LambdaJIT: A Dynamic Compiler for Heterogeneous Optimizations of STL Algorithms

Thibaut Lutz
University of Edinburgh
thibaut.lutz@ed.ac.uk

Vinod Grover
NVIDIA Corporation
vgrover@nvidia.com

Abstract

C++11 introduced a set of new features to extend the core language and the standard library. Amongst the new features are basic blocks for concurrency management like threads and atomic operation support, and a new syntax to declare single purpose, one off functions, called lambda functions, which integrate nicely to the Standard Template Library (STL). The STL provides a set of high level algorithms operating on data ranges, often applying a user defined function, which can now be expressed as a lambda function. Together, an STL algorithm and a lambda function provides a concise and efficient way to express a data traversal pattern and localized computation.

This paper presents LambdaJIT; a C++11 compiler and a runtime system which enable lambda functions used alongside STL algorithms to be optimized or even re-targeted at runtime. We use compiler integration of the new C++ features to analyze and automatically parallelize the code whenever possible. The compiler also injects part of a program's internal representation into the compiled program binary, which can be used by the runtime to re-compile and optimize the code. We take advantage of the features of lambda functions to create runtime optimizations exceeding those of traditional offline or online compilers. Finally, the runtime can use the embedded intermediate representation with a different backend target to safely offload computation to an accelerator such as a GPU, matching and even outperforming CUDA by up to 10%.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel programming; D.3.4 [*Programming Languages*]: Processors - Optimization; D.3.4 [*Programming Languages*]: Processors - Retargetable compilers

Keywords C++11 Closures; Heterogeneous Systems; JIT Compilation; Partial Specialization

1. Introduction

The C++ standard provides a collection of generic functions in the Standard Template Library, such as `std::for_each` and `std::transform`, which allow to represent common traversal patterns, or skeletons, on complex data-structures.

FHPC '14, September 4, 2014, Gothenburg, Sweden.

http://dx.doi.org/10.1145/2636228.2636233

```
1  int main(){
2   using namespace std;
3   vector<float> v1{.1,.2,.3,.4,.5},
4                 v2{.6,.7,.8,.9,.0};
5   vector<float> z(v1.size());
6   float a = 10.f;
7
8   transform(begin(v1),end(v1),begin(v2),begin(z),
9             [a](const float &x, const float &y) {
10              return a * x + y;
11            });
12 }
```

Figure 1. Saxpy implementation in C++11. The STL algorithm are used to define the iteration space: here `std::transform` (line 8) takes two input ranges and an output range. The operation function is expressed as a lambda function (lines 9-10), which takes an element of each input range and returns an element of the output range.

This algorithmic collection relies on user defined functions which, until the most recent standard, were either functions or functor objects. Since C++11, a new feature allows to simply and clearly define and use anonymous functions, called lambda functions. Those are usually very succinct and simple one-off tasks which have a single, well defined purpose. Figure 1 shows an example of an application using the standard algorithm transform with a lambda function to express a Single Precision A-X Plus Y (SAXPY) calculation.

Similar language constructs are also present in C#, Java or Python, but C++ Lambda expressions have a distinct feature which we exploit for optimization: its ability to capture the context around their declaration. The most common usage of the lambda functions in C++ is its natural integration to the STL.

While the current standard does not provide any parallel version of the algorithms, parallelism can be inferred from the structure of the standard algorithms and a semantic analysis of the lambda functions to detect eventual side effects. Automating parallelization of these algorithms allows the LambdaJIT framework to take advantage of the performance boost achievable on modern multicore CPUs, while avoiding to re-write the code.

To further improve the performance, one might want to re-target the code for dedicated hardware accelerators; which outperform traditional processors by a large factor, but for a very restricted class of problems.

One type of accelerator that has been used increasingly for high performance computing are Graphics Processing Units (GPUs). Their high number of cores and SIMD architecture offer tremendous performance for embarrassingly data parallel problems, at a cost of intensive programming efforts.

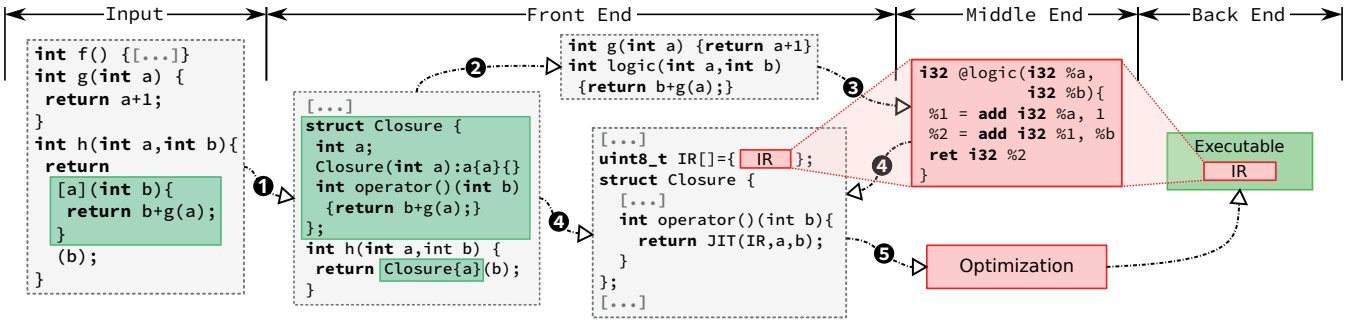


Figure 2. Overview of the LambdaJIT compiler. The compiler takes C++11 as input. The lambda functions are transformed to Closure classes, where the captured variables are transformed to attributes and the body of the lambda is transformed to a function operator (step ❶). For each generated closure, the AST is cloned, and a closure logic function is created to isolate the domain logic of the lambda and its dependencies (step ❷). Each Closure logic module is then lowered to IR (step ❸), before being integrated to the original input file as a binary array (step ❹). The compilation then proceeds normally on the modified input code (step ❺). The final executable contains a representation of the logic in IR for each generated Closure class.

Typically, expert programmers use dedicated programming languages, such as CUDA[13] or OpenCL[9]. This requires a complete re-write of existing code and a comprehensive knowledge of GPU architecture in order to achieve good performance. Even then, the performance is tied to a particular hardware and optimal implementations vary depending on the hardware.

This paper presents LambdaJIT, a compiler and runtime system for auto-parallelization and transformation of one-off lambda functions into re-compilable, re-targetable snippets of code. This allows us to take advantage of multi-core performance and to perform aggressive compiler optimizations which are out of reach for traditional offline compilers. The safety of these transformations are guaranteed by the LambdaJIT runtime, even when offloading computation to an accelerator such as a GPU, and we can dynamically adapt to heterogeneous systems.

The input of LambdaJIT compiler is standard C++ code, compatible with any other C++ compiler. The output is a binary which contains the compiler internal Intermediate Representation (IR) code for a small portion of the program, allowing the runtime system to re-compile and optimize fragments of code at runtime. In summary, the contributions of this work are:

- Integration of new C++ features in the compiler analysis. This enables us to statically check whether an algorithm can be parallelized at compile time.
- A new application for lambda functions. Exploiting their unique properties in a Just-in-time (JIT) compiler, we can efficiently optimize the code at runtime with minimal impact on the binary size. The JIT compiler is able to target a very small portion of the code while yielding high optimization potential.
- A new approach to GPGPU programming: the compiler is able to spot potential candidates for GPU execution in standard C++ code. By integrating its intermediate representation into the binary, the runtime is free to dynamically select an accelerator and use its backend to offload computation.

2. LambdaJIT Compiler

This section presents our offline compiler and details the various compilation steps. The compiler is a C++11 compiler based on clang. It differs from traditional compiler in its compilation stages. While a normal compiler does a single pass from frontend to backend, our compiler performs multiple passes and alters the internal frontend representation between passes. This allows us to integrate some middle-end components into the output binary.

The compiler transformations target lambda functions. Lambda functions are recognizable by their syntax: they are composed of a capture list, a parameter list and a body. An example of lambda declaration is shown inside the input code in Figure 1. The capture list is a distinct feature of C++ lambdas. It enumerates local variables visible inside the scope where the lambda is declared, and used inside the body. Lambda functions can capture variables by reference or value. A capture by value will create a copy of the object, whereas a capture by reference will create a reference on the object. The parameter list and the body of the lambdas have a similar syntax as a standalone function. The return type is either explicit or deduced from the body.

Figure 1 shows an overview of these transformations. A first pass does source to source transformation to lower Lambda functions to standalone Closure objects. This transformation described in more detail in Section 2.1.

For each Closure class generated, the Abstract Syntax Tree (AST) is duplicated, and simplified to keep only the dependencies of the closure function. A standalone function is generated from the capture list and the body of the lambda. This function is then lowered to the compiler internal representation and injected back into the original AST. This process is explained in Section 2.2.

Finally, the compiler performs some static analysis, described in Section 2.3, to guide the targeting decisions of the runtime system, and generates an executable containing the IR representation of the lambda functions.

2.1 Lambda Declaration Lowering

In order to have better control over the execution of lambda functions, the LambdaJIT compiler transforms their declaration into standalone classes providing a function operator, called a functor object. Doing so transformed their types from an unnamed type to a well defined type, allowing the framework to generate type dependent traits and overloads used by the runtime system. Since lambda functions have been introduced primarily as a short form to define functor objects, this transformation can be applied in most cases.

Our compiler extracts the definition of a lambda function whenever possible and transforms it in a standalone functor object. This function object is called a *Closure Object*. The following modifications are applied:

- Capture list generation: the captured variables are transformed to class attributes. Their type is deduced from the combination of the variable type and the capture modifier. For example a variable of type *T* captured by reference will generate a member of type *T&*. A variable of type *T* captured by copy will have a type *T*, but is considered constant unless the lambda is qualified with the *mutable* keyword. The identifiers of the class members are the same as the captured variables. In Figure 1, the lambda captures *a*, which is an integer type. In the closure object generated after step ❶, an attribute of the same type and same identifier is created in the generated closure object.
- Function operator generation: the signature of the function operator is deduced from the parameter list and the return type of the lambda. In Figure 1, the lambda function takes an integer *b* as parameter and implicitly returns an integer. Note that since the lambda is not mutable, the function operator should be declared constant in the generated closure, which is not shown in Figure 1 due to space constraints.
- Constructor generation: Since a lambda declaration is also an implicit instantiation, all captured variables must be initialized to capture their state at the site of the lambda declaration. The signature of the explicit constructor corresponds to the list of captured variables. Each attribute is initialized in the same order as they are captured. Finally the original declaration of the lambda function is replaced with a call to the constructor and the captured list is transformed to an argument list.

Note that these transformations cannot always be applied. For example, a lambda can use a local type, declared inside a function scope. In this case it cannot be extracted outside the scope. Another example is the use of *decltype* inside a lambda body without a capture list. These corner cases can be spotted by the compiler and will cancel the transformation, in effect considering the lambda as plain code. For the sake of simplicity, we do not detail these cases in this paper and only consider lambda function that can be transformed to Closure objects.

2.2 Closure Logic Function

Once the lambda declarations are transformed to functor object definitions, the compiler extracts the domain logic of each lambda, defined in its body, into a separate standalone function. This step provides isolation of the computation in a single function, decoupled from the Closure type.

This is crucial to our approach since this function will integrated into the binary in different forms, allowing the runtime to manipulate it during the execution of the program.

The signature of this function is deduced by concatenating the capture list and the parameter list, which make the arguments of the function, and the return type is the same as the lambda's. The body of the function is the same as the function operator of the Closure object and the identifier names are preserved. We will call this function the *Closure Logic* function in the remainder of this paper, since it represents the business logic of the lambda function.

The closure logic function is lowered to the compiler's internal representation (step ❷). The compiler then serializes this intermediate representation in a binary stream and injects it back into the frontend as byte array in the source code (step ❸).

Finally, the function operator of the Closure object is replaced with a call into the LambdaJIT runtime. This call provides a reference to the bytecode to execute and the execution parameters. The runtime dynamically selects a target for the IR and forwards the execution.

2.3 Compiler Analysis

The goals of the compiler analysis is to guide the runtime system by integrating some information in the executable and to facilitate rest of the compilation process. The analysis pass still operates on the frontend and injects source code in the AST.

In order to make sure the lambda function can safely be run on a given target, the code is analyzed statically at compile time and the information is passed to the runtime through type traits, which are generated and injected to the source code, and as metadata in the intermediate representation stored in the binary. The following type traits are generated for each closure functor type:

- *is_closure<T>*: true if T is a generated closure, false otherwise. This is used to define overload of the STL algorithms. Instead of calling the algorithms defined in the standard library, the compiler will select special implementations of the algorithm which hook into the LambdaJIT runtime to select an execution target and policy at runtime.
- *is_parallel<T>*: true if T is a closure and has no side effect. To be executed in parallel, a functor object must meet this requirement. It guarantees that the lambda object does not access a non atomic shared object. For example, capturing a variable by reference and modifying it breaks this requirement, since it presents a concurrency issue.
- *is_gpu_compatible<T>*: true if T is parallel and is eligible for GPU execution. Because the GPU execution is constrained by its programming model and architecture, many C++ functionalities cannot be translated. In particular, the compiler checks for unresolved symbols which are not built-in functions. These are usually third party library calls which are not defined on the GPU. The analysis also checks that the capture list does not contain pointer or references, since the GPUs usually do not have shared memory with the host.

The analysis examines each lambda functions but also all their dependencies. Exploring the entire call graph is necessary to make sure the requirements are met.

3. LambdaJIT Runtime System

This section describes the runtime, which contains the JIT compiler and the interfaces to communicate with the JITed code and the accelerators. The runtime has a choice between three execution targets for a lambda function. A pre-compiled version of each lambda function is present in the binary. This allows for an immediate execution of the Closure, without compiling it. The second execution policy is JIT-native. The runtime uses the IR representation of a closure to compile it with the same backend as the executable. Finally, the runtime can choose to offload computation to a GPU by selecting the GPU runtime.

The runtime dispatcher is the keystone of the re-targetable lambdas mechanism. It is in charge of loading the intermediate representation from the binary, orchestrating the JIT compilations and picking the target for each algorithm.

During static initialization, an instance of the dispatcher is created and aggregates the intermediate representation, represented as bitcode, from the multiple translation units into a collection of modules, which contains the definition of all the logic functions.

When an algorithm is executed with a closure object for the first time, the closure object asks the dispatcher to pick an execution target. This decision is based on the properties of the algorithm being executed and the characteristics of the closure object. Some optimization heuristics are in place to guide this decision, but the general priority ordering in descending order is as described in 3. The ordering for choosing a policy is as follow:

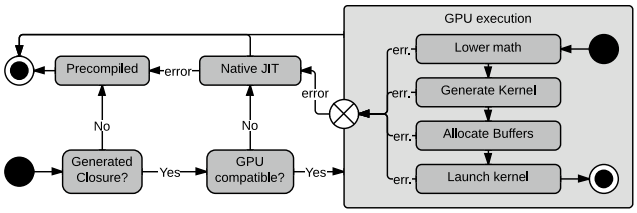


Figure 3. Runtime heuristics to select an execution target and failsafe mechanism. The runtime will try the execute the algorithm either on the GPU, in parallel or sequentially depending on the properties of the algorithms and the properties of the Closure object.

- GPU execution: if the data is large enough and both the data and the closure object meet the GPU execution requirement, the preferred target is the GPU.
- Parallel native code: if the code is not compatible with a GPU execution, the algorithm will be parallelized if the closure object meets the concurrency requirements.
- Specialized native code: if the code cannot be executed in parallel, the closure object is still specialized to the runtime values before being executed.
- Pre-compiled version: if the range used for the algorithm is very small, or the closure object cannot be parallelized nor specialized, or the JIT compiler thread is oversubscribed, the dispatcher falls back to the pre-compiled version.

Note that this failsafe mechanism, shown in Figure 3, guarantees that the execution of the portable cannot abort since we can always execute at least the pre-compiled version.

The remainder of this section describes the execution model for each individual targets. The automatic parallelization is described for the pre-compiled version, however it is orthogonal to the native target choice and can be applied to either the pre-compiled, the JIT compiled or the specialized version of the closure.

3.1 Pre-compiled and JITed Native Runtimes

The runtime has several options when running on the code on the native target. Since the offline compiler integrates both a pre-compiled version of the closure logic functions and their IR, the code can either be executed directly or JIT compiled from the IR using the Native CPU Target.

The Native CPU target recompiles the closure using the same backend as used for the program binary. While JIT compiling the code is expensive, the cost is amortized by optimizations that could not have been performed offline. These optimizations, like partial specialization and inlining, are described in detail in Section 4.

The pre-compiled target bypasses the JIT compiler and executes a version of the lambda function compiled offline. This target is used to amortize JIT compilation costs. It also provides a safe fall back if the compiler fails during the JIT compilation.

For both the pre-compiled version and the JIT compiled code, the runtime can take advantage of the static analysis information generated by our compiler to automatically parallelize the code.

During offline compilation, the calls to standard algorithms are replaced with hooks into the runtime. When executing an algorithm, the runtime selects the target and forward the parameters. If the lambda function meets the concurrency requirements, the runtime will select a parallel version of the algorithm. The parallel runtime for the STL is based on standard threads. The data range is divided in subranges and dispatched to multiple threads for execution. We only provide a parallel version for a subset of the STL.

```
1  void logic(const float a, float &x, float &y){
2   return a*x+y;
3  }
4  __global__
5  void kernel(const float a,
6              float *v1,float *v2,float *out,
7              unsigned range){
8   int idx=blockIdx.x*blockDim.x+threadIdx.x;
9   while(idx<range){
10   out[idx]=logic(a,v1[idx],v2[idx]);
11   idx+=blockDim.x*gridDim.x;
12  }
13 }
```

Figure 4. CUDA equivalent to the IR generated by the GPU runtime for the SAXPY input code shown in Figure 1. The generated kernel determines the array traversal behavior, here a binary transform takes three arrays as ranges, calls the functor on each pair of the input arrays and stores the result in the output array.

The algorithms which are not parallelized default to the sequential execution.

3.2 GPU Runtime

GPUs provide highly parallel systems with good performance at relatively low cost and energy. However, since their architecture typically consists of thousands of wide Single Instruction, Multiple Data (SIMD) units, they are restricted mainly to data parallel problems.

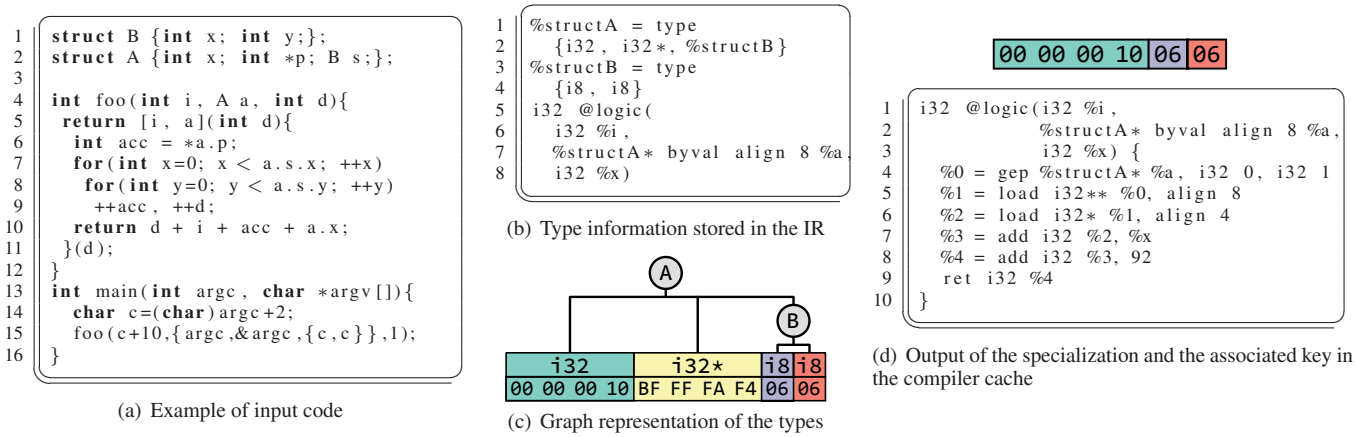
They fit naturally to accelerate STL algorithms exposing data parallelism, such as `for_each`, `transform` or `fill`. This normally requires a re-write of the application to match the restrictions and programming models of GPUs, often in a domain specific language or using specialized frameworks.

The goal of the GPU backend in LambdaJIT is not to translate entire applications radically to make them fit the GPU programming model, but to use GPUs as accelerators for compute-intensive fragments of code which are compatible with this model.

The offline compiler checks whether a particular lambda is compatible with GPU execution, as described in Section 2.3. The runtime also checks that some runtime requirements are met, like the minimal size of the range, which can be adjusted by the user, or the physical presence and availability of a GPU, which is taken from granted when using a GPU specific program. When the runtime decides to offload the computation to a GPU, the closure needs to be adapted to better fit the GPU programming model.

The Closure logic function represents the element function of the algorithm, which is the function applied for each element of the range. This is also the case for most GPU specific languages, which define *kernels* as a special function being invoked for each element of a range. However, since kernels need to compute their position in the array, the closure logic functions are not translated directly into kernels. Instead, a wrapper kernel is created by the framework and invokes the element function. It is generated in the same intermediate representation as the closure logic function. Figure 4 shows a CUDA equivalent of the IR generated by the framework for the SAXPY example shown in Figure 1.

The signature of the kernel is determined by the Closure type and the type of algorithm being applied. Its arguments are the captured data, the size of the range, which is called *grid* in CUDA terminology or the *index space* in OpenCL, and the list of device arrays pointing to the data. The number of arrays depends on the algorithm: `for_each` and `fill` are doing in-place computation and require only one data array, whereas `transform` requires between one array, for in-place transforms, to three arrays in its binary



(a) Example of input code

(b) Type information stored in the IR

(c) Graph representation of the types

(d) Output of the specialization and the associated key in the compiler cache

Figure 5. Example of runtime partial specialization. 5(a) is an input program capturing user defined types for which the attribute values are not known at compile time. 5(b) displays the type information in the IR generated by our compile. 5(c) shows the same type information as a graph, and how the runtime can traverse this graph to get values from the memory, in this case for $argc = 4$. 5(d) is the output of the specialization obtained at runtime. Most of the computation is now simplified, only the computation involving pointers and arguments remain.

operation form. In Figure 4, the first argument of the kernel is the captured floating point variable, passed by value, the two input and one output arrays from the binary transform algorithm and the size of the range.

The body of the kernel uses CUDA specific variables built-in to the language to get the index of the kernel instance in the grid. It then computes the address of the elements of each input array at this offset and finally calls the Closure logic function with the captured list and a reference to the array elements.

Before going through the backend, some simple compiler transformations are applied on the IR module containing the Closure logic and the kernel. These passes translate the calls to built-in mathematics functions or atomic manipulation functions defined in the standard C++ library, which have their equivalent in CUDA. Finally, the module is linked against NVVM, which is a library written in the same IR as the kernel containing definition for the built-in functions. The linked module is then passed to the backend and compiled to PTX code.

The PTX code, which is loaded using the CUDA API. Before launching the kernel, the runtime allocates and initializes memory on the device. The kernel is then started synchronously. Once completed, the data is copied back to the host memory.

Our framework has advantages and disadvantages compared to an application written entirely in CUDA. The main advantage is that one does not require any knowledge of GPU programming in order to use the GPU. Also, the generated binary is more flexible and smaller than a CUDA binary: each generation of device comes with more features, and the CUDA compiler uses those to increase performance. To maintain backward compatibility, a CUDA binary has to integrate several versions of the same device code compiled for different generation of devices, while our binaries only needs a single representation and detect the abilities of the device at runtime and select the appropriate backend.

The main limitation of the GPU target is the poor reuse of device arrays across kernels. Since it is not possible to track the usage of each data containers between STL algorithm calls, the runtime has to assume that the data changed every time: it must re-allocate and copy the data even if it has not changed. GPU specific languages like CUDA and OpenCL puts the programmer in charge of the data transfers explicitly, which exposes buffer re-use across kernels.

4. Online Optimizations

In this section, we present the optimizations performed by the runtime compiler. The runtime JIT compiler exploits characteristics of lambda functions to optimize code further than an offline compiler, as explained in Section refsec:specialization. We also use the IR to do inter-procedural optimization at runtime by providing support for fusable function composition. This API is described in Section 4.2.

4.1 Partial Specialization and Code Versioning

The IR representation of a closure object can be JIT compiled using the native backend. While re-compiling the code adds overhead compared to executing the pre-compiled version, it allows to apply further compiler optimizations which were not available at compile time.

This is especially true for lambda functions: the variables captured by value cannot be modified from outside the lambda and, most of the time, not even from the lambda itself, unless it is explicitly declared mutable. Furthermore, the data is not visible and cannot be accessed from outside the lambda. Whereas functor object can share their attributes, either by defining their visibility or through inheritance or member functions, lambda's captured values give us extreme locality and ownership guarantees. This considerably simplifies the program analysis by providing isolation and local reasoning. Captured values are private and constant by construction throughout the lifetime of the closure instance.

This concept allows programmers to intuitively express computation parameters that they know to be constant, but not necessarily known at compile time, hence are not optimized by traditional compilers. However, since they often represent key parameters of the computation, they are very good candidate for optimization.

Our JIT compiler performs these aggressive optimizations at runtime to dramatically simplify the code in some cases.

Captured values are known as soon as the closure object is instantiated and if they cannot be changed after that point, the compiler can clone the closure logic function for a particular instance and apply aggressive constant propagation and constant folding on the IR. The output is a highly specialized closure, which is still portable since it does not require the backend to specialize the code.

Figure 5 shows an example of specialization involving user defined types. Figure 5(a) shows an example of input where the user declares their own types, `A` and `B`. `B` is an aggregate of primitive types and `A` is a more complex aggregate composed of a primitive type, a pointer and a user defined type. In line 5 to 10, a lambda function is declared inside a function. It captures an integer and a object of type `A` by value and takes an integer as argument. The body of the lambda performs some computation using the captured values and the parameters. Note that by construction, the captured data is invariant. The nested loop could correspond to a matrix traversal of a fixed size for example. In the main function (lines 14 and 15), an object of type `A` is instantiated and the lambda is invoked with values unknown at compile time.

When compiled using our compiler, the type information is represented in the IR, as shown in Figure 5(b). C++ mandates that every instantiated type must be known by the compiler and must be defined in the code. The compiler represents them as an aggregate of primitive types, pointers or reference to another type. Furthermore, the signature of each function contains the type of each argument and their memory alignment.

The runtime uses this information to retrieve the values of class attributes at runtime from memory. Since C++ does not have type introspection, the runtime cannot query the value of individual attributes of a class. Instead, the runtime builds a graph representation of the types, as shown in Figure 5(c), which contains a composition of each type in terms of primitive types and the alignment information. This allows the runtime to retrieve the values of each attribute from a base pointer on an instance of the class. In this case, type `A` is composed of a signed 32-bit integer, a pointer and two signed 8-bit integers. For each attribute, the runtime can deduce the size and the alignment, and fetch its value. In Figure 5(c), the attribute x of the instance of A has a value of 16 and the instance of B contains two values 6.

Since only the immutable data can be propagated, the runtime needs to carefully avoid propagating pointers or references, since their pointee can be changed outside the lambda between instantiation and invocation. In this case, the first integer and the last two can be propagated, since they cannot be changed, but the pointer has to be avoided.

Once the values of the immutable attributes have been decoded, they are inserted in the IR and a set of optimization passes will simplify the code. Since C++ does not have reflection, the signature of the function is preserved but the propagated arguments are not used. The output of the specialization is shown in Figure 5(d). In this case, all the control flow graph disappears since the nested loops become bounded. The logic is replaced with a single add instruction in line 8. The value 92 is computed from an instance where $argc = 4$; the double increment in the nested loop and the final expression can be computed by the compiler:

$$d + 16 + 6 * 6 * (1 + 1) + 4 = d + 92$$

Because the specialization process is time consuming, the specialization are cached by the JIT compiler using a hash of all the propagated values. If the same lambda get instantiated several time with the same captured data, a previous specialization will be reused. Note that we do not apply specialization incrementally because of its cost, so if only a subset of the captured data matches a previous specialization, it is ignored and a completely new specialization is triggered.

Users can control the cache by specifying how many specialization per Closure object the cache has or how many instantiation should trigger a specialization.

One limitation of this method is the overhead introduced by computing the hash when a new instance is created, especially when the specialization does not trigger significant optimizations. However, because of the language constraints and restrictions on captured variables, it is not often the case that a variable is captured by value and has no optimization potential.

4.2 Function Composition and Fusion

In order to express a stream of computation, we provide a polymorphic function wrapper, called `Closure`, which are similar to the standard `std::function`. This class can be used exactly in the same way as the standard wrapper: it can be instantiated from a function reference, a functor object or a lambda function. In the latter case, an internal flag is set and a reference to the Closure object is kept.

Closure API combined with the runtime are able to find function compositions, which are constructs where the output of a Closure is used as input for another Closure. Since the runtime is only able to optimize for Closure objects created from lambda functions, creating a Closure object from a function or a functor object will behave as `std::function` and will not be optimized.

However, if several lambda functions are composed together though Closure objects, the runtime will fuse their IR and inline the calls. In effect, this flattens recursive calls and allows for interprocedural constant folding (which propagates the captures values as well).

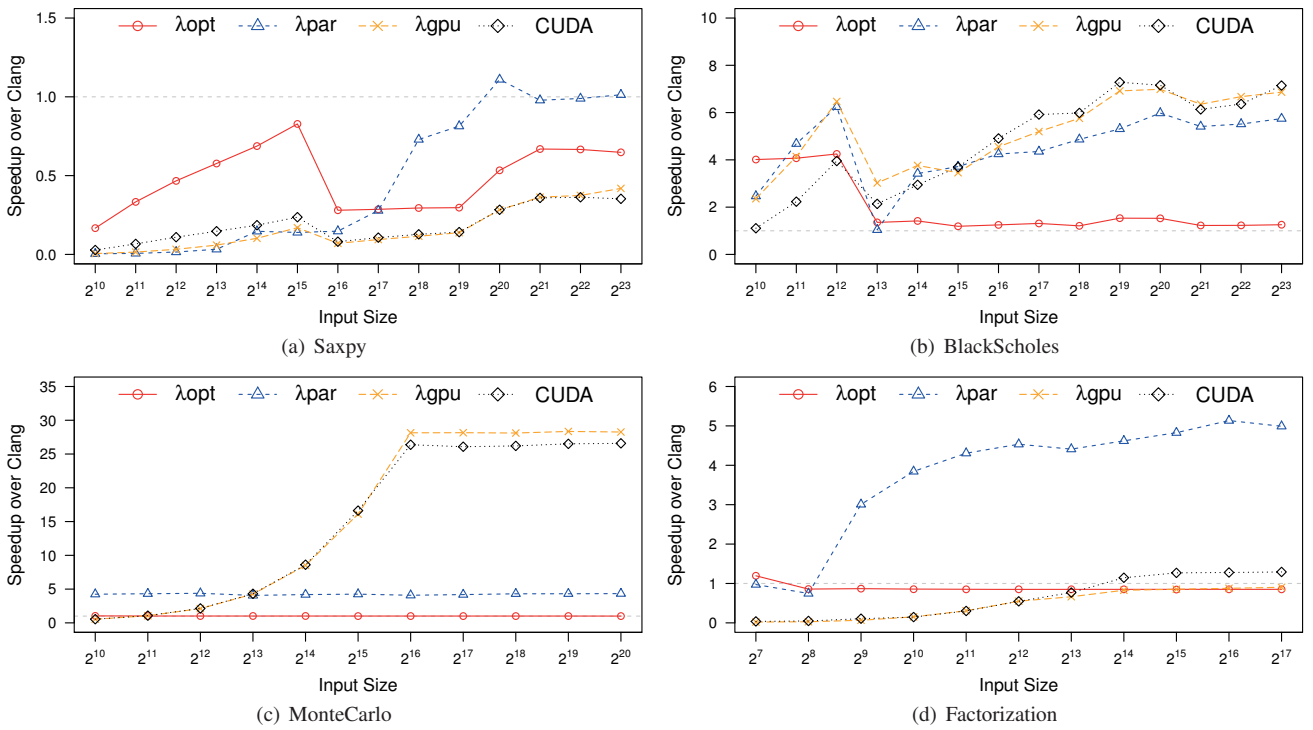
```
1  auto m(char match)
2   ->Closure<const char*(const char*)>{
3   return [match](const char *input)->const char*{
4    if(input == nullptr || *input == 0) return input;
5    else if(*input == match) return ++input;
6    else return nullptr;
7   };
8  }
9  int main(int argc, char *argv[]){
10  Closure<const char*(const char*)> expr;
11  for(char *c = argv[1]; *c != 0; ++c) expr >> m(*c);
12  auto match = expr >> [](const char *p)->bool
13    {return p!=nullptr && *p==0; };
14  cout << match("foo") << endl;
15 }
```

Figure 7. Example of Closure fusion. The function `m` returns a Closure which tests if a character in a string matches a captured char or returns `nullptr` if the string does not match. Line 12 builds a closure checking for the string "foo". When instantiated, the Closure will propagate the captured values and inline the calls, creating a single specialized function.

Figure 7 shows an example of composition. The function `m` returns a Closure object comparing the first character of a C string against a captured value. If the captured character matches, the lambda returns the pointer on the next character in the string, otherwise it returns either `nullptr` or end of string.

Line 10 is a declaration of Closure object. Like `std::function`, a Closure type has a return type and optional input types; here both are pointers on characters.

Line 11 shows an example of fusion. A Closure object defines the stream $>>$ operator to symbolize a composition. This defines a compute stream where a Closure can be appended to the end of a stream if its input type matches the output of the last Closure in the stream. We dynamically chain Closures together to form an expression to compare a C string to the first program argument. Note that this type of composition is not possible with a standard function wrapper. If `m` was returning an std function, a dynamic composition cannot be easily built and would require binding call wrappers. For example, a function matching the string "foo" would have to be defined recursively as:



(a) Saxpy (b) BlackScholes

(c) MonteCarlo (d) Factorization

Figure 6. Evaluation of the LambdaJit runtime. We measure the speedup of three runtime targets compared to the same input source code compiled with clang for a range of applications and input sizes . The three targets are *λopt*, a JIT optimized sequential version, *λpar*, an automatically parallelized version, and *λgpu*, where computation is offloaded to a GPU. To provide a comparison for the later, we also compare against application hand writen in CUDA.

```
1 m('o')(m('o')(m('f')("bar"))); //using std::function
2 (m('f') >> m('o') >> m('o'))("bar"); //using Closures
```

Lines 12 and 13 change the type of the stream by appending another lambda function which checks if the match was successful. The return type of the stream is now a boolean, but its input type is still a C string.

Line 14 invokes the call operator on the Closure. This forces the runtime to realize the composition stream by inlining all the Closures together using their IR. Combined with the partial specialization described in Section 4.1, the generated code is equivalent to a single function containing a sequence of branches testing for hard coded characters,

While the example presented in Figure 7 is trivial, behavioral patterns are common in C++ design and being able to optimize against a combination of patterns at runtime yields good optimization potential.

5. Results

In this section, we evaluate the performance of the LambdaJIT framework. Since LambdaJIT is a C++ compiler, we use the same source code for the baseline and the optimized versions, but the baseline is compiled using a third party compiler.

5.1 Methodology

We run a set of a benchmarks on a system containing a 3.6GHz Intel Core i7-3820 processor with 8G of RAM and a Nvidia Quadro K5000 with 1536 CUDA cores, 4GB GDDR5 memory and a peak performance of 2.1Teraflops. The baselines are compiled using Clang 3.4 for the CPU and the CUDA 6.0 framework on the GPU. The measurements are taken as the median execution time for ten consecutive runs. Hence, due to the presence of compiler caches in both LambdaJIT and CUDA, the JIT compilation time is not taken into account since it is only executed once.

The benchmark suite used for evaluation is composed of:

- *Saxpy*: a simple combination of scalar multiplication and vector addition expressed as a binary transform, as shown in Figure 1. It is a common benchmark, used in standard Basic Linear Algebra Subprograms (BLAS).
- *BlackScholes*: an option pricing application modeling a Partial Differential Equation. This benchmark exists in many variants on CPU [2] and GPU [5]. It uses built-in mathematical functions to compute logarithms and square roots.
- *MonteCarlo*: application using the Monte Carlo method to estimate the value of π. For each element in the range, the functor object generates a given number of random coordinates inside a square and increment a counter each time the generated point fits inside the inscribed circle.
- *Factorization*: An array is initialized with random unsigned values. For each element of the array, the value is decomposed in a product of prime numbers. Each prime number is stored in an output array for each value.

5.2 LambdaJIT Evaluation

Figure 6 shows the performance of each benchmark for all the execution policies and a range of input sizes. Performance is shown as a speedup over a sequential baseline compiled with Clang.

While the code is suitable for GPU execution on each benchmark, the automatic target selection of the LambdaJIT runtime is

overridden by user parameters, allowing each policy to be executed. The runtime explores three policies:

- *λopt*: the algorithm is executed sequentially, but the Closure objects are specialized, as described in Section 4, and JIT compiled before execution.
- *λpar*: the Closure instances are also optimized, and the algorithms is automatically parallelized.
- *λgpu*: the computation is offloaded to a gpu. The performance reported includes memory allocation and communication costs to and from the device.

In order to compare the performance of the *λgpu* execution policy, we also compare against hand-written applications implemented in CUDA.

Figure 6(a) shows the performance for the SAXPY application. Because SAXPY performs very little computation, it exposes the overheads introduced by the LambdaJIT runtime. In this case, the baseline outperforms all the dynamic policies. This is because the runtime has to hash the value of the captured variables (which is the factor value *A* in this case), in order to find the specialized version in the compiler cache. This alone outweights the benefit of the optimization. This graph also shows the sensitivity of the partial specialization optimization: the application uses the size of the range as the factor value, which the optimizer can transform into bit shift operation for small powers of 2. This causes the increasing relative performance of the *λopt* policy until the input size 2^{15}, almost amortizing the lookup overhead. For larger values, this optimization cannot be applied and the optimizer falls back to a multiplication, causing a sharp loss of performance. The extra overhead introduced by the parallel version of the algorithm to start new execution threads causes the parallel policy to be slower than the sequential policy. Note that the GPU target from LambdaJIT is on par with the CUDA implementation and both have very low performance. Both GPU executions suffer from overhead caused by data transfer between CPU and GPU.

Figure 6(b) is the measurement for BlackSchole, a more compute intensive benchmark. This time, all policies outperform the sequential baseline. The *λopt* policy takes advantage of the optimizer to slightly outperform the baseline. The computation is intensive enough to take advantage of the parallel and the GPU policies, which achieve an average speedup of 4.7x and 5.0x respectively across the input sizes. In this case, there is no clear best strategy between the two for medium size inputs. However, this is highly dependent on the hardware: a different CPU or a different GPU would tilt the balance. This demonstrates the importance of code portability.

The MonteCarlo application, shown in Figure 6(c), exposes a typical tradeoff between computation and communication in GPGPU programming. Although the raw compute power of GPUs is by far exceeding the peak performance of CPUs, the communication overhead caused by the lack of shared memory can outweigh the compute time. This results in the three phases shown in the graph. For small input sizes, the parallel policy outperforms the GPU. For very small input size, the GPU is even slower than the sequential baseline. However for larger input size, the balance shifts gradually towards a compute bound behavior rather than communication bound. It quickly overtakes the sequential and the parallel performances. When the GPU reaches full occupancy, the performance scales linearly. This is also the case for the CPU, resulting in the plateau seen for larger input size: the relative performances scale the same way. The JITed version of the closure executing on the GPU outperforms the hand optimized CUDA version by up to 9%. This outlines the benefits of the JIT specialization, which propagates some constants used to generate the random numbers, which are not known at compile time but are captured by the lambda function in the C++ input.

Finally, the Factorization application shown in Figure 6(d) demonstrates another well known issue of GPGPU programs: their inability to cope with highly divergent code. Factorizing a number depends on the number of factors it has and the value of its highest prime. This differs from one array element to another, but since the GPU threads sharing a SIMD unit execute in lockstep, the vector unit has to finish the longest factorization in the data vector before computing another input. As a result, the GPU performance is impaired by the non-uniformity of the computation and is considerably worse than the CPU performance for small input sizes. For larger input, the GPU overtakes the baseline but immediately settles with a small speedup since it reaches maximum thread occupancy even though it is still heavily under-utilized. The parallel policy on the other hand can cope well with irregular computation and achieves over 5.15x speedup over the baseline.

These benchmarks cover a wide range of behaviors on the CPU and the GPU. For this benchmark suite, there isn't a single policy dominating the others. This outlines the need for easily retargetable code. The peak performance depends on the hardware and the applications, or even their input set. The ability to dynamically re-target the code without any programming effort allows LambdaJIT to efficiently explore and exploit the performance potential of heterogeneous systems.

6. Prior Work

Heterogeneity has been the focus of a lot of research since the birth of multi-core processors, and more recently the emergence of compute accelerators. The common motivation is to overcome dramatically increased programming complexity. Nowadays, optimized programs still only achieve a fraction of the peak performance offered by these heterogeneous systems and ongoing projects aim to improve this.

6.1 Multi-Core Frameworks

Many frameworks provide API or compiler support to exploit the performance of multicore systems.

Some of them are also based on or extend the STL. Multi-Core STL [19] implements parallel version of common STL algorithms. STAPL[4] provides a parallel superset of the STL.

Intel provides Threading Building Blocks (TBB)[17] is a template library containing a collection of components for parallel programming. It achieves good performance on multicore systems using a complex task management scheduler.

While these frameworks require the user to use a special API or restrict the usage of the STL algorithms, LambdaJIT is designed to handle any source code and will analyze it to check whether algorithms can be parallelized.

6.2 GPU Programming and Optimizations

Because GPU programs are notoriously difficult to implement and optimize, many domain specific languages and frameworks have been developed to abstract their architecture. Cg [11] was one of the first languages designed for Graphics Hardware, followed by Brook [3], which introduced the streaming programming concept for General Purpose computing. Presently the two main languages used for GPGPU programming are CUDA[13] and OpenCL[9].

The domain specific languages expose properties of the programming model, but don't abstract the complexity of it. Hence, a lot of projects have been developed on top of these languages to provide high level abstraction. Thrust[8] provides a collection of template algorithms and containers, similar to the STL. SkelCL[20] provides a similar approach for OpenCL.

Significant efforts have been put in compiler research to translate existing codebase or automatically generate optimized source code in a GPU specific language. Some compiler approaches are non intrusive and don't require a re-write of the code but need annotations from the user. Lee et al. [10] and Grewe et al.[7] investigated automatic translation from OpenMP to CUDA and OpenMP to OpenCL respectively. OpenACC[14] defines a set of compiler directives like OpenMP to generate GPU code automatically.

Other approaches provide both a domain specific language and a compiler generating GPU compatible code. Liquid Metal[6] provides a Java API and a runtime library generating OpenCL code. Halide [16] provides language extension to C++ and a compiler supporting many backends.

In contrast to existing GPU compilers and frameworks, our approach does not require specific annotation from the source code or a specialized API. Combining the high level information of the STL algorithms with the low level static analysis of the functors provide enough information to select the GPU and generate optimized code automatically whenever possible.

6.3 Dynamic Compiler Optimizations

Specializing code at runtime is a well known optimization. Java for example has an integrated JIT compiler as part of the virtual machine, which has been the target of a lot of research [1][15].

These approaches require a very complex runtime: the intermediate representation of the code needs to be instrumented and profiled in order to find optimization candidates, and complex error recovery mechanisms have to be in place in case the optimizer introduces errors. While in our framework, the offline compiler already isolates snippets of codes which are safe to optimize by construction and yields good optimizations.

Project Lancet [18] aims to reduce the complexity of JIT compilation in Scala by exposing it to the user. Programmers can explicitly provide strong isolation of functions and data by "freezing" them. The frozen variables are used by the JIT to guide symbolic execution and optimization in the same way the LambdaJIT framework uses captured variables in C++ lambdas.

Finally, the most closely related work is Intel's Array Building Blocks (ArBB) [12]. It provides a dynamic compiler which generates Intermediate Representation from C++ code. Their runtime then executes this IR in a virtual machine, using a JIT compiler to optimize, automatically parallelize and re-target the code dynamically. Our approach differs in that we can support any C++ program without restriction whereas ArBB requires the user to express algorithms using a special API. Furthermore, data has to be explicitly transfered between C data types and ArBB parallel types. Whereas ArBB is limited to multi and many cores, LambdaJIT is also able to assess GPU compatibility of the IR and distribute computation.

7. Conclusion and Future Work

We showed that it is possible to automatically parallelize C++11 code using semantic information provided by standard constructs and simple compiler analysis. Furthermore, we prove that by integrating intermediate representation from the compiler into the executable, a runtime can efficiently optimize the code and even recompile it to run on a new target. This is both easily extendable, by providing more backends, and safe, since a precompiled version provides a safe fallback.

We could further improve this work by implementing very lightweight backends to reduce code generation time.

This work uses lambdas as a proof of concept, as they fit naturally with the standard algorithms, which provide high level semantic constructs. However, the same techniques could be applied at a function level, or even at a basic-block level, depending on how much of the intermediate representation can be injected into the binary.

Finally, these techniques are not limited only to C++11, but could be implemented for any compiled language. This would enable languages like Java to also benefit from automatic parallelization and dynamic re-targetting.

References

[1] A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast, effective code generation in a just-in-time java compiler. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 280–290, New York, NY, USA, 1998. ACM. ISBN 0-89791-987-4. . URL `http://doi.acm.org/10.1145/277650.277740`.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-282-5. . URL `http://doi.acm.org/10.1145/1454115.1454128`.

[3] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: Stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 777–786, New York, NY, USA, 2004. ACM. . URL `http://doi.acm.org/10.1145/1186562.1015800`.

[4] A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger. Stapl: Standard template adaptive parallel library. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, SYSTOR '10, pages 14:1–14:10, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-908-4. . URL `http://doi.acm.org/10.1145/1815695.1815713`.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-5156-2. . URL `http://dx.doi.org/10.1109/IISWC.2009.5306797`.

[6] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink. Compiling a high-level language for gpus: (via language support for architectures and compilers). In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 1–12, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9. . URL `http://doi.acm.org/10.1145/2254064.2254066`.

[7] D. Grewe, Z. Wang, and M. O'Boyle. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–10, Feb 2013. .

[8] J. Hoberock and N. Bell. Thrust: A parallel template library, 2010. URL `http://thrust.github.io/`. Version 1.7.0.

[9] *The OpenCL Specification version 1.2*. Khronos OpenCL Working Group, 19 edition, Nov. 2012. `http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`.

[10] S. Lee, S.-J. 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, pages 101–110, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-397-6. . URL `http://doi.acm.org/10.1145/1504176.1504194`.

[11] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: A system for programming graphics hardware in a c-like language. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 896–907, New York, NY, USA, 2003. ACM. ISBN 1-58113-709-5. . URL `http://doi.acm.org/10.1145/1201775.882362`.

[12] C. J. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. D. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and

D. Zhang. Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 224–235, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-61284-356-8. URL http://dl.acm.org/citation.cfm?id=2190025.2190069.

[13] NVIDIA Corporation. Cuda specifications. URL http://docs.nvidia.com/cuda/.

[14] OpenACC Working Group. The OpenACC Application Programming Interface, Version 1.0. November 2011. URL http://www.openacc-standard.org/.

[15] M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. C and tcc: A language and compiler for dynamic code generation. *ACM Trans. Program. Lang. Syst.*, 21(2):324–369, Mar. 1999. ISSN 0164-0925. . URL http://doi.acm.org/10.1145/316686.316697.

[16] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. . URL http://doi.acm.org/10.1145/2491956.2462176.

[17] J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007. ISBN 9780596514808.

[18] T. Rompf, A. K. Sujeeth, K. J. Brown, H. Lee, H. Chafi, and K. Olukotun. Surgical precision jit compilers. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 41–52, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2784-8. . URL http://doi.acm.org/10.1145/2594291.2594316.

[19] J. Singler, P. Sanders, and F. Putze. Mcstl: The multi-core standard template library. In *Proceedings of the 13th International Euro-Par Conference on Parallel Processing*, Euro-Par'07, pages 682–694, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-74465-7, 978-3-540-74465-8. URL http://dl.acm.org/citation.cfm?id=2391541.2391622.

[20] M. Steuwer, P. Kegel, and S. Gorlatch. Skelcl - a portable skeleton library for high-level gpu programming. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1176–1182, May 2011. .