Tiako P.F. (ed) Intelligent Computing and Consumer Support Applications. Chronicle of Computing. OkIP
 https://doi.org/10.55432/978-1-6692-0003-1_4

LitP: Language-integrated Tensor Parallelism

Philipp Kramer
Lecturer at OST
Department of Computer Science
Rapperswil, Switzerland

Abstract

LitP, is a new data-parallel programming model for .NET respectively C#. It has been designed specifically to run data-parallel calculations in a managed runtime system. It can be used to solve a wide range of problems. Programs are described in vectorized form, but abstract from low-level architectural details of the target hardware. This allows for a good compromise between simplicity and performance. The presented library currently targets GPUs and CPUs, but is in principle well suited to any kind of general purpose, parallel processing unit.

Keywords: programming model, DSL, SIMD, data-parallel, data-flow, reactive, GPU, compilation

1 Introduction

This work allows to program GPUs, and in principle also any other data-parallel hardware, in a clean, concise, high-level, platform independent programming model not requiring low-level knowledge of the target hardware. It is a vectorized programming model, hence this is solved by the programmer. All other machine-centric features are however abstracted and do not additionally complicate the high-level source code. This allows for a good compromise between readability and performance. The framework also handles necessary data transfers between host and device as well as garbage collection on the GPU, which is handled by the .NET GC.

CUDA C[8] and OpenCL[33] are the most popular GPU programming models. They convey the GPU *single instruction multiple data* (SIMD) execution principle directly to the programmer. In addition to this fundamental requirement, they also expose many performance-relevant architectural hardware features that need to be addressed in programming. While CUDA or OpenCL implementations can be written in a decent high-level programming language (C++), the code remains in fact dominated by low-level concerns such as coalescing, explicit usage of shared memory, warp-thread-divergence or memory bank conflicts. Writing GPU kernels by hand is demanding and the reason for numerous publications on how to solve various specific problems with GPUs. Furthermore, programming directly in this model also entails writing quite some boiler-plate code for data management.

Corresponding Author: philipp.kramer@ost.ch

In general, there is still a lack of well-performing, easy to use high-level programming frameworks for SIMD architectures.

The LitP API is similar to the Java stream API[28] or .NET's LINQ[20]. Although both of the latter frameworks offer parallel processing of their streams, they have not been designed for it from the ground up. The full spectrum of their methods cannot be mapped to efficient GPU code. Even for frameworks that can translate a subset of LINQ or the Java Stream API to efficient GPU code, programmers are not aware of performance implications from an API standpoint, and they are likely to produce inefficiently running programs. The set of LitP methods naturally leads the programmer to formulate programs that can be well translated to data-parallel architectures respectively only to use it for suitable applications. Due to its clean, simple design, the hurdle to use it is low and makes the GPU a viable option for cases that would otherwise not be considered. The programming model is embedded into C# by cross-compiling stream API like methods with lambda-functions embedded in the host program. This language embedding also proliferates easy adoption by programmers. As evidence of the high-level nature of the programming model and the performance of the compiled programs, Nvidia CUDA and CPUs are targeted.

The key contribution of this work is the careful design of a clean, high-level programming model for data-parallel calculations. The strength of the implementation is optimization by inlining as a better alternative to the occasionally proposed approach to compose or fuse fully optimized GPU kernels.

After a short overview of related work in section 1.1, we present the programming model in section 2 by means of a series of simple examples. Subsequently, we describe the inner workings of the runtime support in section 3 and evaluate the performance of a small number of calculations in section 4 by comparing their execution time with relevant benchmark implementations.

1.1 Related work

The most straight-forward approach is to embed the CUDA C respectively Open CL model as is in a high-level programming language[1, 5, 6, 10, 16, 17]. While this represents a step in the direction of this proposal, the integration remains low-level on a semantical level, carrying over much

Presented at the 2022 OkIP International Conference
on Advances in High-Performance Computing (AHPC)
in Oklahoma City, OK, USA, and Online, on October 3-6, 2022

of the complexity since the transparency of the GPU architecture remains. There are also annotation or directive based C++ centric approaches such as OpenACC[13, 30] and C++ AMP[32] .

The largest family of high-level programming frameworks that allow targeting GPUs can be subsumed under the terms algorithm building blocks[4, 19, 21, 26], stream APIs[3, 14, 20, 27] and (reactive) dataflow frameworks[7, 18, 24]. The most prominent and probably most advanced of these frameworks is Google's TensorFlow[2] mainly aimed at large-scale machine learning; it can however also be used for other purposes. TensorFlow is a large toolbox that is not quickly learned and requires expert knowledge when extending it with functionality that does not come out of the box.

An often used approach in building-block, stream or general dataflow frameworks is to compile blocks, operations or methods individually to highly optimized kernels and then to fuse a number of them together to optimize GPU global memory access[11, 25, 29, 31]. Our approach follows quite an opposite strategy.

A conceptually clean and highliy optimized framework is Halide[23]. It separates the formulation of *what*, referred to as algorithm, from *how*, referred to as schedule, it is calculated. Calculations are implicitly happening in a coordinate system. The algorithm is basically reduced to the formula describing each result element depending on its position in the coordinate system. The schedule describes how the calculation process is organized. This decoupling enables high performance without sacrificing code clarity. Halide is rather aimed at parallel programming experts preferring a high-level programming framework.

Another system that is very advanced and aimed at expert users is Spiral[12]. It applies sophisticated optimization algorithms and can target a wide range of hardware. It can only be used to its full extent using a dedicated programming language. The capabilities and performance properties of the hardware need to be described in an abstract manner. Also, programs are formulated very abstract such that different degrees of freedom are left open allowing the optimization process to use them according to the hardware's properties and capabilities. The system's power makes however also its use demanding. There is a language integration into Scala, but this embedding does not allow using Spiral to its full extent.

2 Programming Model

2.1 Program structure and language embedding

LitP programs are written mostly in fluent notation. The selection of API methods follows a main principle: offer as few methods as possible that allow to solve as many problems as possible. The methods can be further subdivided into calculating and restructuring methods. In addition, the way to do multi-dimensional calculations is via multi-dimensional arrays instead of jagged arrays respectively nesting like e.g. `IEnumerable<IEnumerable<T>>` in LINQ. All calculations are side-effect-free, conceptually producing a new result array from one or more argument arrays, thus all data can be considered as being immutable. The structure of these calculations is restricted to a directed acyclic graph (DAG). A calculation graph can be run on a particular hardware, in case this target is not the invoking host machine, this implies automatically transferring arguments and results. Due to the DAG restriction, LitP programs comprised of a finite number of methods are conceptually guaranteed to terminate. The calculation-graph is composed by composing methods via their arguments and results of type `Tensor<T>`, in which `T` must be a value type.

Tensors represent arguments as well as results of all methods. In contrast to C#'s `IEnumerable<T>` interface, there is no automatic conversion from C# arrays to tensors. This helps making the programmer aware of possible delay that can happen when tensors are transferred between host and device. Instead, a tensor-object representing the array must be created with the array extension method `ToConst()`. [1]

```
var a = new int[...];
var t = a.ToConst();
```

2.2 Methods to define and execute calculations

The method `IMap()` applies a lambda-function to each argument element independently producing a tensor with the result elements. In more formal notation, `IMap()` describes for a small number of tensors with associated arrays a_0 to a_n and a result array r a function of the form $\forall i : (a_0[i], ..., a_n[i]) \rightarrow r[i]$. It is permitted to use the same tensor as argument more than once. The 'I' in `IMap()` simply stands for *identity* referring to the one-to-one index mapping. All LitP lambdas are so called C# expression lambdas. As the term suggests, expression lambdas cannot contain programming constructs like assignments or loops. In addition to the restrictions imposed by C#, reference types are also not supported, and function calls are evaluated when defining the calculation and the returned value will be constant for the calculation.

Continuing the code above, the elementwise square of an array is formulated as

```
var square = t.IMap(v => v*v);
```

This just represents the calculation, which can subsequently be evaluated on a device.

```
int[] result = GPU.Evaluate(square);
```

`CPU` or `GPU` specify the platform and `Evaluate()` takes a tensor as argument, executes the calculation and returns

[1] All example code is written in C# and each snippet is self-contained, except for variables that can be defined in preceding code-snippets.

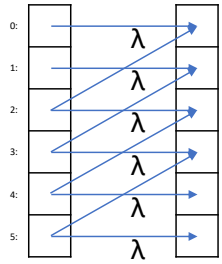

Figure 1. `VMap` method: left is the argument and right the result array.

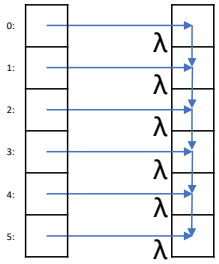

Figure 2. `Sweep` method: the arrays are traversed sequentially.

the result array. There is a more flexible way to execute calculations shown in section 2.4.

With multiple arguments, purely fluent notation is not possible. For these overloads, LitP resorts back to all normal arguments instead of using one of them as `this`-parameter; e.g., adding two tensors together is formulated as

```
var b = new int[...];
var t2 = b.ToConst();
var sum = Tensor1D<int>.IMap(t, t2, (v, w) => v + w);
```

Lambdas can use the element's index with the static property `Index`. This also allows to produce initial argument arrays directly on the platform without having to transfer them.

```
var range = new Tensor1D<int>(length)
  .IMap(_ => Index);
```

The `VMap()` method combines two elements from the same argument array. It features an additional offset parameter (noted o in the formula below) that identifies the argument's element index relative to the result element. More formally, `VMap()` describes a calculation of the form $\forall i : (a[i], a[i+o]) \rightarrow r[i]$. The simple function $\forall i : a[i] = a[i] + a[i+2]$ is written as

```
var r = t.VMap(vs => vs[0] + vs[1], 2);
```

The lambda describes the value calculation whose argument is an array containing the element values. The following 2-argument represents the offset. `vs[0]` refers to `a[i]` and `vs[1]` refers to `a[i + 2]`. `VMap()` also works with multiple, possibly negative offsets adding further elements to the parameter-array. The 'V' refers to the two-to-one index mapping as can be seen in figure 1 visualizing the example code above. The general `VMap()` method describes a one-dimensional convolution e.g. with the gauss kernel.

```
t.VMap(vs =>
  0.005980 * vs[0] +
  0.060626 * vs[1] +
  0.241843 * vs[2] +
  0.383103 * vs[3] +
  0.241843 * vs[4] +
  0.060626 * vs[5] +
  0.005980 * vs[6],
  1, 2, 3, 4, 5, 6)
```

To arrive at more useful calculations, methods must of course be composed. The sum of all elements in an array can for example be calculated with

```
int len = IntMath.Pow(2, IntMath.Log2(a.Length) + 1);
while (len > 1) {
  len /= 2;
  t = t
    .VMap(vs => vs[0] + vs[1], len)
    .Resize(len);
}
```

`Resize()` can return a shortened or extended tensor with the same dimensionality. When evaluated, the calculation above yields a result array containing a single element with the sum of all elements.

All elements within array bounds are calculated. For argument elements outside array bounds the default value is assumed. This behavior can be overridden using the C# coalescing operator "??" as used in the following example.

```
var p = t.VMap(vs => vs[0] * (vs[1] ?? 1), 1);
```

Numbers can be summed up using the `Reduce()` method. Unlike the map-methods, it features a lambda with two scalar parameters, the first represents a lower or left value and the second a higher or right value. The framework is free to choose any tree-topology to aggregate all values represented by `t`. `Reduce()` returns an object of type `Tensor0D<T>` that represents a single number.

```
t.Reduce((l, r) => l + r)
```

`Reduce()` is an exception to the general design philosophy: it is both calculating and restructuring. It is part of the API for performance reasons, because it allows the framework to select a reduction strategy that fits the hardware. For example, an efficient GPU aggregation algorithm for large arrays must reduce all data in a thread-block down to a single number and only write back that number to global memory instead of half the data as shown in the aggregation example above using `VMap()`.

`Sweep()` describes a sequential traversal of the tensor either in positive, visualized in figure 2, or in negative direction. A simple example calculating the scan sum is

```
var r = t.Sweep(vs => vs[0] + vs[1], -1);
```

In general, `Sweep()` describes a front of one or more elements traversing a tensor one element at a time. The front cannot leave out any elements. Using the `Sweep()` method, the Fibonacci numbers can be calculated with

```
var fib = new Tensor1D<int>(length)
  .Sweep(vs => (vs[1] ?? 1) + vs[2], -1, -2);
```

This might not look like a useful GPU method at the moment, but that will become apparent subsequently.

2.3 Methods for multi-dimensional calculations

In addition to the so far presented classes `Tensor0D<T>` and `Tensor1D<T>`, LitP features the classes `Tensor2D<T>` and `Tensor3D<T>` containing the multi-dimensional equivalents of these methods. This multi-dimensional extension is straight-forward for `ToConst()`, `IMap()`, `VMap()` and `Resize()`.

In multiple dimensions, there is the additional method `Transpose()` that is self-explanatory. Also, in 0 to 2 dimensions there is the `Replicate()` method that inserts a new dimension by replicating all values along a new dimension a particular number of times.

The following example shows how to calculate the squared distance matrix. The two arrays `px` and `py` contain the x and y coordinates of the points. The inner maps calculate the squared x and y deltas of the points. The first `Replicate()` argument denotes at which position to insert the new dimension and is chosen such that all combinations (in fact twice as many) are generated. The outer `IMap()` adds them together.

```
int[] px = ...;
int[] py = ...;
int len = px.Length;
var tx = px.ToConst();
var ty = py.ToConst();
var dMatrix = Tensor2D<int>.IMap(
  Tensor2D<int>.IMap(
    tx.Replicate(1, len),
    tx.Replicate(0, len),
    (a, b) => (a - b)*(a - b)),
  Tensor2D<int>.IMap(
    ty.Replicate(1, len),
    ty.Replicate(0, len),
    (a, b) => (a - b)*(a - b)),
  (x2, y2) => x2 + y2);
```

For `VMap()`, the offsets obviously need to be specified with the appropriate number of components to calculate e.g. a two-dimensional box convolution.

```
var convolution = m.VMap(
  vs => Sum(vs) / Count(vs),
        (0, 1), (0, 2),
  (1, 0), (1, 1), (1, 2),
  (2, 0), (2, 1), (2, 2));
```

This example also demonstrates the usage of the two convenience methods `Sum()` and `Count()`, which can either be given the lambda-parameter as argument or any number of individual argument array elements.

In two dimensions, if one offset is specified for `Sweep()`, the front line moves in that direction. In case offsets are provided in both directions, the front moves diagonal as shown in figure 3 for the sequential steps 1, 2, 3, ... The same behavior generalizes to three dimensions, e.g. when offsets in two directions are given in a three dimensional calculation, the plane moves orthogonal over the plane shown in figure 3. The maximum angle between any pair of offsets is 90°, which is a necessary and sufficient condition that all elements of the front can be calculated independently and thus in parallel in each step.

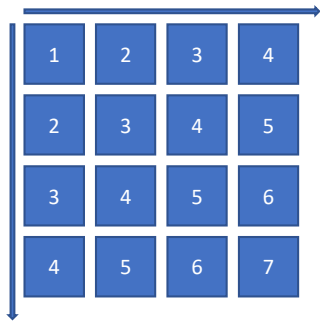

Figure 3. steps of a `Sweep()` calculation front.

`Sweep()` can be used for dynamic programming problems, for example text comparison. The first `IMap()` creates a product matrix and sets a 1 at coordinates where the letters from the first and the second word match. `Sweep()` subsequently traverses the matrix effectively calculating the maximal number of matches following any path from the origin to a particular front element.

```
var xs = "MZJAWXU".ToIntTensor();
var ys = "XMJYAUZ".ToIntTensor();
var maxMatch = Tensor2D<int>
  .IMap(
    xs.Replicate(1, b.Length),
    ys.Replicate(0, a.Length),
    (x, y) => x == y ? 1 : 0)
  .Sweep(
    vs => Max(vs[0]+vs[3], vs[1], vs[2]),
    (-1, -0),
    (-0, -1),
    (-1, -1));
```

`Max()` and `Min()` are two further convenience methods like `Sum()`.

The last example in this section is matrix multiplication.

```
int[,] a = ...;
int[,] b = ...;
var ta = a.ToConst();
var tb = b.ToConst();
var product = Tensor3D<int>
  .IMap(
    ta.Replicate(1, b.GetLength(1)),
    tb.Transpose()
      .Replicate(0, a.GetLength(0)),
    (a, b) => a*b)
  .Reduce((i, o) => i + o),
```

Multiplication in LitP is conceptually three dimensional reflecting the fact that this involves $O(n^3)$ scalar multiplications. The row-index is 0 and the column index is 1. So,

the number of columns in `a` must correspond to the number of rows in `b`. Since the size of the `IMap()` arguments must be equal, each argument must be extended by the non-shared dimension of the other. `Reduce()` then aggregates the pairwise products along the shared dimension.

2.4 Calculations with multiple results

If the use case has multiple results, the compact `Evaluate()` call is not sufficient anymore; instead, an `Evaluation` object can be created. All calculation result tensors need to be passed to its constructor. Subsequently, the calculation is triggered and the results can be retrieved using the `Get()` method. The evaluation is pull-based, meaning that only calculations are performed whose result is directly or indirectly requested. Calls to `Get()` are blocking.

The following example calculates the elementwise differences and their squares, both the intermediate as well as the end result are used by the host program.

```
var deltas = t.VMap(vs => vs[1] - vs[0], 1);
var squares = deltas.IMap(v => v*v);
var eval = new CpuEvaluation(deltas, squares);
... = eval.Get(deltas);
... = eval.Get(squares);
```

The initially introduced `Evaluate()` call is simply a convenience function applying the above for the special case of one result.

In case of CPU execution, memory is deallocated by the dotnet garbage collector. LitP can dispose intermediate results on the GPU automatically as soon as they are not used anymore because it has the full information about the dataflow.

3 Compilation and Execution

The runtime support for the execution of LitP programs is comprised of a just-in-time compilation and an execution phase. The compilation phase constructs a high-level AST-like structure using C# reflection of lambda expressions. The structure contains a node for each method invocation. This structure is optimized by extensively inlining methods as far as possible. This is key for GPU performance especially in this model, because the individual methods are too primitive and would, when compiled to individual kernels, most often be memory bound. Inlining effectively reduces loads and stores. All `Replicate()` and `Transpose()` methods are inlined and, because of this, artificially introduced dimensionality blow up such as in the matrix multiplication example is not a performance issue. `IMap()` is also always inlined, but `VMap()` only very limited because there is a blow up of parameters. This allows e.g. to compile the matrix multiplication into a single GPU kernel.

The high-level structure is then compiled down to .NET IL code, which is finally executed. The IL code is of course different for the CPU and the GPU target. The main purpose of the CPU implementation is currently to serve as functional reference and evidence of multi-platform capability.

There is a number of GPU specific optimization techniques that need to be applied in order to get well optimized kernels. The current implementation is in that respect very basic leaving a lot of potential for performance improvements. Currently, only two GPU specific optimizations are applied, namely using shared memory and specialized kernels for inner and boundary regions of the calculated arrays. The former helps to further reduce GPU main memory access. The latter uses simplified kernels for inner regions of the involved arrays. The domain of inner regions is implicitly defined by the offsets used in the `VMap()` methods, namely such that no argument of any inner element is out-of-bounds. This allows to eliminate boundary checks that are necessary in the equivalent boundary version of the kernel. Further (GPU specific) optimization are admittedly also needed to generate well optimized kernels. The .NET IL code targeting the GPU is finally just-in-time cross-compiled and executed using the *Alea GPU* framework[22].

In the execution phase, the nodes of the optimized structure are scheduled on the target platform. In case of the GPU, this includes transferring the data to and from the device as well as managing the used device memory. Scheduling happens fully asynchronously, which is particularly important for the GPU platform that relies on a well filled command pipeline to avoid stalls and use the freedom to run independent calculations in parallel.

4 Evaluation

The evaluation is foremost about validating the programming model with respect to its applicability to a good vaiety of problems that can be mapped well to data-parallel hardware. It is assessed if the generated kernels achieve reasonable performance of the GPU, i.e., that the optimization by inlining is sufficient to achieve this. State-of-the-art performance cannot be expected sind a number of further non-trivial, very GPU specific optimizations have not been implemented. The performance of GPUs depends much more on well optimized code when comparing with CPUs.

Table 1. Used hardware for the performance tests.

hardware property	CPU	GPU
processor	i7-10850H	Quadro RTX 5000
cores	6 physical / 12 logical	3072 = 48mp x 64cores
spec. memory data rate	46GB/s	384 GB/s
onchip cache size	12MB	4 MB

The presented performance numbers are put into perspective by comparing them to hand written C# code using Parallel.For, parallel LINQ where applicable and handwritten CUDA code. The hand-written CUDA code is optimized

Table 2. Compute time measurements in ms with input tensors of approx. 1M elements unless otherwise noted.

case	C# TPL	Parallel LINQ	LitP GPU	CUDA C
cumulative normal distribution	3.7	11	0.38	0.47
option pricing	0.14	0.21	0.26	0.29
convolution	7.2	n/a	0.14	0.1
power iteration	6.7	n/a	1 to 5.7	2.9
matrix multiplication	630	1400	19	17
electrostatic potential	450	n/a	160	160

to the same degree as LitP to serve as directly comparable reference. We benchmark only the compute time not including memory (de-) allocation and transfer delay for the GPU platform, because it does anyway only make sense to use the GPU for less compute intensive tasks when calculation throughput must be maximized and not delay minimized. We do not claim that the presented measurements are representative for all kinds of computations. In the following we discuss each of the benchmark programs. The used hardware is sketched in figure 1 and the measured times are summarized in table 2.

The first case calculates the result of the cumulative normal distribution function for a number of inputs. It applies the approximation function described in reference [9] to each tensor element individually.

The option valuation case prices a single option using the binomial model. Usually this is performed by building up a recombining tree of potential stock prices, the *underlying* of the option, here in 1'000 steps and then evaluating the tree in reverse direction to calculate the present value by weighting the scenarios with corresponding probabilities. To achieve better performance, the tree is completely factored out into a sum of 1'000 addends that can be calculated in parallel and are subsequently summed up. The compute times in the option valuation case are in the expected range for all platforms. This application simply does not involve enough computation for the GPU.

The convolution case simply applies a convolution-kernel with 16 elements. The dimensionality of the convolution is not relevant for performance. In this case, the total amount of calculation is higher than in the preceding case.

Power iteration is a simple algorithm to find the first eigenvector of an invertible matrix. The vector can be found by starting with a random vector and repeatedly multiplying it with the matrix and normalizing it. To ensure that the times are comparable, the number of iterations is fixed to 25 and the length of the vectors is 1'000. In CUDA C, we schedule all launches in a loop. In LitP we have two implementations, one generating a completely unrolled dataflow corresponding to CUDA C, and one sending data to the host to allow for dynamic termination of the loop being the reason for the time difference.

The matrix multiplication case simply measures one matrix multiplication as defined in section 2.3. Parallel-for performs as expected and parallel LINQ is slower by a factor of approximately 2, probably because it intermittently generates data. GPUs are designed for fast matrix multiplications, which is reflected in the measurement even though the GPU kernels are simple.

The electrostatic potential calculation case is taken from the book [15]. It calculates the total the electric field strength at each point of a 2D grid when a number of electric charges is placed at arbitrary points in space.

To summarize, LitP compiles GPU code that corresponds quite well to handwritten unoptimized cuda code. This is different from and, e.g. in the case of matrix multiplication, far better than executing a sequence of highly optimized kernels each corresponding to a single LitP method.

5 Conclusion

This work presents a genuine high-level programming model that is easy to use and able to describe an interesting range of data-parallel algorithms that are efficiently translated to parallel hardware. Using this library, GPU algorithms can be developed in C# in a fully managed environment.

There are opportunities to extend the programming model and further GPU specific optimizations would be desirable. Furthermore, Intel AVX could be supported to have a good alternative with different performance characteristics. There is also an opportunity to further improve the memory management by analyzing the dataflow on tensor level and e.g. performing calculations in place.

The evaluation shows that for suitable problems, the framework has the potential to perform well and can already, without extensive optimization, speed up calculations using the GPU instead of the CPU without the usual complexity of GPU programming.

References

[1] 2016. Aparapi CUDA programming model. http://aparapi.com. [Online; accessed 06-April-2021].

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga,

Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[3] Pritesh Agrawal. 2012. Parallelizing LINQ Program for GPGPU. (2012).

[4] Marco Aldinucci, Sonia Campa, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. 2012. Targeting distributed systems in fastflow. In *European Conference on Parallel Processing*. Springer, 47–56.

[5] Altimesh. 2017. Hybridizer. http://www.altimesh.com/hybridizer-essentials/. [Online; accessed 06-April-2021].

[6] Anaconda. 2012. Numba. http://numba.pydata.org. [Online; accessed 06-April-2021].

[7] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23, 2 (2011), 187–198.

[8] John Cheng, Max Grossman, and Ty McKercher. 2014. *Professional CUDA c programming*. John Wiley & Sons.

[9] Amit Choudhury. 2014. A simple approximation to the area under standard normal curve. *Mathematics and Statistics* 2, 3 (2014), 147–149.

[10] Florent Duguet and Guillaume de Roujoux. 2014. Altimesh Hybridizer™ Enabling Accelerators in. Net and more. In *GPU Technology Conference*.

[11] Jiří Filipovič, Matúš Madzin, Jan Fousek, and Luděk Matyska. 2015. Optimizing CUDA code by kernel fusion: application on BLAS. *The Journal of Supercomputing* 71, 10 (2015), 3934–3957.

[12] Franz Franchetti, Tze Meng Low, Doru Thom Popovici, Richard M Veras, Daniele G Spampinato, Jeremy R Johnson, Markus Püschel, James C Hoe, and José MF Moura. 2018. SPIRAL: Extreme performance portability. *Proc. IEEE* 106, 11 (2018), 1935–1968.

[13] K Gregory and A Miller. 2012. C++ AMP: Accelerated Massive Parallelism with Microsoft Visual C++, Published with the authorization of Microsoft Corporation by O'Relly Media.

[14] Kazuaki Ishizaki, Akihiro Hayashi, Gita Koblents, and Vivek Sarkar. 2015. Compiling and optimizing java 8 programs for gpu execution. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 419–431.

[15] David B Kirk and W Hwu Wen-Mei. 2016. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann.

[16] Andreas Kloeckner. 2009. PyCUDA. https://documen.tician.de/pycuda/. [Online; accessed 06-April-2021].

[17] Andreas Kloeckner. 2009. PyOpenCL. https://documen.tician.de/pyopencl/. [Online; accessed 06-April-2021].

[18] Philipp Kramer, Daniel Egloff, and L Blaser. 2016. The alea reactive dataflow system for gpu parallelization. In *Proc. of the HLGPU 2016 Workshop, HiPEAC*.

[19] Calle Lejdfors and Lennart Ohlsson. 2007. PyGPU: A high-level language for high-speed image processing. In *IADIS International Conference Applied Computing 2007*. 66–81.

[20] Erik Meijer, Brian Beckman, and Gavin Bierman. 2006. Linq: reconciling object, relations and xml in the. net framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 706–706.

[21] Biagio Peccerillo and Sandro Bartolini. 2018. PHAST-A portable high-level modern C++ programming library for GPUs and multi-cores. *IEEE Transactions on Parallel and Distributed Systems* 30, 1 (2018), 174–189.

[22] Quantalea. [n.d.]. Alea GPU. https://developer.nvidia.com/blog/accelerate-net-applications-alea-gpu/. [Online; accessed 06-April-2021].

[23] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[24] Christopher J Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. 2013. Dandelion: a compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 49–68.

[25] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. 2014. Whippletree: Task-based scheduling of dynamic workloads on the GPU. *ACM Transactions on Graphics (TOG)* 33, 6 (2014), 1–11.

[26] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. 2011. Skelcl-a portable skeleton library for high-level gpu programming. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE, 1176–1182.

[27] Nessos Information Technologies. 2014. GpuLinq. https://github.com/nessos/GpuLinq. [Online; accessed 08-April-2021].

[28] Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft. 2014. *Java 8 in action*. Manning publications.

[29] Mohamed Wahib and Naoya Maruyama. 2014. Scalable kernel fusion for memory-bound GPU applications. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 191–202.

[30] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. 2012. OpenACC—first experiences with real-world applications. In *European Conference on Parallel Processing*. Springer, 859–870.

[31] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. 2012. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 107–118.

[32] Erik Wynters. 2016. Fast and easy parallel processing on GPUs using C++ AMP. *Journal of Computing Sciences in Colleges* 31, 6 (2016), 27–33.

[33] JY Xu. 2008. OpenCL–the open standard for parallel programming of heterogeneous systems. (2008).