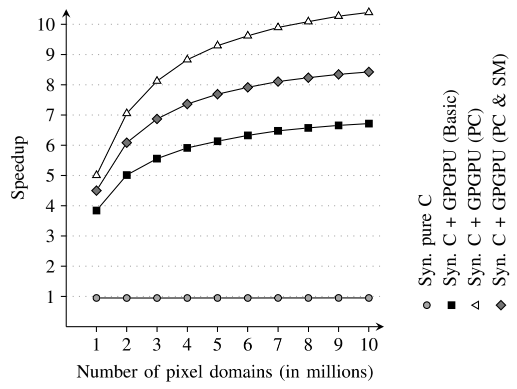
Maximum measured standard deviation: 2.53%



(a) Test results from the Mandelbrot model

Problem size (pixel domains)	Execution time (s)				
	Pure C impl.		C + GPGPU impl.		
	HW	Syn.	Basic	PC	PC & SM
1,000,000	0.38	0.40	0.10	0.08	0.09
2,000,000	0.77	0.81	0.15	0.11	0.13
3,000,000	1.15	1.21	0.21	0.14	0.17
4,000,000	1.53	1.62	0.26	0.17	0.21
5,000,000	1.92	2.02	0.31	0.21	0.25
6,000,000	2.30	2.42	0.36	0.24	0.29
7,000,000	2.68	2.82	0.41	0.27	0.33
8,000,000	3.06	3.22	0.47	0.30	0.37
9,000,000	3.45	3.62	0.52	0.34	0.41
10,000,000	3.83	4.03	0.57	0.37	0.45

Maximum measured standard deviation: 0.86%



(b) Test results from the image processing model

Fig. 10. Experimental data. HW stands for *hand-written*, and "Syn." refers to the code generated by *f2cc*, where PC and SM indicates whether process coalescing or shared memory on the GPGPU was used, respectively

synthesized C code performs equally with the hand-written C version, and the synthesized C + GPGPU code performs $28\times$ better. The relatively low speedup for small input data sizes is due to the restricted amount of computations which can be offloaded to the GPGPU. As the input data size increases, so does the extent to which the GPGPU overhead can be amortized. Since there is very little input data reuse and no data sharing, using shared memory has no impact on the performance. The output of the synthesized C code was exactly equal to that of the hand-written version, but for the GPGPU code the the integer values were slightly different for some coordinates. We believe this discrepancy to be caused by the floating point units whose architecture differ between the GPGPU and CPU.

B. Image processing tests

The second model was derived from an existing industrial-scale image processor application provided by XaarJet AB, a company specializing in piezoelectric drop-on-demand ink-jet printing. At its core, the model consists of a single data-parallel component composed of 3 data-parallel segments. Using the *parallelMapSY* process constructor and process coalescing, this model also shrinks to a single process. The details of the C functions will not be covered as not to disclose any industry secrets. The performance results are given in Figure 10b. Again, the synthesized C code is on par with the hand-written version, and the synthesized C + GPGPU code is $10\times$ faster. This relatively low speedup is due to lack of computational complexity in the model, and the continued slope indicates that greater speedup is achievable with even larger problem sizes. Furthermore, as the input data size per thread is much greater than in the Mandelbrot model, the performance of the synthesized GPGPU code is reduced when the shared memory is used since doing so will limit the number of thread blocks that can simultaneously reside in an SM, which in turn lowers performance. Like with the Mandelbrot tests, the synthesized code produces slightly different output when executed on the GPGPU compared to the CPU. Since floating point operations are involved, we again believe the differing architectures of FPUs between the CPU and GPGPU to be the cause.

V. Related work

Existing GPGPU programming frameworks can generally be divided into three categories: *declarative-based frameworks*, where code to execute on the GPGPU is marked by annotations; *library-based frameworks*, where the core is implemented as programming libraries; or *domain-specific languages* (DSLs), where the framework is embedded into an existing programming language.

Declarative-based frameworks include *hi*CUDA [12] and OpenMP-to-GPGPU [16]. In *hi*CUDA parallelizable C code is annotated with pragma directives which control dynamic memory allocation, thread configuration, work distribution per thread over loops, and more. The *hi*CUDA compiler then processes the code to generate GPGPU kernels based on the annotations. The framework therefore relieves the developer from having to produce the data addressing schemes, handle the CPU-GPGPU data transfers, and manage the shared memory. Consequently, *hi*CUDA relies on the developer to identify and tweak the code for execution on the GPGPU. In OpenMP-to-GPGPU the existing OpenMP pragma notations are used to identify parallelizable code, but these miss the information about thread blocks and shared memory. In both cases, the frameworks completely lack a formal foundation and are thus unsuitable for automated verification and testing.

Library-based frameworks include Thrust [2] and SkePU [7], which are both implemented in C++ and provide a set of *skeletons* (a skeleton is akin to the notion of process constructors used in ForSyDe, see Section II-B). The developer provides the computation part to the skeletons, and the skeletons then decide the appropriate thread configuration, memory management, and other execution-related details. Unlike Thrust, SkePU is also capable of generating code for multi-core CPUs, OpenCL, and single-threaded C code. But although the use of skeletons provides a more formal base than pragmas, they are not based on a well-defined model of computation, and can therefore not be analyzed using existing mathematical tools. Moreover, the skeletons do not extend into the rest of the application.

Two GPGPU-oriented DSLs, both embedded in Haskell (a purely functional programming language), include Accelerate [6] and Obsidian [20]. Accelerate also uses the notion of skeletons by providing a collection of arrays and array operations that can be offloaded on a GPGPU. In order to compile into an application that can be executed on a GPGPU, Accelerate comes with a Haskell-to-CUDA compiler which translates Accelerate-based Haskell programs into CUDA-annotated C code. Obsidian is similar to Accelerate but instead provides a collection of *combinators* that allow array functions to be converted into GPGPU kernels. Through the combinators, the developer gains access to use of the shared memory and insertion of synchronization barriers, but this requires the developer to know when and how to use the combinators in order to match the underlying architecture of the GPGPU. Moreover, neither is based on a well-defined MoC, which again inhibits automated verification and testing.

VI. Conclusion and future work

In this paper we have presented *f2cc*, a software synthesis tool which is capable of synthesizing abstract formal models based on the synchronous model of computation into GPGPU code. Unlike existing frameworks which elevate the task of GPGPU programming, *f2cc* operates on abstract formal models which enables the potential to apply automated tools on the applications for verification, testing, and design space exploration. Through experimental validation, we have shown that the tool produces correct and high-performing GPGPU code from its input models.

Future work will primarily focus on integrating the results of [1] to achieve a completely automated flow from ForSyDe-SystemC to GPGPU code. In addition, the number of recognizable and exploitable patterns of data parallelism that can be executed on the GPGPU will be expanded. This in turn entails making more efficient use of its resources, such as shared memory, constant cache, and texture memory. Another consideration is more efficient signal handling methods to eliminate redundant memory transfers between execution of separate data-parallel components.

References

[1] S. Attarzadeh Niaki, M. Jakobsen, T. Sulonen, and I. Sander. Formal Heterogeneous System Modeling with SystemC. In *Forum on Specification and Design Languages*, FDL 2012, pages 160–167, Vienna, Austria, Sept. 2012.

[2] N. Bell and J. Hoberock. Thrust: A Productivity-Oriented Library for cuda. In W. W. Hwu, editor, *GPU Computing Gems*, chapter 26, pages 356–371. Morgan Kaufmann Publishers, Jade edition, 2011.

[3] A. Benveniste and G. Berry. The Synchronous Approach to Reactive and Real-Time Systems. In *Proceedings of the IEEE*, volume 79, pages 1270–1280, 9 1991.

[4] G. Berry and L. Cosserat. The ESTEREL Synchronous Programming Language and Its Mathematical Semantics. In S. Brookes, A. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 389–448. Springer Berlin, 1985.

[5] U. Brandes, M. Eiglsperger, and J. Lerner. *GraphML Primer*, Jun. 2004. URL: http://graphml.graphdrawing.org/primer/graphml-primer.html (last visited 2014-05-19).

[6] M. M. T. Chackravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell Array Codes with Multicore GPUs. In *Proceedings of the 6th Workshop on Declarative Aspects of Multicore Programming (DAMP'11)*, pages 3–14, 2011.

[7] U. Dastgeer, C. W. Kessler, and S. Thibault. Flexible Runtime Support for Efficient Skeleton Programming on Hybrid Systems. In *Proceedings of the International Conference on Parallel Programming (ParCo'11)*, Heraklion, Greece, 2011.

[8] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of Embedded Systems: Formal Models, Validation, and Synthesis. In *Proceedings of the IEEE*, volume 85, pages 366–387, Mar. 1997.

[9] M. Garland, S. L. Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel Computation Experiences with cuda. *Micro, IEEE*, 28:13–27, 2008.

[10] M. Garland and D. B. Kirk. Understanding Throughput-Oriented Architectures. *Communications of the ACM*, 53:58–66, Nov. 2010.

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Dataflow Programming Language LUSTRE. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[12] T. D. Han and T. S. Abdelrahman. *hi*CUDA: High-Level GPGPU Programming. In *IEEE Transactions on Parallel and Distributed Systems*, volume 22, pages 78–90, Jan. 2011.

[13] G. Hjort Blindell. Synthesizing Software from a ForSyDe Model Targeting GPGPUs. Master's thesis, KTH Royal Institute of Technology, Stockholm, Sweden, 2012.

[14] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors*. Morgan Kaufmann Publishers, 2010.

[15] E. A. Lee and A. Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, Dec. 1998.

[16] S. Lee, S. Min, and R. Eigenmann. OpenMP-to-GPGPU: A Compiler Framework for Automatic Translation and Optimization. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'09)*, volume 44, pages 101–110, Apr. 2009.

[17] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia Tesla: A Unified Graphics and Computing Architecture. *Micro, IEEE*, 30:39–55, 2010.

[18] J. Nickolls and W. J. Dally. The GPU Computing Era. *Micro, IEEE*, 30:56–69, 2010.

[19] I. Sander and A. Jantsch. System Modeling and Transformational Design Refinement in ForSyDe. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 23, pages 17–32, Jan. 2004.

[20] J. Svensson, K. Claessen, and M. Sheeran. GPGPU Kernel Implementation and Refinement Using Obsidian. In *Proceedings of the International Conference on Computational Science (ICCS'10)*, volume 1, pages 2065–2074, May – Jun. 2010.

[21] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196, 2002.