And Now for Something Completely Different: Running Lisp on GPUs

Tim Süß*, Nils Döring†, André Brinkmann†
Department of Computer Science*
Zentrum für Datenverarbeitung†
Johannes Gutenberg University Mainz
Mainz, Germany
{suesst, doeringn, brinkman}@uni-mainz.de

Lars Nagel
Department of Computer Science
Loughborough University
Loughborough, UK
L.Nagel@lboro.ac.uk

***Abstract*—The internal parallelism of compute resources increases permanently, and graphics processing units (GPUs) and other accelerators have been gaining importance in many domains. Researchers from life science, bioinformatics or artificial intelligence, for example, use GPUs to accelerate their computations. However, languages typically used in some of these disciplines often do not benefit from the technical developments because they cannot be executed natively on GPUs. Instead existing programs must be rewritten in other, less dynamic programming languages. On the other hand, the gap in programming features between accelerators and common CPUs shrinks permanently. Since accelerators are becoming more competitive with regard to general computations, they will not be mere special-purpose processors in the future. It is a valid assumption that future GPU generations can be used in a similar or even the same way as CPUs and that compilers or interpreters will be needed for a wider range of computer languages.**

We present CuLi, an interactive Lisp interpreter, that performs all computations on a CUDA-capable GPU. The host system is needed only for the input and the output. At the moment, Lisp programs running on CPUs outperform Lisp programs on GPUs, but we present trends indicating that this might change in the future. Our study gives an outlook on the possibility of running Lisp programs or other dynamic programming languages on next-generation accelerators.

I. Introduction

In recent years, graphics processing units (GPUs) have been gaining importance as the development of CPUs is reaching physical limits. The performance of processors cannot be much improved by increasing the processor frequency, and vendors rather raise the number of cores or boost the performance by integrated vector units such as SSE or AVX. GPUs, on the other hand, achieve a significant performance speedup due to their massive internal parallelism.

In the beginning, GPUs were only used for rendering images displayed on the screen. Over time features were added that allow users to push geometry descriptions into these devices for which the data was transformed into three-dimensional scenes and finally into pictures. These rendering pipelines were extended by so-called *shaders* which gave programmers more flexibility in manipulating data (usually geometry or textures). However, programmers quickly used these shaders not only for rendering images, but also as a new programming interface to solve computational problems with a high degree of parallelism. This *General Purpose Computation on Graphics Processing*, or short: *GPGPU computing*, required the difficult translation of general problems into languages understood by graphics adapters. Solutions were sought that allow programs for GPUs to be written in the same way as programs for CPUs. In 2006 *Nvidia* introduced *CUDA*, a parallel computing platform and library which works with C, C++ and Fortran [29]. CUDA started with severe limitations like the absence of recursions, but over the years many of them were removed. In the future developing programs for GPUs will probably not differ much from programming CPUs.

CUDA and other libraries like OpenCL enable developers in many domains (such as machine learning or artificial intelligence) to benefit from the massive parallelism in GPUs [5], [16], [19], [15], [30].

In this paper we present *CuLi*, an CUDA implementation of the functional programming language *Lisp*. Functional languages avoid side-effects [12] and are therefore ideal for exploiting parallel architectures. There are many functional languages, dialects, and extensions designed for the parallel execution of tasks [34], [2], [28], [1], [9], [11], but all these languages use the CPU for their computations by generating multiple threads or processes. None of them makes use of GPUs (even when they suggest that they are designed for heterogeneous platforms). Our implementation *CuLi*, on the other hand, is a complete Lisp interpreter running on the GPU. CuLi performs all computations on the GPU (device side). Only the *read* and the *print* part of the *read-eval-print loop* (REPL) are executed on the CPU (host side) which cannot be avoided because the host side is responsible for the user interface.

The main reasons for choosing Lisp are that it is a functional language and that Lisp's language core is small but extremely powerful. Lisp statements entered are submitted to our Lisp interpreter on the GPU where the statements are *evaluated*. The result is then copied to the CPU where they are displayed. The interpreter was written in C using the CUDA library. Aside from this, no further libraries were used. Our system supports familiar Lisp features like function and variable definitions, arithmetic operations and macros. The interpreter allows an interactive usage which requires that the successively created

environment on the GPU is persistent until the interpreter is terminated.

The main contribution of this work is to show that it is possible to run a complete and highly dynamical programming language on the GPU. To the best of our knowledge, this is the first interactive language on the GPU using the host side only for input and output. This allows the usage of systems which are equipped with a rather slow CPU, but a powerful GPU (like the *Nvidia Tegra* platform) to perform massively parallel tasks. It should be mentioned, however, that current GPUs do not provide all the features required to outperform CPU implementations of Lisp (e.g., hierarchical data structures with good performance). For this reason, this is foremost meant as a foundational work and feasibility study. We expect that GPUs will be programmed in the same way as CPUs in the future and that they will become the main compute resource due to their high computational power.

II. Related Work

Since NVIDIA announced CUDA in 2006, a lot of programs have been ported to GPUs [29] and usually allowed researchers and companies to compute faster. This huge success has led to many adaptations of CUDA and the development of alternatives. Some of them are extensions to already popular programming languages, others are native languages that use GPUs to compute data in parallel. Moreover, there is hardware-specific research on topics related to GPUs and on how to efficiently adapt programs to GPUs.

A. Language Extensions

CUDA [29] is an extension of the C language and relatively easy to learn for C programmers. CUDA provides an API offering device functions for administrating the GPU, transferring data between host and device and synchronizing device with host. To generate machine code for the GPU, NVIDIA provides a compiler tool set that takes C code and builds kernels and functions for the device. Using C and CUDA, it is possible to achieve a very fine-grained access and use of the GPU.

OpenCL [10] (Open Compute Language) is a standardized programming interface to implement software for heterogeneous systems. It provides a subset of the C++-14 standard and can be compiled to various platforms to provide parallelized code. OpenCL is a competing standard to NVIDIA's CUDA. While CUDA is limited to NVIDIA GPUs, OpenCL can be used to generate code for NVIDIA and AMD GPUs, as well as for mainframe CPUs and desktop CPUs and even for FPGAs (Field Programmable Gate Arrays). OpenCL is therefore more versatile than NVIDIA CUDA which is virtually the standard for GPU programming.

PyCUDA [18] extends the Python programming language so that Python programs can use CUDA-enabled GPUs. It exposes the CUDA API to the programmer who can develop kernels in CUDA code to improve the performance of the computation. The PyCUDA framework takes these code blocks, gathers information about the hardware currently installed in the computer and optimizes the kernel it generates for this hardware. It provides a cache to store these kernels so that they can be executed at any time without recompiling them.

Petersen et al. introduce a method for Haskell that exploits the potential of SIMD compute units [26], and Mainland et al. show how Haskell SIMD operations can be used in streams [21]. However, these approaches do not use the device with the highest degree of parallelism, the GPU.

There are others who let the GPU perform computations in functional languages [23], [3]. Some projects even bring interpreted languages to the GPU. *JavaScript on the GPU* is a project that ports the language *JavaScript* to GPUs [24], and the *ParallelJS* framework allows the execution of JavaScript code on CUDA-capable accelerators [35]. Both implementations have in common that they perform the computation on the GPU, but interpret the input on the CPU. Prior to this work, there has been no project or framework that enables users to run Lisp code directly on accelerators. This may have historical reasons because the first CUDA version did not support recursion, a requirement for Lisp, and many developers are not aware that this has changed with CUDA version 2.0. There are different Lisp wrappers projects that allow for executing CUDA functions in Lisp programs or that can translate Lisp codes to CUDA codes [33], [25].

Another approach to using the GPU is taken by *Loo.py* [17]. In contrast to exposing the CUDA API to the programmer, Loo.py requires the programmer to define code areas that are to be run in parallel and automatically builds the code for the GPU. Loo.py uses the polyhedral model and annotations by the programmer to optimize the code and parallelize it on the GPU. For this it provides a markup language to define areas for parallelization and data dependencies. With this information and the information about the hardware, the extension can build optimized kernels.

A similar approach is taken by compiler-based language extensions like *OpenACC* [8] and *OpenMP* [6]. Both provide compiler extensions, so-called pragmas, that declare parts of the code to be built for parallel execution. These pragmas are defined by the standards of OpenMP and OpenACC and implemented by different compilers (leading to diverging and competing standards).

With these pragma constructs it is possible to declare parts of the code to run in parallel and have the compiler build kernels for the devices chosen at compilation time. The code itself rarely has to be changed. This helps to port code from one device to another and to get sequential code to run in parallel. Although the performance is reasonable for a fast transition, hand-tuned or hand-written code usually performs much better.

B. Native programming languages

The programming language *Rust* [14] was introduced to replace C++ as a fast and secure programming language and is being developed to be a system programming language focussing on the integration of heterogeneous systems into one language. Due to the use of the *LLVM* (Low-level Virtual

Machine) project for building and compiling code, it can generate kernels for different devices, including GPUs, and therefore presents a native way to generate code for the GPU. In contrast to language extensions like CUDA or OpenCL, Rust provides a consistent programming environment.

For simplifying GPU programming, the programming language *Chestnut* [31] was designed. In contrast to Rust, this language focuses mainly on the GPU-side of the computation. While Rust has a very low-level view of the GPU and requires the programmer to think about data layout and location, Chestnut provides a sequential view of the program and builds the parallel kernel from that. The Chestnut compiler builds an AST (Abstract Syntax Tree) from the user's program and compiles it to C++ and CUDA code that will finally be compiled to machine code.

Relatively similar to CuLi, *Domain Specific Languages* (DSLs) like *Ebb* [4], [27], [13], [7] that work on both, CPU and GPU, provide the programmer with a domain-specific API for simplified access to the GPU. In contrast to CuLi they only offer a limited set of functions and are therefore bound to their domain.

All these programming languages are examples for more or less hiding the differences of CPU and GPU code which makes programming easier and reduces the effort of writing code for GPUs.

C. Hardware-specific adaptations

There are many examples of languages that provide built-in methods for exploiting the parallel nature of modern multi-core processors [34], [2], [28], [1], [9], [11]. Next to this obvious parallelism, CPUs provide vector units (e. g. SSE or AVX) that extend the processor's internal parallelism [20].

Since the computation model of GPUs is very different to the one of CPUs, certain parallel programming tasks are more difficult. The foremost problem is the lack of support for inter-block synchronization and communication. This is due to the hardware and low-level design of GPUs. Hence, one of the things needed in multi-threaded environments is proper synchronization which helps to exchange data and signals between threads and to prevent race conditions. CUDA provides basic functions to synchronize threads within blocks but not throughout multiple blocks. Synchronization based on an atomic spin-lock [36], [32] provides a possible way to have all threads, regardless in which block they reside, to wait for one signal and, while waiting, stop other computation.

This mechanism can also be used to exchange messages between arbitrary threads. They access global memory [36] while waiting for a signal, telling them that new data has arrived.

Both of these use cases suffer from rather poor performance of the spin-locks used. Although NVIDIA has improved the performance of atomic access to memory, it is rather energy-consuming and inefficient to have all threads actively waiting for the same memory area to change and while doing so, have all processors of the GPU waiting busily for this to happen.

III. Implementation

The Lisp programming language has a long history. John McCarthy introduced this language in 1960 [22]. Since then Lisp has become an important language in artificial intelligence (AI). However, until now there was no way for using Lisp directly and interactively on GPUs, although AI is one of the area where those devices are widely used.

In this section we describe details of our Lisp implementation *CuLi*. This implementation provides all features of Lisp, even built-in functions for parallel computations.

A. CuLi

CuLi is implemented in pure ANSI C and the NVIDIA CUDA C extension. Since CUDA lacks a string library, we implemented our own with functions to parse strings. These functions are also used in the CPU tests for comparison reasons.

To implement dynamic multi-threading, CuLi uses the threads provided by CUDA for the GPUs (for the CPU version we use pthreads). Beside the CUDA and the standard C libraries no others are used.

CuLi is a dialect of the LISP programming languages. It uses a fully parenthesized prefix notation where the first symbol determines the function to apply. For example `(* 2 (+ 4 3) 6)` first sums up the values 4 and 3 and multiplies the result with 2 and 6.

a) Nodes: The most basic structure of CuLi is the node, implemented as a C `struct` (see Figure 1). Such a node stores values, functions and links to other nodes. After a value has been assigned to a node, it becomes immutable. This is necessary for parallel execution, because it prohibits side-effects.

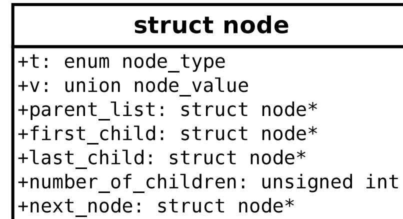


Fig. 1: Structure of a basic node.

b) Node types: Each node contains a type describing its content. It can either have one of the primitive types `N_NIL`, `N_TRUE`, `N_INT`, `N_FLOAT`, `N_STRING`, `N_SYMBOL` and `N_FUNCTION` or one of the complex types `N_LIST`, `N_EXPRESSION`.

`N_NIL` nodes contain the value `nil` that determines a `false` value. In Lisp empty lists and false conditions evaluate to *nil*. Since `N_NIL` nodes already define their value there is no need for checking the value at run time. The opposite of `N_NIL` is `N_TRUE`. Non-empty lists and fulfilled conditions evaluate to true which is equivalent to `N_TRUE`.

Numeric values are indicated by the types `N_INT` or `N_FLOAT`.

Strings are represented by the type N_STRING. They are stored in a constant pointer (const char *) within a node.

Symbols, denoted by the type N_SYMBOL, are basically the same as strings. These two types are differentiated during the evaluation of an input. Symbols are replaced (even recursively) while they are processed.

Finally the last primitive type is N_FUNCTION which applies to built-in functions that are stored in the global environment (like +, -, defun and cdr). Environments are similar to namespaces in other languages. Other user-defined functions, that are defined by the defun built-in function, are stored as expressions. Functions are stored as function pointers and they expect a list of nodes containing the parameters and a pointer to the environment that should be used for its execution. This provides the Lisp-typical feature that functions can behave differently to the same parameters in different environments.

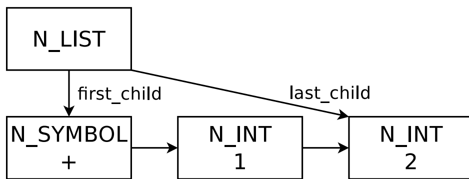


Fig. 2: An example of a list, with a symbol and two integers.

An N_LIST contains the starting and end point of a linked list of nodes. For example (+ 1 2) will be parsed into a list that has a symbol node for + as its first child (see Figure 2). That node points to an integer node with the value of 1 and that points to another integer node with a value of 2. The list finally points to that last node, to mark the end of the list. Since lists are accessed in Lisp with variations of the functions car and cdr linked lists are the natural data structure to use.

If a list starts with a symbol as the first child, this list will be evaluated as an expression, as shown in 3. The symbol will be searched in the environment of the evaluation and replaced with the function or value associated with the symbol. Since the evaluation of a symbol is dependent on the position in the syntax tree, it will be evaluated with different environments. Therefore the same symbol can be evaluated differently in different positions.

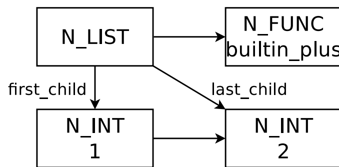


Fig. 3: Example of an expression with a built-in function and two integers. This is the intermediate step within evaluation of Figure 2

Another type of lists are forms (N_FORM) which are user-defined functions that are stored in the global environment by the keyword defun. In contrast to expressions, forms additionally store a list of parameters. These parameters are symbols used to store arguments in the local environment during evaluation. If a form is evaluated, it adds the given arguments to the local environment and evaluates the stored subtree with this environment. This ensures the existence of the required parameters in the subtree.

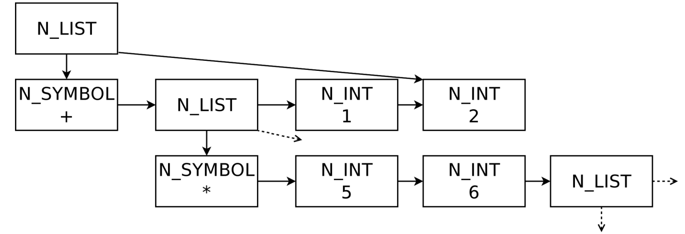


Fig. 4: A tree built from lists. This tree was built by the input (+ (* 5 6 (...)) 1 2). The last_child-pointers are not shown for clarity.

Every correct CuLi input consists of at least one list. This list can contain arbitrary elements and therefore lists that itself may contain lists. This builds the tree that will be executed. Figure 4 shows a part of a tree of lists that are built by CuLi.

c) Memory: Nodes are stored in a large array that is created at the beginning of the program. This array has a fixed length set during the compilation of CuLi. The length limits the number of nodes that can be used during a run of CuLi. Whenever a function asks for a new node to store a value, the sequentially next free node of this array will be returned. When the nodes are not needed anymore, they are marked as free.

B. *Execution flow*

To understand how expressions are processed, it is necessary to understand how variables, symbols and environments are handled. This is also crucial for the understanding of parallel execution. The basic element for expression processing is the environment structure.

The complete processing of an input is done in the following steps illustrated in Figure 5: (a) The input string is passed to the parser. (b) The parser generates a tree of nodes, the parse tree. (c) The evaluation traverses the parse tree, evaluates it and generates the result tree. (d) The printer traverses the result tree and generates the output string.

The subsequent paragraphs explain these steps in more detail.

a) Environment: Expressions are processed in trees like the one described in Figure 4. An environment in CuLi is built like a tree. The structure of the environment is displayed in Figure 6 and 7.

An environment contains a linked list of *environment nodes* and a link to a parent environment. The only exception is the global environment that has no link to other environments. Each *environment node* itself contains a symbol for comparison and the node that the symbol points to. Since each list and

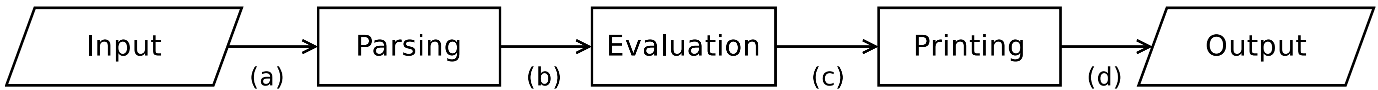


Fig. 5: Execution flow.

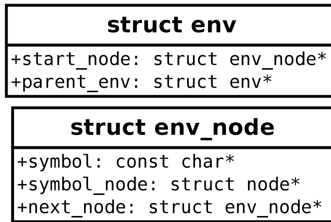


Fig. 6: Structure of the environment and the elements of it.

each expression has an environment that, through its parent environment, points to the global environment, values that are stored in the global environment are accessible through each environment in the tree.

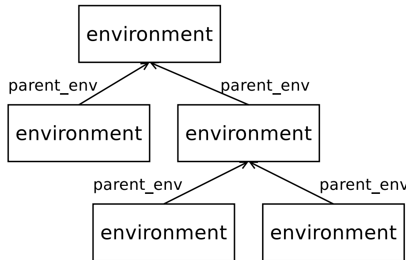


Fig. 7: Tree of multiple environments linking to each other.

If an expression defines a symbol, the symbol is normally stored in the local environment of this expression. It is only accessible for this expression and the expressions nested within (i.e. which are defined later). Previous symbol definitions are unaffected. Using this mechanism it is possible to create locally scoped variables. It is also possible to have locally different values for the same symbol. Local variables are defined by the built-in function `let`. This function adds a new symbol and the corresponding value to the environment of the current expression. In contrast, the built-in function `setq` updates the nearest existing symbol that matches. Hence, it can change a local variable as well as a global one, if there is no other variable with the same symbol in the current subtree. Thus, this function might cause side-effects and must be used carefully in parallel computations.

b) Parsing: The parser builds the parse tree, a tree of nodes describing the input string. For this it reads the string character by character. An opening parenthesis builds a new list and generates a new environment for this new list. This new list will be the current list until the parser reaches a matching closing parenthesis. All nodes generated within these two are added to the new list. When the new list is finished by the closing parenthesis, its parent list becomes the current list. This builds the parse tree. A possible parse tree is displayed in Figure 4.

The parser walks the string until it sees a whitespace character, or an opening or closing parenthesis. These characters are markers for the parser. The substring between the last marker and the current marker is the input to generate a new node.

If the substring starts with quotation marks, the new node will be an `N_STRING` node with the substring as value. The quotation marks are not carried into the value. `N_NIL` and `N_TRUE` nodes are built from the substring `nil` and `T`, respectively. If the substring starts with a digit or a character indicating a number (`+-.E`), the new node will either be an `N_FLOAT` node or an `N_INT` node. The `N_FLOAT` node will be generated if a dot (`.`) is found in the substring. If none of the above applies the new node will be of type `N_SYMBOL`. The substring will be stored as value in the new node.

c) Evaluation: The parse tree is traversed recursively by the evaluation stage of the execution flow.

If the current node is an `N_LIST`, the first element will be evaluated in order to decide whether the list is an expression or a form. If it is either of them, the list will be evaluated as an expression or form, respectively. Otherwise, all other elements are evaluated, and the resulting list will be returned.

If the current node type is an `N_SYMBOL`, the evaluation tries to match the symbol within the current environment. The first occurrence in the environment tree will be used to exchange the symbol. This facilitates a late binding of symbols. The matched symbol decides whether to handle the list as an expression or a form.

If there is no matching symbol, the symbol is not replaced.

If the current node is an expression, its children will be passed to the function pointer stored in the expression node. They are not evaluated first since built-in functions might use them without evaluation (e. g. the `setq` function).

If the current node is a form, a new environment is created to store the parameter values for the user-defined function. The arguments are evaluated to be used as parameter. Afterwards the user-defined form will be evaluated within the new environment.

If the node type is none of the previously mentioned ones it must be a primitive and can be returned unchanged.

d) Printing: During the evaluation phase a node tree is generated that only consists of primitives. The tree's nodes are passed in postfix order to the printer that generates the output string. For each node it appends the corresponding string representation to the output string. Although CUDA kernels (NVidia GPUs with compute capability `>= 2.0`) are able to print from the device code directly to the standard output of the host, the output generated will only be transferred to

the host by calling `cudaDeviceSynchronize` or blocking calls to `cudaMemcpy`. Since CuLi does not use them within evaluation or for communication, it sends the output string back to the host the same way it uses for the input and have the host print it directly.

C. GPU implementation

CuLi runs the complete interpreter code on the GPU. This includes all steps displayed in Figure 5, except input and output. While the evaluation is performed by the GPU's kernel, the host is responsible for input and output.

a) Host code: Input and output are both located in the host code. The host code is running in a loop and provides a command-line prompt which fetches, sanitizes and uploads the input to the GPU. The host uploads the input to the GPU if the number of opening and closing parentheses is equal. The GPU signals the host when the computation has finished, and the host prints the output on the screen and waits for another user input.

b) Host-GPU Communication: To be able to send the input from the host to the GPU, both share a common C struct (see Figure 8). The elements of this structure are generated by the `cudaHostAlloc` API call. This call allocates the variables on both, the host and the GPU and links them. Using the parameter `cudaHostAllocMapped` ensures an automatic update on both sides when values alter. With this parameter set, neither the host nor the GPU has to call `cudaMemcpy` to initiate the copying of data.

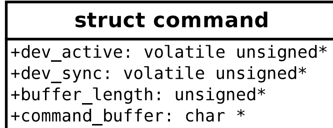


Fig. 8: The command buffer used to exchange input and output between the CPU and the GPU and to synchronize the host and the device.

The values of the `struct` members `dev_active` and `dev_sync` are used for sending signals between host and device. The `dev_active` signal is used to terminate the GPU kernel. When the `dev_active` flag is set to `0` by the host code, the kernel on the GPU ends itself.

The variable `dev_sync` is used at both sides, the host and the device. If this value is set to `0`, the GPU must wait for an input from the host. When a new input the string will be copied to `command_buffer` and `buffer_length` is set to the input length. The host then sets the `dev_sync` to `1` and waits until the GPU sets it back to `0`. In the meantime the GPU evaluates the input and copies the result and its length into the command structure. It then sets `dev_sync = 0` and waits for the next input (see Figure 9). The host reacts to the synchronization flag and prints the output back to the user.

c) Device code: CuLi uses a CUDA kernel with a one-dimensional grid of thread blocks, each addressable by a single integer. The grid has at least as many blocks as it needs to saturate all streaming multiprocessors (SMs) of the device used. Since each block has 32 threads (exactly the size of a warp), the grid size is a multiple of 32. Each thread has an address within the grid, determined by the block address and its position within this block.

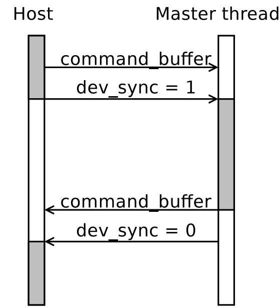


Fig. 9: Communication between the host code and the device code. Grey areas denotes active periods and white areas waiting periods.

The GPU kernel is directly started after the host code. The master thread of the kernel (thread and block id `0`) sets up the global environment used by all worker threads. Afterwards it waits for a new input.

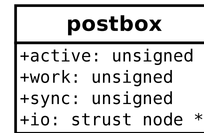


Fig. 10: The postbox used for the communication between the master thread and the worker threads.

Each thread has its own, exclusive postbox which is stored in an array in global memory (see Figure 10). Initially the variables in all postboxes are set to `active=1`, `work=0`, and `synchronization=0`. The master thread sets the `active` flag of all threads to `0` when it terminates. This causes the worker threads to break their loop and end themselves.

To ensure that all values are written correctly, atomic memory functions are used to access, read and modify the values of the postboxes. This also prevents CUDA's transparent caching of variables and ensures that values are only read directly from global memory. This implies a performance penalty [32] since atomic memory accesses are slower than direct ones.

D. Multi-threaded execution

In a CuLi program a parallel section is indicated by the newly defined `|||`-expression. Such an expression is structured as follows: the first parameter after `|||` is an integer that defines the number of threads, the second parameter is the function to be executed in parallel, and the remaining parameters are the arguments of that function. When the evaluation step of REPL encounters this expression, it distributes the work among multiple workers and waits for their results.

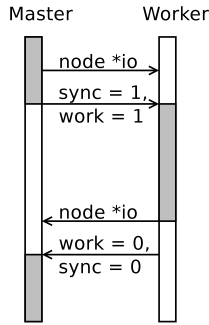


Fig. 11: Communication between master thread and worker thread. Grey denotes active periods, white waiting periods.

a) Work distribution: When the master thread encounters the `|||`-function it distributes the given function and variables to the given number of workers. A typical call could look like the following: `(||| 3 + (1 2 3) (4 5 6))`. The master thread will distribute the work among three workers. First it creates a new expression for each worker thread, which links to the function for `+`. Next, it takes the n-th element of both lists as function arguments, where n is the worker id, and adds them to the previously generated expression as children. In our example, the first worker's expression is `(+ 1 4)`, the second one's is `(+ 2 5)`, and the third one's is `(+ 3 6)`. These expressions are stored in the `io` variable of the workers' postboxes. Next the `work` and `sync` flags are set to `1` (see Figure 11). Finally, after the work distribution, the master thread waits for all workers to signalize the completion of their tasks.

b) Parallel execution: Each worker thread evaluates only its own subtree, provided in its postbox. The root of this subtree is linked to the environment of the `|||`-expression which itself has a path to the global environment. Therefore each worker has access to the environments preceding its own but not to the environments of other workers. This ensures a local scope for all workers. Values stored in a worker's environment do not affect other workers.

The evaluation is done as described before. The result of the evaluation replaces the subtree in the `io` variable of the workers postbox, once the worker has finished its evaluation.

The master thread waits until each worker has set its `sync` flag to `0`. Then it generates a new `N_LIST` node and appends the workers' results in the same order as the work was distributed. The resulting list is handed over to the expression that has called the `|||`-expression or it is printed if it is the root expression.

c) Synchronization: Once the `sync` flag is set, the worker is able to start working. If CuLi used only one thread per block, this would be sufficient to synchronize the master and the workers. If more than one thread is used per block, this is not sufficient anymore. CUDA always bundles 32 threads to a warp. These threads execute the same code. As a consequence this can lead to situations where some threads of a warp are stuck in busy-waiting loops indefinitely while the remaining threads are waiting for them to finish. In brief, this prohibits to have less than 32 workers or a number of workers unequal to a multiple of 32.

To cope with this burden, CuLi uses an additional synchronization flag for each block that signals a complete block if there are changes to workers in it (see Alg. 1 line 6). When this flag is set all threads in a block end their busy-waiting and check if there is work for them. Threads with work evaluate their subtrees while the others wait until they finished. Afterwards all threads re-enter the busy-waiting loop. The CUDA function `__threadfence_block` ensures that all threads of a block enter the busy-waiting loop simultaneously (see Alg. 1 line 5).

d) Warp divergence: Warp divergence is a concept in CUDA programming that has no equivalent in CPUs. In this work the GPUs do not execute single threads but warps of 32 threads (to fill the warp completely) and all threads execute the same instructions in parallel. Since not alway all threads are needed for a computation CuLi perform conditional branches. Due to the hardware architecture, all threads of a warp execute the first branch and discard the results if they are not set active. Those branches impact the performance but the thread finish one after another or join on a barrier. However, if one thread of a warp enters an long-last loop all other warp threads wait for this one, i.e. in case of an endless loop the computation cannot terminate.

For multi-threading and synchronization, CuLi uses a busy-waiting, i.e. a long-lasting loop, that waits for the change of one value as reaction to changes in the program flow, this can end-up in a livelock. Threads in a waiting loop could hinder other threads from changing the condition that interrupts the loop. If that happens, the code is stuck in an infinite loop.

CuLi has one master thread and a lot of workers. Since grids in CUDA have to be symmetrical, i.e. each block in the grid has the same size and therefore the same amount of threads, the block with index `0`, the block of the master thread, has the same amount of threads as any other block. To ensure that the master thread can work freely, the other threads of this block have to be disabled (see Figure 12). With this solution it is possible to generally have more than one thread per block, although it wastes nearly one complete block.

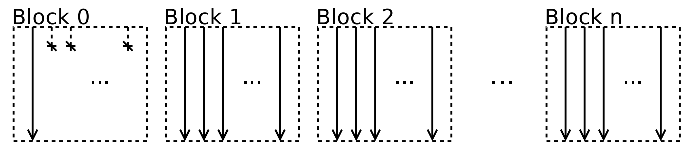


Fig. 12: The solution to the warp divergence of the master threads block as it is used by CuLi.

The other part of CuLi suffering from warp divergence is the distribution of work by the built-in function `|||`. Due to this function an arbitrary number of jobs can be created that will be distributed onto the active workers. This is no problem as long as the number of jobs is a multiple of 32 (all threads of one warp get a synchronization signal). However, if only

one of the threads is not assigned a job, that thread will stay in its waiting loop. Since the `|||`-expression terminates only after all workers have returned their results this will also lead to a livelock.

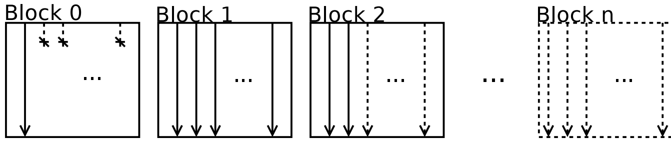


Fig. 13: The solution to the warp divergence of the worker blocks as it is used by CuLi. The solid boxes show activated blocks, the solid arrows show active threads, dotted arrows and boxes show inactive threads and blocks.

To solve this issue CuLi uses an additional synchronization flag for each block to solve this problem (see Alg. 1 line 6). Every thread of a block checks this synchronization flag in each loop. The `|||`-expression activates that flag for parallel execution each time it assigns a job to every worker in that block or if there are no more jobs to distribute (illustrated in Figure 13). Afterwards each thread checks if it has some assigned work (Alg. 1 line 6). After all threads have reached a barrier the first thread of each block resets the synchronization flag (Alg. 1 line 12) before all threads enter the busy-waiting loop again.

Algorithm 1 Evaluation loop executed by all workers.

```
1:  Required local: threadID, blockThreadID
2:  Required shared: blockSyncFlag, availableWorkArray
3:  procedure WORKERLOOP
4:      while running do                                  ▷ evaluation loop
5:          threadBlockBarrier()              ▷ sync all threads in block
6:          while blockSyncFlag = 0 do                    ▷ no work
7:              doNothing()                        ▷ wait for sync flag
8:          if availableWorkArray[threadID] = 1 then
9:              evaluateTask()                              ▷ evaluate
10:             availableWorkArray[threadID] ← 0
11:         threadBlockBarrier()              ▷ sync all threads in block
12:         if blockThreadID = 0 then
13:             blockSyncPtr ← 1                       ▷ reset sync flag
```

The current implementation allows for executing and modifying Lisp programs at runtime on the GPU. A negative point of our implementation is the fact that the size of the possible inputs is currently limited. This limitation is reasoned by the organization of the nodes used for storing objects. A missing feature to mention is the unavailability of program internal file I/O in the current version. This feature can be realized by using the buffer for exchanging messages between host and device for this purpose and will be added in future versions.

IV. EVALUATION

We evaluated CuLi on systems equipped with different GPUs and CPUs. The only system that has no GPU is equipped with four AMD 6272 CPUs (64 cores, 1.8 GHz and 128 GiB DDR3 RAM). All other nodes are equipped with an Intel Xeon E5-2620 CPU (6 core + hyperthreads, 2.00 GHz, and 16 GiB DDR3 RAM). For our GPU tests we use Tesla GPUs as well as consumer GeForce GPUs, namely: Tesla C2075, Tesla K20, Tesla M40, GeForce GTX480, GeForce GTX680 and GeForce GTX1080. Scientific Linux 6.4 was installed on all systems.

CuLi's upload of input strings was not bounded by the bandwidth limits of PCIe as the strings are rather short (17 to 8207 characters per transfer, around 8 KB in size). In our test all threads compute the 5th Fibonacci number recursively. Although this test is rather small, it provides a sufficient amount of operations and recursions. Another important fact is that the size of the tests is sufficient for showing the trends in GPU evolution.

In order to compare the performance of these systems, different scenarios were tested and run multiple times. In our evaluation we considered the base latency (set up and shutdown time), the runtime on the devices, and the kernel proportions (timings of different execution phases).

a) Base latency: The base latency describes the pure setup time of CuLi, i. e. the time for preparing the built-in functions and setting up the REPL. The different timings are shown in Figure 14. Interestingly, the newer the GPU, the higher the base latency. The latency of the GTX 680 is about six times lower than the latency of the GTX1080 or the Tesla M40.

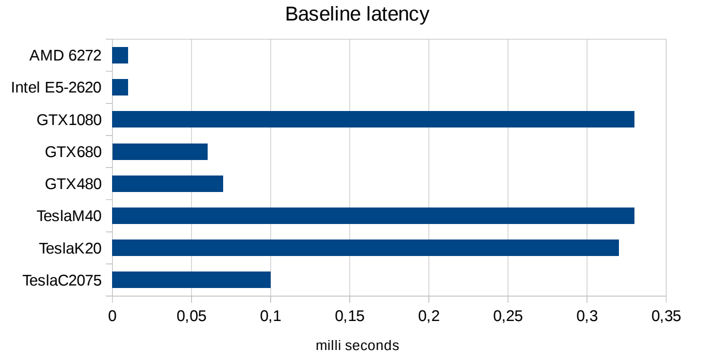


Fig. 14: Base latency for all devices. This includes the time needed for the start and graceful stop of CuLi.

The CPUs outperform the GPUs significantly. Both systems are more than thirty times faster than the fastest GPU.

b) Runtime on the devices: When the amount of processed data increases, the execution time increases, too. Figure 15 illustrates the dependency between the size of the input and the execution time. The number of threads represent the input size since the input is distributed among the threads. The GPUs were clearly outperformed by the CPUs by a factor of at least ten.

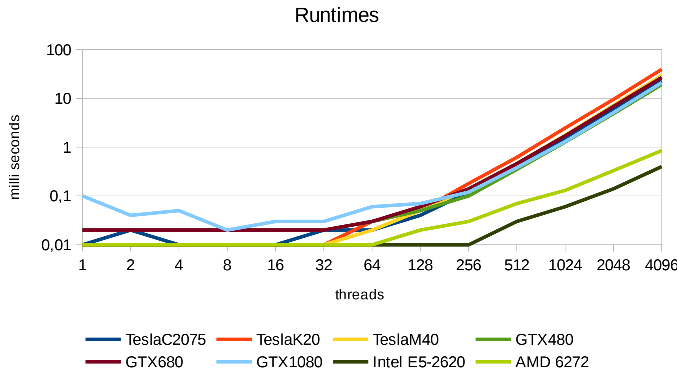


Fig. 15: Runtime for all devices. The time scale is logarithmic.

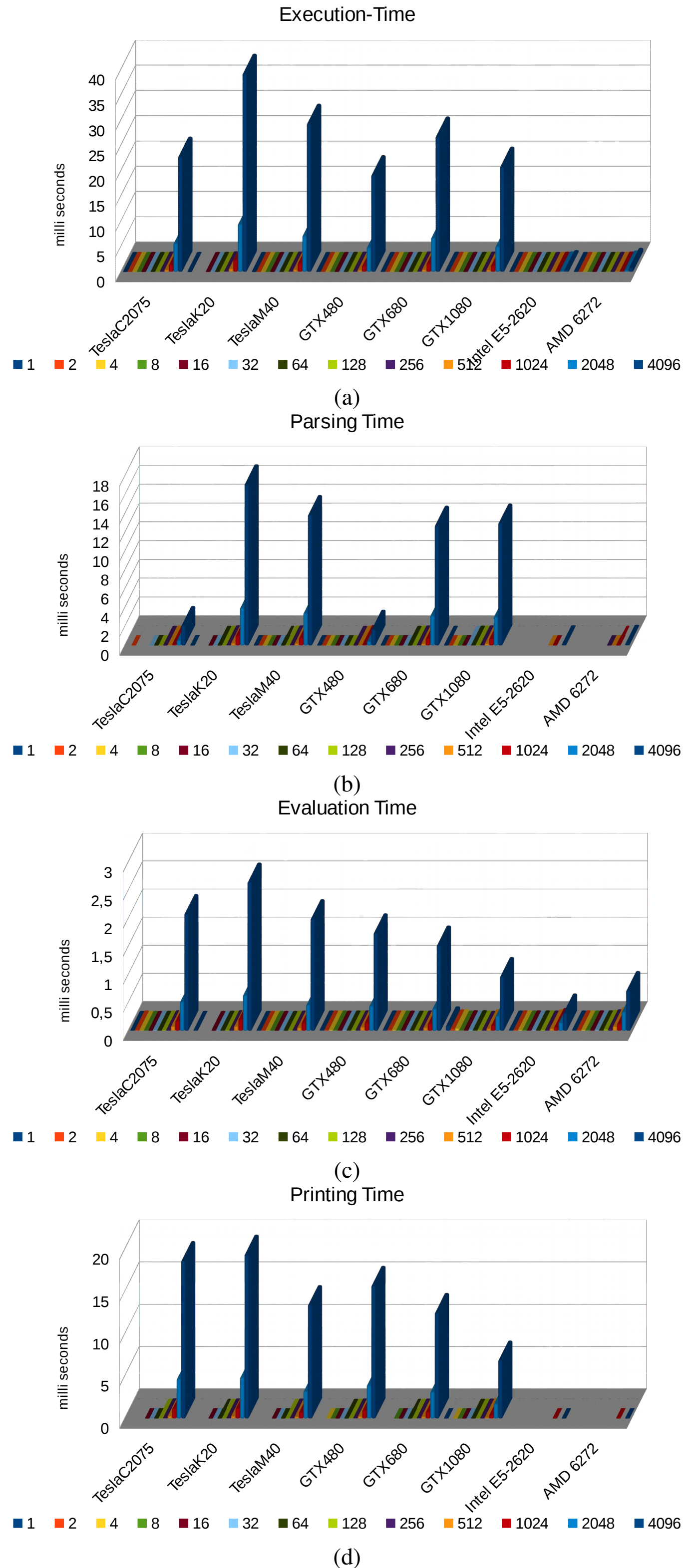


Fig. 16: Different aspects of the kernel execution on GPUs and CPUs. The different colors illustrate the number of used threads.

Although the execution times on the GPUs are higher than on the CPUs, the increase of the run times behaves very similar. All devices show a plateau for 1 to 64 elements. For longer vectors there is a linear growth in runtime.

All GPUs achieve similar runtimes. In this scenario the GTX480 is the fastest GPU followed GTX1080. This result can be explained by the good string parsing performance of Fermi GPUs (Tesla C2075, GTX480). Figure 16b shows the timings required for parsing.

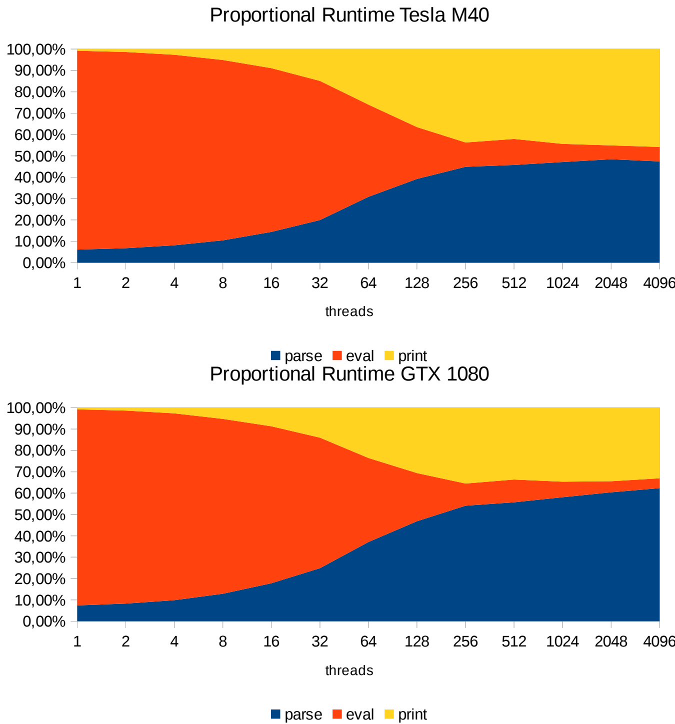
Parsing on Fermi based GPUs outperforms the newer GPUs. This can be explained by architecture changes in the modern GPUs. The two most important changes are the reduction of the L2 Cache size (from 768 KiB to 512 KiB) and the narrowing of the memory interface from 384 bit to 256 bit. These changes reduce the throughput for single threads. However, the evaluation of the other operations (Figure 16c) and printing (Figure 16d) show a clear trend that here the performance of GPUs draws nearer to the one of CPUs.

c) Kernel proportions: Each command execution consists of the tree phases on the GPU: parsing, evaluating, and printing. The accumulated time of these phases corresponds to the execution time on the GPU. The following figures show the proportions of the different steps during an execution. The figures are separated by the device evaluated. In general, all devices behave similarly. Only the GPU based on the Fermi architecture differs in the parsing phase.

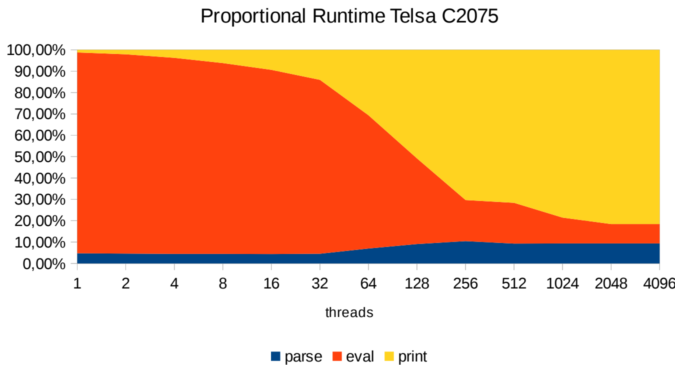
Figure 17 shows the differences in the kernel runtimes on the older GPUs in our tests. While parsing can require more than 50% of the runtime in GPUs newer than Fermi, the parsing on older GPUs never exceeds 11%.

On the system using the AMD CPU, parsing and printing is almost negligible (see Figure 18). Here the runtime is also dominated by the evaluation phase.

Nevertheless, the illustrations in Figure 16 show that, beside the parsing speed, the performance of GPUs get closer to the performance of CPUs. The only exceptions are the Fermi based GPUs during the parsing phase. Especially the trend of the evaluation phase shows that the newer the GPU, the lower the computation time (see Figure 16c). If this trend continues, GPUs and CPUs will achieve the same performance in a few generations.



(a) The proportions of the kernel runtimes on nearly all GPUs look the same.



(b) Only the GPUs based on the Fermi architecture behave differently.

Fig. 17: Comparison of the different phases of the kernel runtime on two GPUs.

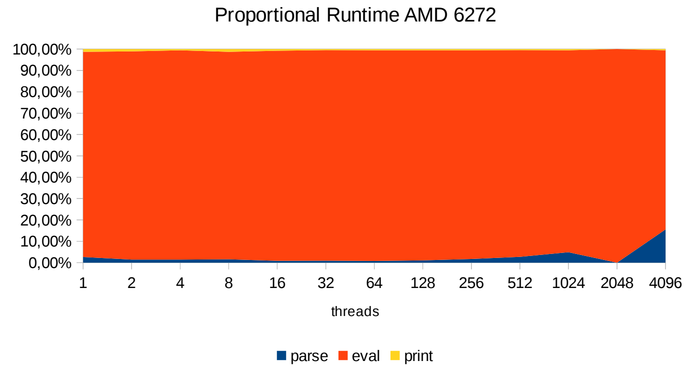


Fig. 18: Proportions of the kernel runtime on AMD CPU (64 Threads).

V. Conclusion

In this work we have introduced CuLi, a Lisp implementation running completely on GPU. CuLi interactively executes Lisp commands. Thus it allows dynamic GPU programming. There is no need for compiling the code to a binary program. Instead Lisp expressions are sent to the GPU where a run-evaluate-print-loop evaluates the given user input.

Since Lisp is a functional programming language, it is a good candidate for using parallel compute resources. This programming paradigm supports program code parallelization since it denies side-effects.

The current GPU version is outperformed by the CPU version. This is largely due to the offset of the baseline latency of starting the CUDA context. Our tests indicate that GPUs will outperform multi-processor computers because they use memory accessible to all threads at the same speed. CuLi's performance highly depends on the single-thread performance of the hardware, due to the dominant part of the execution flow, the parsing of the input. Since GPUs are not primarily built for single thread performance, this seriously limits the potential performance of CuLi. Parsing takes a great share of the runtime of every interpreted language and CuLi is no exception. Improvements in this area will have the most impact on the overall performance of CuLi. Our tests show that CuLi profits from new hardware generations. If the trend continues, the performance gap between CPU and GPU will become smaller with every new GPU generation. When the performance of the single hardware threads on CPU and GPU is equal, it will be possible to harvest the massive parallel potential of GPUs. One important observation that strengthens our assumptions are the efforts made to increase the integer-8 (i.e. character) performance of GPUs. Driven by the machine learning community, GPU vendors increase these capabilities on GPUs which (most probably) also have a positive impact on string parsing.

New versions of NVidia GPUs provide a new threading model that is closer to the model provided on CPUs. The next versions of CuLi will exploit this new feature and gain advantages by this. Another profitable feature is the configurable cache of these devices which can help to reduce the parsing penalties.

Availability

CuLi and test applications will be published on the web server of the Johannes Gutenberg University Mainz under https://version.zdv.uni-mainz.de.

REFERENCES

[1] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Sam Tobin-Hochstadt. The fortress language specification version 1.0, mar 2008. http://research.sun.com/projects/plrg/fortress.pdf.

[2] M. Aswad, P. W. Trinder, and H. Loidl. Architecture aware parallel programming in glasgow parallel haskell (GPH). In *Proceedings of the International Conference on Computational Science, ICCS 2012, Omaha, Nebraska, USA, 4-6 June, 2012*, pages 1807–1816, 2012.

[3] L. Bergstrom and J. Reppy. Nested data-parallelism on the gpu. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 247–258, New York, NY, USA, 2012. ACM.

[4] G. L. Bernstein, C. Shah, C. Lemire, Z. DeVito, M. Fisher, P. Levis, and P. Hanrahan. Ebb: A DSL for Physical Simulation on CPUs and GPUs. *CoRR*, pages 1–12, 2015.

[5] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep learning with cots hpc systems. In S. Dasgupta and D. Mcallester, editors, *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, volume 28, pages 1337–1345. JMLR Workshop and Conference Proceedings, May 2013.

[6] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, Jan. 1998.

[7] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: A domain specific language for building portable mesh-based PDE solvers. *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, 2011.

[8] R. Farber. *Parallel Programming with OpenACC*. Morgan Kaufmann Publishers Inc., 1st edition, 2016.

[9] M. Fluet, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Manticore: A heterogeneous parallel language. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 37–44, New York, NY, USA, 2007. ACM.

[10] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous Computing with OpenCL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[11] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, Oct. 1985.

[12] K. Hammond and G. Michelson, editors. *Research Directions in Parallel Functional Programming*. Springer-Verlag, London, UK, UK, 2000.

[13] P. Hanrahan and J. Lawson. A Language for Shading and Lighting Calculations. *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, 24(4):289–298, 1990.

[14] E. Holk, M. Pathirage, A. Chauhan, A. Lumsdaine, and N. D. Matsakis. GPU programming in rust: Implementing high-level abstractions in a systems-level language. *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium Workshops and PhD Forum, IPDPSW 2013*, (Section III):315–324, 2013.

[15] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 448–456, 2015.

[16] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[17] A. Klöckner. Loo.py: transformation-based code generation for GPUs and CPUs. pages 1–6, 2014.

[18] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3):157–174, 2012.

[19] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012.*, pages 1106–1114, 2012.

[20] B. Lippmeier, M. M. Chakravarty, G. Keller, R. Leshchinskiy, and S. Peyton Jones. Work efficient higher-order vectorisation. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 259–270, New York, NY, USA, 2012. ACM.

[21] G. Mainland, R. Leshchinskiy, and S. Peyton Jones. Exploiting vector instructions with generalized stream fusion. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 37–48, New York, NY, USA, 2013. ACM.

[22] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, Apr. 1960.

[23] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier. Optimising purely functional gpu programs. *SIGPLAN Not.*, 48(9):49–60, Sept. 2013.

[24] J. Nicholls. JavaScript on the GPU, 2012. *https://www.slideshare.net/jarrednicholls/javascript-on-the-gpu*.

[25] M. Pelletier. hillisp – CUDA parallel Lisp, 2018. *https://github.com/michelp/hillisp*.

[26] L. Petersen, D. Orchard, and N. Glew. Automatic simd vectorization for haskell. *SIGPLAN Not.*, 48(9):25–36, Sept. 2013.

[27] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics*, 31(4):1–12, 2012.

[28] S. Ryu. Parsing fortress syntax. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 76–84, New York, NY, USA, 2009. ACM.

[29] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.

[30] D. Strnad and N. Guid. Parallel alpha-beta algorithm on the gpu. In *Information Technology Interfaces (ITI), Proceedings of the ITI 2011 33rd International Conference on*, pages 571–576, June 2011.

[31] A. Stromme, R. Carlson, and T. Newhall. Chestnut. *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores - PMAM '12*, pages 156–167, 2012.

[32] J. A. Stuart and J. D. Owens. Efficient Synchronization Primitives for GPUs. *October*, page 13, 2011.

[33] M. Takagi. Cl-cuda – Library to use NVIDIA CUDA in Common Lisp programs, 2018. *https://github.com/takagi/cl-cuda*.

[34] P. W. Trinder, K. Hammond, J. S. Mattson, Jr., A. S. Partridge, and S. L. Peyton Jones. Gum: A portable parallel implementation of haskell. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, pages 79–88, New York, NY, USA, 1996. ACM.

[35] J. Wang, N. Rubin, and S. Yalamanchili. Paralleljs: An execution framework for javascript on heterogeneous systems. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, pages 72:72–72:80, New York, NY, USA, 2014. ACM.

[36] S. Xiao and W. C. Feng. Inter-block GPU communication via fast barrier synchronization. *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010*, 2010.