Efficient Dynamic Derived Field Generation on Many-Core Architectures Using Python

Cyrus Harrison*, Paul Navrátil†, Maysam Moussalem‡, Ming Jiang*, and Hank Childs§
* Lawrence Livermore National Laboratory, [cyrush, jiang4] @ llnl.gov
† Texas Advanced Computing Center, University of Texas at Austin, pnav@tacc.utexas.edu
‡ Department of Computer Science, University of Texas at Austin, maysam@cs.utexas.edu
§ Lawrence Berkeley National Laboratory, hchilds@lbl.gov

***Abstract*—Derived field generation is a critical aspect of many visualization and analysis systems. This capability is frequently implemented by providing users with a language to create new fields and then translating their "programs" into a pipeline of filters that are combined in sequential fashion. Although this design is highly extensible and practical for development, the runtime characteristics of the typical implementation are poor, since it iterates over large arrays many times. As we reconsider visualization and analysis systems for many-core architectures, we must re-think the best way to implement derived fields while being cognizant of data movement. In this paper, we describe a flexible Python-based framework that realizes efficient derived field generation on many-core architectures using OpenCL. Our framework supports the development of different execution strategies for composing operations using a common library of building blocks. We present an evaluation of our framework by testing three execution strategies to explore tradeoffs between runtime performance and memory constraints. We successfully demonstrate our framework in an HPC environment using the vortex detection application on a large-scale simulation.**

I. Introduction

Derived field generation refers to creating new fields from existing fields in the simulation data. This capability is a critical aspect of the exploration process that occurs with scientific visualization and analysis. Typically, simulations are limited in memory size and file size, which restricts the number of fields that can be generated during each run, and users often want to derive fields that are beyond what the simulation considers. As the increasing power cost of data movement will force visualization and analysis to occur *in situ* [5], there is a growing need for a more flexible and efficient approach to generate derived fields that can exploit emerging many-core architectures.

There are three keys areas to derived field generation: 1) the primitives that create new fields, 2) the interface that allows users to dynamically compose these primitives, and 3) the mechanism that transforms and executes the composed primitives. In VisIt [13], this capability comes through its "expression" language, and in EnSight [14] and ParaView [7], through a calculator interface. In all cases, the user is presented with primitives ranging from simple arithmetic functions (e.g., $+, -, *, /$) to complex mathematical operators (e.g., $\cdot, \times, \nabla$) as well as conditionals. The following is a string-based example of composing primitives, which resembles VisIt's expression language, where a new scalar field a is created from scalar fields b and c:

$$a = \textbf{if } (\mathbf{norm}(\mathbf{grad}(b)) > 10) \textbf{ then } (c * c) \textbf{ else } (-c * c)$$

One of the issues with derived field generation is the lack of flexibility to exploit emerging many-core architectures (many-core CPUs and GPUs) that offer unprecedented potential for energy efficient HPC. Due to the diversity of capabilities of many-core architectures, it is imperative to develop different mechanisms, or *execution strategies*, that can transform and execute the composed primitives. When developing these execution strategies, one must consider both runtime performance and memory constraints. The ability to design and test different execution strategies for derived field generation on many-core architectures will be a prerequisite for success on supercomputers in the coming years.

Another ongoing issue with derived field generation is inefficiency. For example, in VisIt's expression language a syntax is defined to enable user composition. VisIt employs a traditional parser that breaks the composed expressions into their atomic form and then constructs a dataflow network that applies filters in one-to-one correspondence with its operations. A shortcoming of this design is that each filter operates on a data array in its entirety before moving on to the next one. This means that the data arrays are iterated over multiple times, which is not only contrary to the *in situ* approach but can also lead to cache thrashing for large data sets.

In this paper, we present a flexible and efficient framework for dynamic derived field generation on many-core architectures through the use of Python and OpenCL [38]. We build upon the previous design, which provides primitives that can be composed dynamically, and focus on the challenge of providing a framework that enables designing and testing efficient execution strategies for derived field generation on many-core architectures. Our framework includes a Python-based parser for user composed expressions, a Python dataflow network that uses PyOpenCL [24] to execute kernels on many-core devices, and a host interface that enables a host application to use our framework *in situ*.

We developed three execution strategies using our framework and use the application of vortex detection in a Rayleigh-Taylor instability simulation to evaluate our strategies. The

evaluation explores runtime performance and memory constraints.

At a high level, the contribution of this paper is illuminating the path for moving a key component of visualization tools to many-core architectures. Specifically, the contributions of this paper are:

- A first-ever development of derived field generation system that works on many-core architectures.
- A flexible Python-based framework for designing and testing efficient execution strategies.
- An evaluation of our framework, exploring memory constraints and the runtime efficiency tradeoffs of our execution strategies.

The rest of the paper is organized as follows: Section II describes related work; Section III describes our framework in detail; Section IV describes our evaluation methodology; in Section V we present our results and then conclude in Section VI.

II. Related Work

Python adoption by the scientific community is growing for simulation and analysis tasks due to the broad set of capabilities provided by popular packages such as NumPy [32] and SciPy [23]. Popular end user visualization tools for large data, such as EnSight [14], ParaView [7] and VisIt [13], use Python to provide client interfaces and interfaces for data manipulation. Many third-party Python modules provide the foundational software used in our framework. We use PLY [9] to build a parser for expression inputs. NumPy and VTK's [37] Python wrappers enable efficient handling of mesh coordinates and field arrays. We use PyOpenCL to access and manage OpenCL devices. PyOpenCL provides interfaces for kernel compilation, kernel dispatch and data array transfer.

Python is well suited for parsing user expressions. In this work, we use PLY, which has been closely modeled on the original Lex and Yacc tools. It uses Look-Ahead LR(1) parsing [6], which is fast, memory-efficient and well-suited for processing our expression grammar. There are many other Python-based parser generators available, including ANTLR [33], SimpleParse [18], PyParsing [28], SPARK [8], Yapps [34] and Plex [17].

The Python community has several active projects that focus on using Python to create high performance code. Cython [10] uses extensions to the Python syntax to introduce type information. This information is used to generate fast compiled code and to create an easy bridge for calling C-functions. Numba [31] translates NumPy based Python into high performance code using LLVM [25] via llvmpy [1]. PyPy [36] uses an aggressive Just-in-Time compiler to provide a fast Python runtime.

A subset of Python performance efforts focus on targeting many-core devices. Theano [11] allows users to generate CUDA and C++ code from NumPy-based Python expressions. Clyther [20] uses a similar strategy to Cython, but creates code that targets OpenCL devices. Our work differs from these efforts in that we provide a derived field generation framework that computes a user's expression leveraging a set of existing OpenCL kernels. The framework allows developers to easily deploy primitives and lets users to dynamically compose them in a host application.

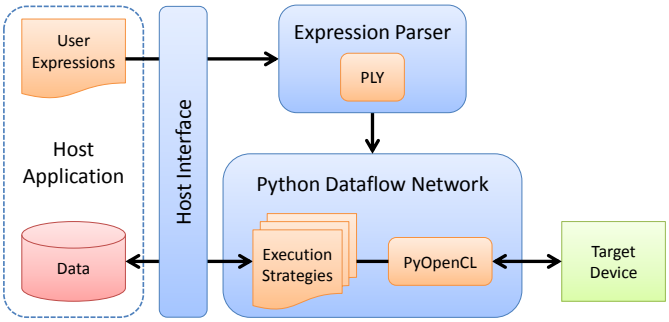


Fig. 1. The system architecture for our derived field generation framework: the expression parser, the Python dataflow network and the host interface. Note that the target device can be a multi-core CPU or a GPU.

Currently, there are many low-level programming frameworks that expose parallelism for multi- and many-core CPUs and accelerators, including GPUs. Frameworks targeting multi- and many-core CPUs include PThreads, OpenMP [15] compiler directives, OpenCL and Intel's Thread Building Blocks [35]. Frameworks targeting GPUs include CUDA [2], OpenCL kernel programming languages and runtimes and OpenACC [3] compiler directives.

There are several emerging efforts focused on producing new programming and data models that can efficiently utilize massive on-node parallelism for visualization and analysis across a diverse set of hardware architectures. These efforts include DAX [30], PISTON [26], EAVL [29] and HyperFlow [39]. Domain specific languages, such as SCOUT [27] and Liszt [16] are also an attractive option, because they provide high-level data and programming abstractions to developers. Our work differs from these efforts in that it focuses on the area of derived field generation: exploring efficient execution strategies and a framework that allows for operations to be implemented independent of execution strategies.

III. System Architecture

This section describes the three-part architecture of our expression framework: the expression parser, the Python dataflow network, and the host interface. Using this framework, we provide three execution strategies, each with different data movement and kernel invocation characteristics. The interaction between the architecture components is shown in Figure 1. The host interface accepts user-generated expressions and incorporates them into the Python dataflow network by parsing them with a PLY-based parser. The dataflow network is executed using one or more OpenCL kernels, as determined by the execution strategy. The resulting derived field is returned via the host interface. Below, we describe each of these components in greater detail.

A. Expression Parser

At the front-end of our framework is a PLY-based parser [9], which takes a user defined input expression and produces a dataflow network specification. An expression is a statement, or set of statements, that compose a more complex calculation. The parser constructs a parse tree using the rules specified by a limited expression grammar. Statements can either be "simple", (i.e. consisting of a constant value, a variable, or a single filter invocation along with a set of inputs), or they can be "nested", (i.e. filter invocations calls with sub-expressions as arguments).

In the resulting parse tree, the root of each sub-tree corresponds to an assignment statement or a filter invocation. Assignment statements are used to associate a specific name with the filter invocation defined in its child node. For filter invocation sub-trees, child nodes are the filter's inputs, each of which may be one of two types. In the first case, the child node is a leaf and is therefore a final type, such as a constant value or an identifier mapping to the output of another filter. In the second case, the child node is the root of another sub-tree and is therefore defines a nested filter invocation.

We traverse the parse tree to generate a dataflow network specification. Filter invocations are given a generic name when encountered. Assignment statements map generic names to those provided by user. Throughout the traversal, each filter invocation, with the the names of its immediate inputs, is added to a Python list.

Basic operations, such as binary math operations are translated into the equivalent dataflow filter names as required to generate the network specification. For some expressions, accessing the components of a multi-dimensional variable is required (see vorticity magnitude and Q-criterion in Section IV). To capture this, the parser supports a bracket syntax similar to C/C++ arrays, which it translates into a "decompose" filter in the dataflow network specification.

Using the list of all filter invocations, common constants are reduced to single instances of source filters. We also use a limited common sub-expression elimination strategy to avoid computing unnecessary intermediate results. After these transformations, the list of filter invocations is fed into the Python dataflow network.

B. Python Dataflow Network

Dataflow networks create "pipelines" made up of "sources", "sinks" and "filters" to carry out a desired operation [4], [22], [37]. Our framework employs this design to create user defined derived fields. Filters correspond to the primitives that generate new fields, while sources and sinks get data in to and out of the target device (i.e. host-to-device and device-to-host memory exchanges). The dataflow network approach is very flexible, as its modules can be dynamically connected in unforeseen ways by the user. Our Python-based approach enhances the flexibility of this design by enabling multiple execution strategies; this aspect is described in Section III-C.

Here, we discuss how the user can define a network, how that network is constructed and initialized, and what filters are supported within our dataflow module. In our discussion, we will make a logical distinction between the OpenCL "host" and target "device": *host* will refer to system CPU and memory; *device* will refer to the component executing OpenCL kernels and its associated memory, whether the component is a GPU or the system CPU.

1) Network Definition: Our system provides a network definition API that reflects the "create and connect" modality of the dataflow paradigm. Our front-end parser uses this API to construct a dataflow network specification that realizes the user's expression. The process optionally creates a Python script that outlines all API calls, which can be inspected by the user. The API can also be used directly from Python, by a user or by a host application.

2) Network Initialization: Executing a dataflow network requires understanding the dependencies between filters. Our dataflow network module uses a topological sort to ensure proper precedence. It provides reference counting and reuses intermediate results multiple times to avoid unnecessary computation and reduce memory overhead. NumPy arrays are the primary data representation used in PyOpenCL and thus the form of input/output data adopted. PyOpenCL is used to transfer NumPy arrays between the OpenCL host and target device.

3) Supported Primitives: To support the dataflow construct, we implemented a set of basic primitives that act as flexible building blocks that can be combined to express more complex calculations. These building blocks are small OpenCL source functions that are written once and shared by all execution strategies. Each function contains minimal metadata to describe global memory requirements and the return type. The subset of operations necessary to support the expressions explored in this paper include: addition, subtraction, multiplication, square root, vector decomposition, and a 3D rectilinear mesh field gradient. This subset of operations is sufficient to demonstrate a capability that supports complex derived fields.

C. Execution Strategies

Our Python-based framework is able to support different execution strategies that control data movement and how the OpenCL kernels for each of the derived field primitives are composed to compute the final result. For this study we implemented three strategies: *roundtrip*, *staged* and *fusion*, which are described below. Our system could easily be extended to generate other execution strategies as well. This extension would involve modifying only the Python-based transformations – the OpenCL kernels for each primitive would not need to be modified.

1) Roundtrip: This execution strategy dispatches a distinct OpenCL kernel for each derived field primitive used in the user's expression. Further, at the end of each operation, the resulting array is transferred back to the host. From a performance perspective, this is a poor choice, since the back-and-forth traffic is often unnecessary. The advantage of this strategy is that it can utilize the host memory to hold intermediate results, allowing it to compute derived quantities

that may be constrained by faster strategies that require more global memory on a target device. Explicitly, if a derived field requires inputs F_1, F_2, ... F_n, then the target device may not be able to store all of the F_i's (see Figure 2 for an example).

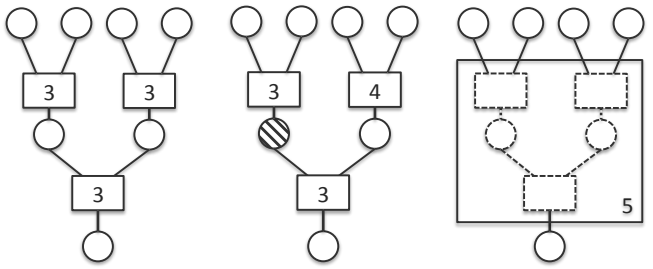
2) Staged: For this execution strategy, like the *roundtrip* strategy, one kernel is executed for each derived field primitive used in the user's expression. It contrasts with *roundtrip* in that data holding intermediate results is not transferred between host and target device. Instead, the data is staged in the device's global memory between kernel invocations.

3) Fusion: For this execution strategy, a dynamic kernel generator employs kernel fusion to construct and execute a single OpenCL kernel that implements all of the operations. This execution strategy contrasts with *staged* in that it uses a single kernel invocation and the fused kernel stores the intermediate results computed using the derived field primitives in local device registers. This minimizes the number of accesses to global memory, as long as the generated kernel program can fit on the device and avoid spilling results intended for local registers into the global memory.

The dynamic kernel generator used by this strategy is flexible enough to incorporate derived field primitives for simple one-line math functions, as well as more complex multi-line operations, e.g., the 3D rectilinear mesh field gradient requires over 50 lines of OpenCL source code. In addition, to efficiently support complex dataflow networks, the generator provides the following features:

- Per-element function calls for simple primitives (e.g. add, subtract, etc).
- Direct access to device global memory arrays for operations with more complex memory requirements (e.g. gradient).
- Source-code level insertion of constants.
- Operations that return multiple values per element are represented using built-in OpenCL vector types (e.g. float2, float4, ...).
- Source-code level implementation of array-decompose operations to select the proper OpenCL vector subcomponent (e.g. val.s0, val.s1).

Although the *fusion* strategy appears at first glance to be a clear-cut winner, the path to many-core success will require exploring many different strategies. Especially in an *in situ* setting, visualization applications will be subject to practices that are optimal for the simulation code, possibly preventing strategies such as creating new data decompositions or streaming data. While *roundtrip* in particular would appear to be a poor use of resources (because of its excessive memory bandwidth requirements), it is able to use CPU memory to store intermediate results, allowing it to consider data sets bigger than *fusion*. Figure 2 gives an example showing the different memory constraints required to execute the same dataflow using our three strategies. This example shows the constraints depend on the relationships of the filters in the dataflow network. Our system was designed with such subtleties in mind, allowing primitives to be written one time and then used as part of multiple execution strategies as required by target device constraints.



(a) *Roundtrip* (b) *Staged* (c) *Fusion*

Fig. 2. This figure presents a simple example dataflow network and shows the varying global device memory constraints required to execute this network with each of our three strategies. The circles indicate data arrays and the boxes indicate filters, annotated with the amount of memory required to execute. *Roundtrip* (left) is the least constrained, at *3* problem sized arrays, because it stores all intermediate results in the host memory. *Staged* (center) requires *4* arrays due to the need to keep an intermediate data array in device memory while the second filter is executing. *Fusion* (right) requires *5* arrays, since all filters are combined into a single kernel.

D. Host Interface

The parser front-end and dataflow network are implemented as Python modules and ran using Python from within a host application. This allows the framework to efficiently operate on existing data arrays *in situ*. The host application provides both the user's expression and NumPy objects for the input data arrays. Our framework processes the expression, executes the operations, and returns the resulting data array with the field representing the user's expression. Due to their wide use and the integration into PyOpenCL, NumPy arrays are the input/output data interface adopted for our framework.

Our framework executes in a standard Python interpreter with dependent modules (PLY, NumPy, PyOpenCL) installed. As stated above, the module can be used *in situ* for codes that provide a NumPy interface to their mesh data fields. For this paper, we use VisIt as an example host application. To call our framework from within VisIt, we wrote a custom VisIt Python Expression. This capability allowed us to create a Python filter that processes Python-wrapped instances of VTK data sets from a VisIt pipeline to create a new mesh field. For input data, VTK Python wrappers provide access to NumPy objects for existing VTK data arrays. These NumPy objects provide an efficient way to access existing raw data arrays and are compatible with PyOpenCL. The VisIt engine executes Python Expressions using a Python interpreter per MPI task.

Once the pipeline is constructed and our framework computes the user's expression, each subsequent rendering step reuses the resulting mesh. The pipeline is executed only once per time step for all rendering operations, such as changing the viewpoint, and it is executed again if the data set changes, such as when a different time step is loaded.

IV. Evaluation Methodology

In this section, we describe the methodology used to evaluate our framework and execution strategies. We first present

the application expressions and the simulation data sets used in our evaluation. We then describe the HPC cluster where we ran our experiments and the three studies we performed.

A. Application Expressions

To test our framework, we selected three derived quantities that are useful in vortex detection and analysis. Detecting vortices in complex flow fields is a problem of interest for many scientific applications. For a comprehensive overview of existing vortex detection algorithms see [21]. These expressions represent a range of computational complexity and memory usage, from the near-trivial vector magnitude to the expensive Q-criterion.

The first expression is *vector magnitude*, which is a common measure of the flow intensity across a vector field. We apply this expression to both velocity vectors and vorticity vectors. The input expression for velocity magnitude is shown in Figure 3A.

The second expression is *vorticity magnitude*, which is a measure of local spin in the vector field and is often used as a simple vortex detection method. The vorticity vector is the curl of the velocity vector and can be computed as follows:

$$\boldsymbol{\omega} = \nabla \times \boldsymbol{v} = \left(\frac{\partial w}{\partial y} - \frac{\partial v}{\partial z} \quad \frac{\partial u}{\partial z} - \frac{\partial w}{\partial x} \quad \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} \right) \tag{1}$$

The input expression for vorticity magnitude is shown in Figure 3B.

The third expression is *Q-criterion*, developed by Hunt et al. [19]. It is based on the observation that, in regions where $Q = \frac{1}{2}\left(||\Omega||^2 - ||S||^2\right) > 0$, rotation exceeds strain and, in conjunction with a pressure minimum, indicates the presence of a vortex. S is the symmetric rate of strain tensor and Ω is the antisymmetric rate of rotation tensor, which are defined in terms of the velocity gradient tensor $J = \nabla \boldsymbol{v}$ as:

$$S = \frac{1}{2}\left(J + J^T\right), \quad \Omega = \frac{1}{2}\left(J - J^T\right) \tag{2}$$

Note that $||A||$ is the Frobenius (matrix) norm defined as:

$$||A|| = \sqrt{Tr(AA^T)} = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2} \tag{3}$$

See Figure 3C for the input expression for Q-criterion. The corresponding dataflow network specification for the Q-criterion is shown in Figure 4. This is the most complex expression used in our evaluation.

B. Data Sets

To test the performance of our execution strategies on these expressions, we use sub-grids of a single time step of a 3072^3 DNS Raleigh-Taylor (RT) instability simulation run from Lawrence Livermore National Laboratory (LLNL) [12]. This large-scale simulation contains vortical features and the simulation data files provide a velocity vector field that can be used as input to our application expressions. To study single device performance as data per device grows, we selected twelve sub-grids of the RT data set that vary from 9.4 to 113.3 million cells. Each sub-grid contains cell-centered values

A: Velocity Magnitude

```
v_mag = sqrt(u*u + v*v + w*w)
```

B: Vorticity Magnitude

```
du = grad3d(u,dims,x,y,z)
dv = grad3d(v,dims,x,y,z)
dw = grad3d(w,dims,x,y,z)
w_x = dw[1] - dv[2]
w_y = du[2] - dw[0]
w_z = dv[0] - du[1]
w_mag = sqrt(w_x*w_x + w_y*w_y + w_z*w_z)
```

C: Q-criterion

```
du = grad3d(u,dims,x,y,z)
dv = grad3d(v,dims,x,y,z)
dw = grad3d(w,dims,x,y,z)
s_1 = 0.5 * (du[1] + dv[0])
s_2 = 0.5 * (du[2] + dw[0])
s_3 = 0.5 * (dv[0] + du[1])
s_5 = 0.5 * (dv[2] + dw[1])
s_6 = 0.5 * (dw[0] + du[2])
s_7 = 0.5 * (dw[1] + dv[2])
w_1 = 0.5 * (du[1] - dv[0])
w_2 = 0.5 * (du[2] - dw[0])
w_3 = 0.5 * (dv[0] - du[1])
w_5 = 0.5 * (dv[2] - dw[1])
w_6 = 0.5 * (dw[0] - du[2])
w_7 = 0.5 * (dw[1] - dv[2])
s_norm = du[0]*du[0] + s_1*s_1 + s_2*s_2 +
         s_3*s_3 + dv[1]*dv[1] + s_5*s_5 +
         s_6*s_6 + s_7*s_7 + dw[2]*dw[2]
w_norm = w_1*w_1 + w_2*w_2 + w_3*w_3 +
         w_5*w_5 + w_6*w_6 + w_7*w_7
q_crit = 0.5 * (w_norm - s_norm)
```

Fig. 3. Expressions for vortex detection algorithms used in our tests: (A) velocity magnitude, (B) vorticity magnitude and (C) Q-criterion.

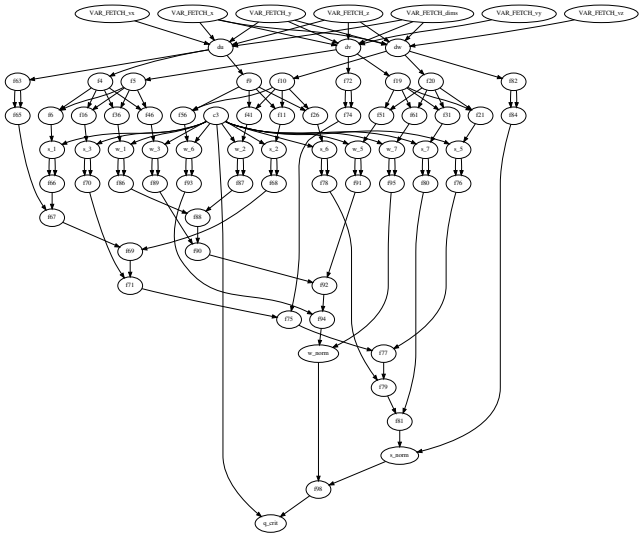


Fig. 4. An illustration of the dataflow network corresponding to the Q-criterion input expression.

Sub-grid Dimensions	# of Cells	Data Size
$192 \times 192 \times 0256$	9,437,184	218 MB
$192 \times 192 \times 0512$	18,874,368	435 MB
$192 \times 192 \times 0768$	28,311,552	652 MB
$192 \times 192 \times 1024$	37,748,736	869 MB
$192 \times 192 \times 1280$	47,185,920	1.1 GB
$192 \times 192 \times 1536$	56,623,104	1.3 GB
$192 \times 192 \times 1792$	66,060,288	1.5 GB
$192 \times 192 \times 2048$	75,497,472	1.7 GB
$192 \times 192 \times 2304$	84,934,656	2.0 GB
$192 \times 192 \times 2560$	94,371,840	2.2 GB
$192 \times 192 \times 2816$	103,809,024	2.4 GB
$192 \times 192 \times 3072$	113,246,208	2.6 GB

TABLE I
SUB-GRIDS OF 3072^3 RT SIMULATION TIME STEP USED FOR SINGLE DEVICE EVALUATION.

for the velocity vector (u, v, w) and point coordinate values (x, y, z). Table I provides a summary of the data sets used for our single device evaluation runs.

For distributed-memory parallel evaluation, we use the full 27 billion cell solution from the selected time step of the RT simulation. The 3072^3 rectilinear mesh contains the same cell and point field arrays and is decomposed into 3072 smaller sub-grids of size $192 \times 192 \times 256$. From this data set, the velocity magnitude calculation requires 3 input field arrays (u, v, w) and 1 output array. For the vorticity magnitude and Q-criterion calculations, an additional 3 input field arrays (x, y, z) are also required.

C. Test Environment

We chose LLNL's Edge GPU cluster to evaluate our framework. Edge is located in Livermore Computing's Open Computing Facility (LC-OCF). The system is a 216 node Linux cluster with each node containing two 2.8 GHz six-core Intel X5660 "Westmere" processors, 96 GB RAM, and two NVIDIA Tesla M2050 GPUs with 3 GB GDDR5 each. The GPUs each have a dedicated x16 PCIe gen 2 slot on the node motherboard, and the nodes are connected via a Mellanox QDR InfiniBand interconnect.

Edge's batch nodes support both Intel and NVIDIA OpenCL runtime platforms. This is an important feature for our evaluation, allowing us to test two different target architectures: the Intel CPUs and the NVIDIA Tesla M2050 GPUs. Both runtime platforms support the OpenCL 1.1 specification and the NVIDIA OpenCL platform leverages the CUDA 4.2 toolkit.

D. Evaluation Studies

We conducted three studies to evaluate our framework. The first was a study investigating runtime performance. The second was an evaluation of the memory usage of our strategies during execution. The third was a demonstration of integrating our framework into a larger pipeline, executing in a distributed-memory parallel context. For all of our studies we used VisIt as a host application for our framework. VisIt handles reading the data sets from disk, passing expression definitions and mesh data fields to our framework via the host interface, and rendering the derived field result returned by our framework.

1) Single Device Runtime Performance: The first component of our evaluation is a timing study of our execution strategies applied to the three selected application expressions. We test the single device runtime performance of both the Intel CPU and the NVIDIA M2050 GPU. Test cases probe the effects of increasing data size using the twelve sub-grids of an RT simulation time step, as outlined in Table I. The purpose of this study is to explore the runtime performance of our execution strategies.

To conduct this study, a device event timing infrastructure is necessary. Our framework provides an OpenCL environment interface built on top of PyOpenCL that records and categorizes timing events. The timings results in our performance study were obtained using the standard OpenCL device profiling API. Timings include all host-to-device transfers (transfers of input data), kernel executions, and device-to-host transfers (transfers of output data). For each test case we ran seven identical tests, removed the fastest and slowest results, and averaged the remaining five runtimes.

For our runtime performance study, we also compared our *roundtrip*, *staged* and *fusion* execution strategies to reference OpenCL kernels written for each of the three vortex detection expressions. The reference kernels have the same input and output global device memory constraints as our *fusion* strategy. They were written to directly compute the desired expression and hence are able to execute the expressions using less memory fetches and floating point operations than our strategies. The reference kernels are also executed using VisIt as a host application, and results were recorded using the same timing measurement methodology.

2) Single Device Memory Usage: The second component of our evaluation is a memory-centric study that explores the maximum amount of global device memory allocated to OpenCL buffers during the execution of the test cases used in the runtime performance study. The purpose of this study is to identify the memory constraints of our execution strategies on each target device as data sizes grow. The results of this study will show the memory constraints of our strategies and will be used to guide the development of future streaming strategies.

This study requires accurate recording of the global memory usage of the target device. This is also provided by our framework's OpenCL environment interface. In addition to recording timing events, the interface manages requests for device buffers. The amount of memory reserved for each device buffer is tracked. This information is used to calculate the amount of global device memory available at any time, as well as the high-water mark of device memory allocated.

3) Distributed-Memory Parallel Evaluation: The final component of our evaluation is a test of our framework processing a large data set on a HPC cluster in a realistic distributed-memory parallel context. The goal is to use multiple cluster nodes and multiple OpenCL target devices per node.

For this test, we selected the Q-criterion expression, the most complex of our test expressions, and the *fusion* execution

strategy. For test data, we use the full time step of the RT simulation described in Section IV-B. Using the original decomposition, the 27 billion cells of this 3072^3 rectilinear data set are grouped into 3072 sub-grids of size $192 \times 192 \times 256$.

For this test, we again run our framework *in situ* using VisIt as the host application. Although our kernel calculations are all embarrassingly parallel, we felt that evaluating our framework's use in the context of the larger HPC setting was important to demonstrate the validity of the design. To show integration in this context and to accurately compute the correct Q-criterion across the entire mesh, our framework explicitly requests ghost data generation from VisIt. To fulfill this request for our framework, VisIt will duplicate and exchange a stencil of cells around each sub-grid (i.e. "ghost data"). The data passed to our framework will be the sub-grids with these ghost cells, allowing the gradient primitives to compute the proper values on the boundaries of all sub-grids.

V. Evaluation Results

This section outlines the results from the three studies of our evaluation and provides a discussion of the implications of these results.

A. Single Device Runtime Performance

The first component of our evaluation studied the runtime performance of executing our three test expressions on data sets of increasing size using a single OpenCL target device. The study evaluated our three execution strategies and the OpenCL reference kernel, executing on both the Intel CPU and NVIDIA M2050 GPU.

The runtime performance results were acquired from these runs as outlined in Section IV-D. The results for each of the three test expressions are shown in the three graphs in Figure 5. In these graphs, the x-axes show the sizes of data sets used for each test case. The y-axes report the device execution runtimes in seconds, which include all host-to-device data transfers, kernel executions, and device-to-host data transfers. The CPU results are indicated by the blue series and GPU results by the red series. The gray series identifies test cases where the GPU failed. Results for each test case of the three execution strategies and the OpenCL reference kernel are indicted by a unique symbol.

As expected, increased computational complexity and data set sizes yield higher runtimes. This trend reflects both the increased data transfer and kernel execution times required to process larger data sizes. The CPU completed all test cases, while the GPU was able to complete 106 of the 144 test cases (73%). A discussion of the GPU test cases failures is provided in Section V-D. The GPU ran faster or on-par with the CPU for all test cases that the GPU executed successfully.

The *fusion* strategy yielded the best runtime performance, followed by the *staged* and finally the *roundtrip* strategy. The runtime study results confirm that the *fusion* strategy is competitive with the OpenCL reference kernel. For the vorticity magnitude and Q-criterion expressions, the *fusion* strategy and the OpenCL reference kernel transfer the same amount of data to and from the target device. Differences in runtimes of the two are a direct indication of the increased amount of computational work between the vorticity magnitude and Q-criterion expressions.

Expression	Strategy	Dev-W	Dev-R	K-Exe
VelMag	*Roundtrip*	11	6	6
	Staged	3	1	6
	Fusion	3	1	1
VortMag	*Roundtrip*	32	12	12
	Staged	7	1	18
	Fusion	7	1	1
Q-Crit	*Roundtrip*	123	57	57
	Staged	7	1	67
	Fusion	7	1	1

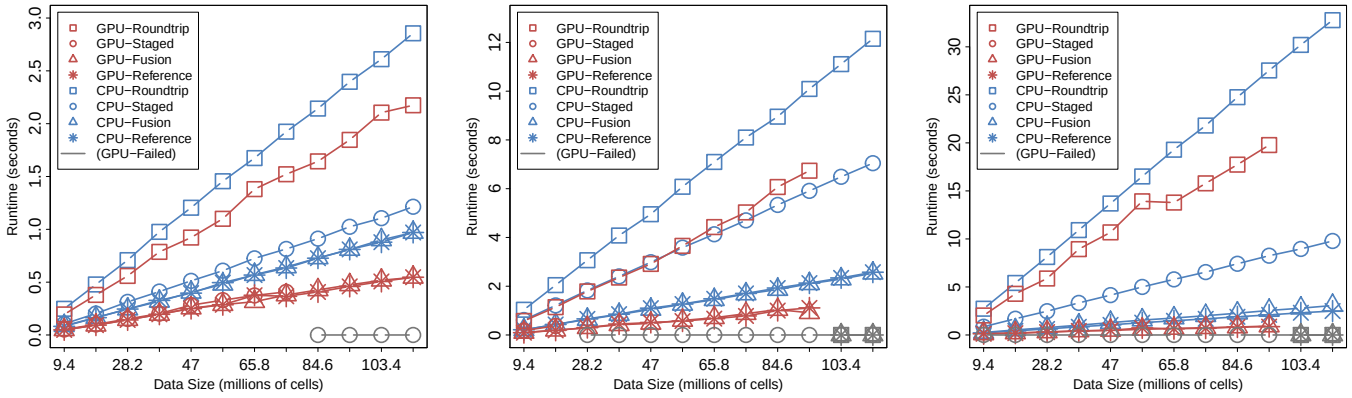
TABLE II
NUMBER OF HOST-TO-DEVICE TRANSFERS (*Dev-W*), DEVICE-TO-HOST TRANSFERS (*Dev-R*), AND KERNEL EXECUTIONS (*K-Exe*) FOR OUR TEST EXPRESSIONS DERIVING VELOCITY MAGNITUDE (*VelMag*), VORTICITY MAGNITUDE (*VortMag*), AND Q-CRITERION (*Q-Crit*) USING OUR EXECUTION STRATEGIES.

Through our OpenCL environment interface, we have access to the number of occurrences of each type of device event. Table II shows a break down of the device events recorded during the execution of the test expressions using our three strategies. The results in this table match the expected data movement and kernel dispatch characteristics of our strategies. For all expressions, *roundtrip* used the most host-to-device and device-to-host data transfers. The amount of host-device data transfers was equal for *staged* and *fusion*. For the vorticity magnitude and Q-criterion test cases, *staged* used more kernel dispatches than *roundtrip*, because it implements the decomposition primitive using a kernel to move intermediate results on the OpenCL target device. In all cases, *fusion* used the lowest total number of target device requests.

B. Single Device Memory Usage

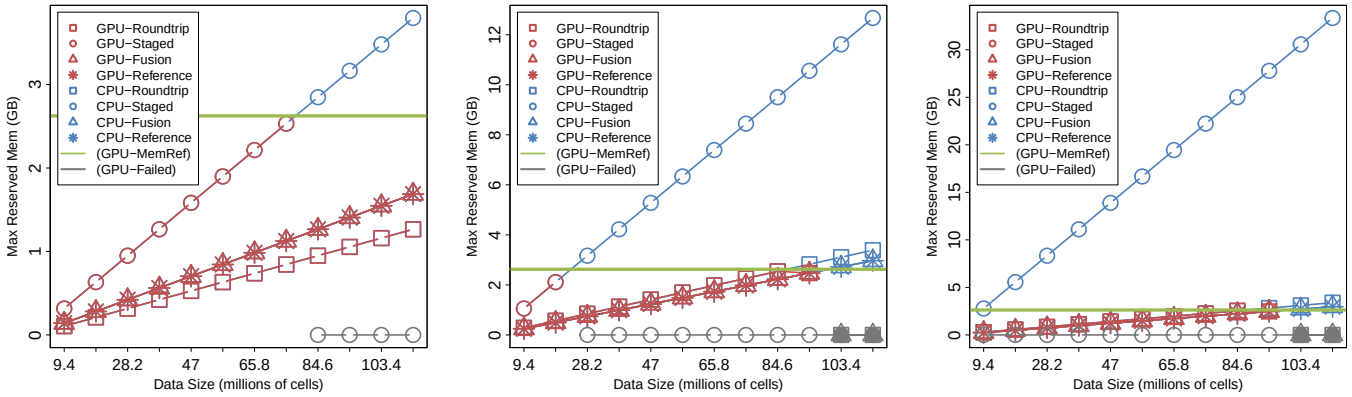
The second component of our evaluation focused on the device memory usage of the same runs used to measure runtime performance in Section V-A. The memory levels were acquired as outlined in Section IV-D. The memory usage results for each of the three test expressions are shown in the three graphs in Figure 6. In these graphs, the x-axes again show the sizes of data sets used for each test case. The y-axes report the maximum amount of global device memory reserved for OpenCL buffers during execution. The same color scheme is used: CPU results are indicated by the blue series and GPU results by the red series. The gray series identifies test cases where the GPU failed. For the successful GPU test cases, the GPU results are identical to the CPU results, and the red series overlays the blue. Thus, the blue series is shown only where the GPU test cases failed. Results for each test case of the three execution strategies and the OpenCL reference kernel are indicted by a unique symbol.

As expected, the reserved memory grows linearly as the input data size grows and the memory usage of our three strategies show unique slopes. *staged* required the most memory,



(a) Velocity Magnitude (b) Vorticity Magnitude (c) Q-criterion

Fig. 5. Runtime performance for single device execution of our three test expressions on two OpenCL target devices: an Intel Xeon CPU and a NVIDIA M2050 Tesla GPU. The blue series provides the timing results from the CPU tests and red series provides results from the GPU tests. The gray series represents GPU test cases that failed to run due to memory constraints. Our three execution strategies and the OpenCL reference kernel are represented with unique symbols.



(a) Velocity Magnitude (b) Vorticity Magnitude (c) Q-criterion

Fig. 6. Memory evaluation results for single device execution of our three test expressions on two OpenCL target devices: an Intel Xeon CPU and a NVIDIA M2050 Tesla GPU. The blue series provides the measured memory high-water mark from the CPU tests and the red series provides results from the GPU tests. The gray series represents GPU test cases that failed to run due to memory constraints. The green line serves as a reference for the total global device memory available on the NVIDIA 2050 GPU. Our three execution strategies and the OpenCL reference kernel are represented with unique symbols.

because all intermediate results are kept in the global memory of the device. Due to the number of inputs, *roundtrip* used less memory for the velocity magnitude test cases than the other two strategies and the OpenCL reference kernel. For the vorticity magnitude and Q-criterion cases, *roundtrip* used more memory than *fusion* due to buffers required for constant values, which are compiled into the kernels generated as part of *fusion*. Both *fusion* and the OpenCL reference kernel showed the same the memory usage.

These results verify that memory constraints were the cause of the failed GPU test cases in our runtime performance study. The maximum available global device memory of the NVIDIA M2050 GPU is indicated on these graphs by a green line. When the memory requirements of an execution strategy or the OpenCL reference kernel exceeded this line, the test case failed on the GPU. The CPU successfully ran all test cases. The CPU memory results provide a reference for the amount of memory required for a device to succeed on the test cases that the M2050 GPU failed on.

C. Distributed-Memory Parallel Test

The final component of our evaluation was to test our framework using a large data set on a HPC cluster in a realistic distributed-memory parallel context. We used our framework to successfully calculate the Q-criterion expression on the 27 billion cell data set with the *fusion* execution strategy. A pseudo-color rendering of the full Q-criterion result with an inset zoom of one sub-grid is shown in Figure 7. This test used 256 GPUs and 128 nodes of LLNL's Edge cluster. On each node, our framework used two GPUs in two independent MPI tasks. To obtain ghost data, our framework explicitly requested ghost stencil generation as part of the VisIt pipeline execution. Each GPU processed twelve sub-grids of the full data set, sized $192 \times 192 \times 256$ plus the necessary ghost cells.

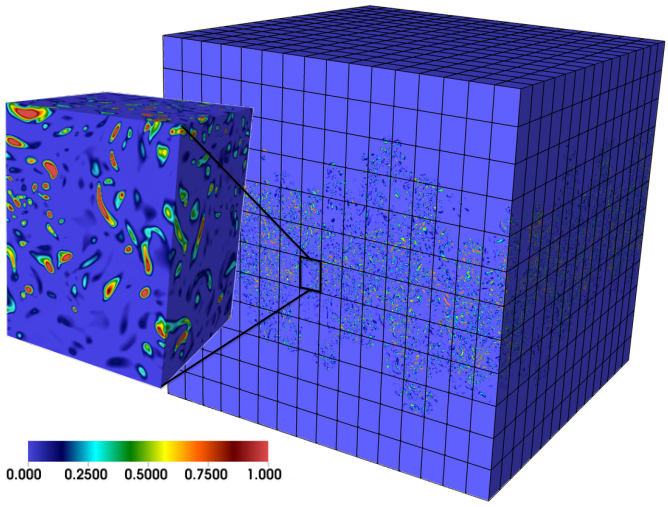


Fig. 7. This figure shows a successful rendering from the distributed-memory parallel test of our framework. For this test, the Q-criterion expression was executed using the *fusion* execution strategy on 256 GPUs using 128 nodes of LLNL's Edge cluster. The inset shows a zoomed-in rendering of one of the 3072 sub-grids, sized $192 \times 192 \times 256$, that make up the 3072^3, 27 billion cell rectilinear mesh. The full data set is shown with an outline of the sub-grid decomposition.

D. Discussion

Our evaluation confirmed the need for a flexible framework, capable of targeting different types of devices and selecting different execution strategies. The runtime performance study results show that in general the GPU runtimes were faster than the CPU. However, only the CPU was able to meet the global device memory requirements to run all test cases. The vorticity magnitude and Q-criterion results show test case instances where the GPU using *staged* failed to run, and the CPU using *staged* was faster than the available GPU *roundtrip* option. This result highlights the benefit of being able to select from multiple execution strategies and target devices with different hardware architectures.

The runtime performance and memory evaluation results highlight benefits and constraints of the two device architectures we tested. The GPU provides the best runtime performance for our expressions, assuming the data set can fit onto the device (or can be efficiently streamed to the device). The CPU is appropriate when large data sizes are expensive to decompose or difficult to stream to the GPU.

Our evaluation also provided insight into our three execution strategies. *Roundtrip* was the slowest of our three strategies. As expected, its runtime was dominated by host-to-device and device-to-host memory transfers. *Staged* performed much faster than *roundtrip*; however, it was the most constrained by device memory for our test expressions. *Fusion* was the fastest of our three strategies, and approached the speed of the reference OpenCL kernels. Using our framework, we are able to show that this execution strategy can provide efficiency that approaches a custom or one-off solution.

Each of the three strategies provide different benefits and constraints. *Fusion* is the fastest and preferred when possible. Though not demonstrated by the expressions tested in our study, there are cases where *staged* can be used, while memory constraints would prevent *fusion* from executing (recall the discussion of Figure 2). In these cases, *staged* would provide a performance boost over *roundtrip*, which has the least global device memory constraints, at the expense of the slowest runtime performance.

Our evaluation also showed that we can embed our framework in a host application and execute on a HPC cluster in a distributed-memory parallel context. The successful evaluation highlighted three important aspects not tested in our runtime performance test cases:

- The ability to use multiple OpenCL target devices per node.
- The ability to process multiple sub-grid chunks per target device.
- The ability to embed our framework into a larger analysis pipeline.

VI. Conclusion

In this paper, we presented a flexible Python-based framework for efficient derived field generation on many-core architectures using OpenCL. A key feature of our framework is the ability to support multiple execution strategies for composing operations using a common library of building blocks. We designed and tested three execution strategies: *roundtrip*, *staged* and *fusion*, using the vortex detection application to evaluate their runtime performance and memory constraints. We also demonstrated the use of our framework as part of a larger visualization and analysis pipeline embedded in VisIt using the Raleigh-Taylor instability simulation. Our evaluation confirmed the need for a flexible and efficient framework, capable of designing and testing different execution strategies on many-core architectures.

For future work, we plan to investigate the runtime performance of our execution strategies in a streaming context. We also plan to explore new execution strategies, including strategies that use multiple target devices on a single node. Finally, we plan to conduct a comprehensive performance study of our framework in a distributed-memory parallel setting.

Acknowledgments

We would like to thank Bill Cabot, Andy Cook, and Paul Miller at LLNL for access to the Rayleigh-Taylor DNS data set. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. This work was also supported in part by the U.S. National Science Foundation grant OCI-0906379. This work was also supported by the Director, Office of Advanced Scientific Computing Research, and Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] llvmpy: Python wrapper around the llvm c++ library. http://www.llvmpy.org/.

[2] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[3] *The OpenACC application programming interface.*, 2011. "http://www.openacc-standard.org/Downloads/OpenACC.1.0.pdf".

[4] G. Abram and L. A. Treinish. An extended data-flow architecture for data analysis and visualization. Research report RC 20001 (88338), IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, Feb. 1995.

[5] S. Ahern, A. Shoshani, K.-L. Ma, A. Choudhary, T. Critchlow, S. Klasky, V. Pascucci, J. Ahrens, E. W. Bethel, H. Childs, J. Huang, K. I. Joy, Q. Koziol, J. Lofstead, J. Meredith, K. Moreland, G. Ostrouchov, M. Papka, V. Vishwanath, M. Wold, N. Wright, and K. J. Wu. Scientific Discovery at the Exascale. Technical report, Report for the DOE ASCR 2011 Workshop on Exascale Data Management, Analysis, and Visualization, 2011.

[6] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison—Wesley, 1986.

[7] J. Ahrens, B. Geveci, and C. Law. Visualization in the paraview framework. In C. Hansen and C. Johnson, editors, *The Visualization Handbook*, pages 162–170, 2005.

[8] J. Aycock. Compiling little languages in Python: http://pages.cpsc.ucalgary.ca/ aycock/spark/.

[9] D. Beazley. PLY (Python Lex-Yacc) 3.4: http://www.dabeaz.com/ply/.

[10] S. Behnel, R. W. Bradshaw, and D. S. Seljebotn. Cython tutorial. In *Proceedings of the 8th Python in Science Conference*, pages 4 – 14, 2009.

[11] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: A CPU and GPU math compiler in Python. In *Proceedings of 9th Python in Science Conference (SCIPY)*. 2010.

[12] W. H. Cabot and A. W. Cook. Reynolds number effects on rayleightaylor instability with possible implications for type ia supernovae. 2006.

[13] H. Childs, E. Brugger, K. Bonnell, J. Meredith, M. Miller, B. Whitlock, and N. Max. A contract based system for large data visualization. In *VIS '05: Proceedings of the conference on Visualization '05*, 2005.

[14] Computational Engineering International, Inc. *EnSight User Manual*, 2009.

[15] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5:46–55, 1998.

[16] 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. *SC Conference*, 0:1–12, 2011.

[17] G. Ewing. Plex - a lexical analysis module for Python: http://www.cosc.canterbury.ac.nz/greg.ewing/python/plex/1.1.1/doc/.

[18] M. Fletcher. Simpleparse: http://simpleparse.sourceforge.net/.

[19] J. Hunt, A. Wray, and P. Moin. Eddies, Stream, and Convergence Zones in Turbulent Flows. Technical Report CTR-S88, Center for Turbulence Research, Stanford University, 1988.

[20] G. Inc. CLyther: A just-in-time specialization engine for opencl., 2010. http://srossross.github.com/Clyther/.

[21] M. Jiang, R. Machiraju, and D. S. Thompson. Detection and Visualization of Vortices. In *Visualization Handbook*, pages 287–301. Academic Press, 2005.

[22] C. Johnson, S. Parker, and D. Weinstein. Large-scale computational science applications using the SCIRun problem solving environment. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, 2000.

[23] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001. http://www.scipy.org/.

[24] 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.

[25] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, pages 75–, 2004.

[26] L. Lo, C. Sewell, and J. Ahrens. PISTON: A portable cross-platform framework for data-parallel visualization operators. In *Eurographics Symposium on Parallel Graphics and Visualization*, 2012.

[27] P. McCormick, J. Inman, J. Ahrens, J. Mohd-Yusof, G. Roth, and S. Cummins. Scout: a data-parallel programming language for graphics processors. *Parallel Computing*, 33(1011):648 – 662, 2007.

[28] P. McGuire. Introduction to pyparsing: An object-oriented easy-to-use toolkit for building recursive descent parsers. PyCon, 2006.

[29] J. S. Meredith, R. Sisneros, D. Pugmire, and S. Ahern. A distributed data-parallel framework for analysis and visualization algorithm development. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, pages 11–19, 2012.

[30] K. Moreland, U. Ayachit, B. Geveci, and K.-L. Ma. Dax toolkit: A proposed framework for data analysis and visualization at extreme scale. In *IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, 2011.

[31] T. Oliphant et al. Numba: Numpy aware dynamic compiler for Python, 2012. http://numba.pydata.org/.

[32] T. E. Oliphant. *Guide to NumPy*. Provo, UT, Mar. 2006. http://www.tramy.us/.

[33] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[34] A. Patel. Yapps: http://theory.stanford.edu/ amitp/yapps/.

[35] C. Pheatt. Intel threading building blocks. *J. Comput. Sci. Coll.*, 23(4):298–298, 2008.

[36] A. Rigo and S. Pedroni. JIT compiler architecture. Technical Report D08.2, 2007. http://codespeak.net/pypy/dist/pypy/doc/index-report.html.

[37] W. J. Schroeder, K. M. Martin, and W. E. Lorensen. The design and implementation of an object-oriented toolkit for 3d graphics and visualization. In *Proceedings of the 7th conference on Visualization '96*, VIS '96, pages 93–ff., Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[38] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering*, 12(3):66–73, 2010.

[39] H. Vo, D. Osmari, J. Comba, P. Lindstrom, and C. Silva. Hyperflow: A heterogeneous dataflow architecture. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, 2012.