Why Python as Programmer Frontend?

The TIOBE Programming Community Index is an indicator of the popularity of programming languages. The index is updated once a month. The ratings are based on the number of skilled engineers world-wide, courses and third party vendors. Popular search engines such as Google, Bing, Yahoo, Wikipedia, Amazon, YouTube and Baidu are used to calculate the ratings. It is important to note that the TIOBE index is not about the best programming language or the language in which most lines of code have been written.

The index can be used to check whether your programming skills are still up to date or to make a strategic decision about what programming language should be adopted when starting to build a new software system. The definition of the TIOBE index can be found here.

<table>
<thead>
<tr>
<th>Jun 2023</th>
<th>Jun 2022</th>
<th>Change</th>
<th>Programming Language</th>
<th>Ratings</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>Python</td>
<td>12.46%</td>
<td>+0.26%</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
<td>C</td>
<td>12.37%</td>
<td>+0.46%</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>↑</td>
<td>C++</td>
<td>11.36%</td>
<td>+1.73%</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>↓</td>
<td>Java</td>
<td>11.28%</td>
<td>+0.81%</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td></td>
<td>C#</td>
<td>6.71%</td>
<td>+0.59%</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td></td>
<td>Visual Basic</td>
<td>3.34%</td>
<td>-2.08%</td>
</tr>
</tbody>
</table>

Python is the lingua franca of computational, data sciences, and AI.

DaCe also supports C and Fortran but not as part of this talk 😊
It’s All About the Ecosystem – and NumPy (forget about lists & dictionaries if you’re after performance)
Scientific Python – a Long History of Optimization

NumPy is accelerated by numerous frameworks!

We need a fair comparison between those

Meet NPBench

51 kernels from 9 domains
- Learning (6)
- LinAlg (12)
- Chemistry (4)
- Signals (3)
- Physics (9)
- Graphs (2)
- Weather (2)
- Solver (10)
- Other (3)

Meet NPBench

Metrics
- Performance

Frameworks
- NumPy baseline
- Pythran
- Numba
- CuPy
- DaCe

Productivity

https://github.com/spcl/npbench
Machine with two 16-core Intel Xeon Gold 6130 processors and an Nvidia V100 GPU with 32GB of memory
How to address locality challenges on standard architectures and programming?

D. Unat et al.: “Trends in Data Locality Abstractions for HPC Systems”


Three Ls of modern computing:

**Spatial Locality**

**Temporal Locality**

**Control Locality**
Control in Load-store vs. Dataflow

Load-store ("von Neumann")

\[ x = a + b \]

Energy per instruction: 70pJ

Static Dataflow ("non von Neumann")

\[ y = (a + b) \times (c + d) \]

Energy per operation: 1-3pJ

Turing Award 1977 (Backus): "Surely there must be a less primitive way of making big changes in the store than pushing vast numbers of words back and forth through the von Neumann bottleneck."

Energy per operation: 1-3pJ
Single Instruction Multiple Data/Threads (SIMD - Vector CPU, SIMT - GPU)

High Performance Computing really became a data management challenge (which requires spatial programming / thinking)

[1]: Marc Horowitz, Computing’s Energy Problem (and what we can do about it), ISSC 2014, plenary
Data movement dominates everything!

- “In future microprocessors, the energy expended for data movement will have a critical effect on achievable performance.”
- “…movement consumes almost 58 watts with hardly any energy budget left for computation.”
- “…the cost of data movement starts to dominate.”
- “…data movement over these networks must be limited to conserve energy…”
- the phrase “data movement” appears 18 times on 11 pages (usually in concerning contexts)!
- “Efficient data orchestration will increasingly be critical, evolving to more efficient memory hierarchies and new types of interconnect tailored for locality and that depend on sophisticated software to place computation and data so as to minimize data movement.”
“Sophisticated software”: How do we program today?

- Well, to a good approximation how we programmed yesterday
  - Or last year?
  - Or four decades ago?

- Control-centric programming
  - Worry about operation counts (flop/s is the metric, isn’t it?)
  - Data movement is at best implicit (or invisible/ignored)
  - In fact, this is how we think about algorithms
    *Do this then that*

- How to cross the chasm between algorithms and data-centric computing?
  - Domain scientists and algorithm developers remain in the control-centric land
  - Performance engineers enter the data-centric/spatial land
  - An intermediate representation connects the two – both may need to adapt!
    *→ Performance Metaprogramming connects the two!*

Backus ’77: “The assignment statement is the von Neumann bottleneck of programming languages and keeps us thinking in word-at-a-time terms in much the same way the computer’s bottleneck does.”
Upleveling programming in the 21st century – Performance Metaprogramming

\[ \frac{\partial u}{\partial t} - \alpha \nabla^2 u = 0 \]

**Domain Scientist**

- translate DSL into an Internal Representation
- SDFG Builder API
- Multi-Level Library Nodes

**Applied Scientist**

- DSLs
  - NumPy
  - TensorFlow
  - PyTorch
  - MATLAB

**Performance Engineer**

- Graph Transformations (API, Interactive)
- Transformed Dataflow
- Performance Results
- 100s of reusable SLOC

**Specialized Code Generation**

- CPU Code
- GPU Code
- FPGA Code
- C++ code generation/runtime

**Runtime**

- 10s of SLOC
- 1000s of auto-generated SLOC

Ben-Nun, de Fine Licht, Ziogas, TH: Stateful Dataflow Multigraphs: A Data-Centric Model for High-Performance Parallel Programs, SC19
DaCe.Python translates to Parametric Dataflow Graphs (SDFGs) as IR

\[ y = x^2 + \sin(\pi x) \]

Tasklet

Memlets

Ben-Nun, de Fine Licht, Ziogas, TH: Stateful Dataflow Multigraphs: A Data-Centric Model for High-Performance Parallel Programs, SC19
Data-Parallelism in Parametric Dataflow Graphs (SDFGs)

Represent Parameters as Symbols!
Data-Parallelism in Parametric Dataflow Graphs (SDFGs)

Tasklet

A


B


Tasklet

Tasklet

…”

Parametric Iteration space (map)

A

A[0:N]

[i=0:N]

A[i]

Tasklet

B

B[i]

[i=0:N]

B[0:N]
Non-dataflow ordering: **States** in Parametric Dataflow Graphs (SDFGs)

Ben-Nun, de Fine Licht, Ziogas, TH: Stateful Dataflow Multigraphs: A Data-Centric Model for High-Performance Parallel Programs, SC19
Non-dataflow ordering: **States** in Parametric Dataflow Graphs (SDFGs)

Ben-Nun, de Fine Licht, Ziogas, TH: Stateful Dataflow Multigraphs: A Data-Centric Model for High-Performance Parallel Programs, SC19
### Parametric Dataflow Graphs - Concepts

#### Data Containers
- Store volatile (buffers, queues, RAM) and nonvolatile (files, I/O) information
- Can be sources or sinks of data

#### Computation
- Stateless functions that perform computations at any granularity
- Data access only through ports

#### Data Movement / Dependencies
- Data flowing between containers and tasklets/ports
- Implemented as access, copies, streaming, ...

#### Parallelism and States
- Map scopes provide parallelism
- States constrain parallelism outside of datatflow
Performance Engineer view: Visual Studio Code Integration & Transformations

Schaad et al.: Boosting Performance Optimization with Interactive Data Movement Visualization
Performance Engineer view: Analyzing Data Flows & Computation Volume
Performance Engineer view: Analyzing Memory Access Patterns & Locality
Programmer/Performance Engineer view: Debugging & Code Generation

Visual Studio Code integration

Plugin "DaCe SDFG Viewer (SDFV)"

Schaad et al.: Boosting Performance Optimization with Interactive Data Movement Visualization
DaCe Offers Full Support for ML Pipelines Through ONNX

Key principle: minimize data movement and optimize data layout

**MLSys’20**

Data Movement Is All You Need: A Case Study on Optimizing Transformers

Andrej Ivanov*, Nikalai Bykov*, Tal Ben-Nun, Shigang Li, Toreæ Hoefler
ETH Zurich
efficient@computer.org, ef@ethz.ch
* Equal contribution

Abstract—Transformers, have become widely used for language modeling and sequence learning tasks, and are one of the most important machine learning workhorses today. Training one is a very compute-intensive task, often taking days or weeks, and significant attention has been given to optimizing transformers for better performance and faster training. This work addresses the key bottleneck of memory access (data transfer) in ONNX, the most widely used open-source framework for deep learning. We propose a new solution, which we refer to as DaCe, that decouples data and model movement and optimization. DaCe employs a flow-based framework for data movement, and a unified ONNX-based approach to model movement and optimization.

Rausch et al. “A Data-Centric Optimization Framework for Machine Learning”, in ICS’22
Gordon Bell Prize 2019 on ORNL’s Summit (Top-1 Machine)

- **Gordon Bell Prize 2019**
  - Optimized twice-finalist code OMEN
  - Quantum Nano Transport simulation
    - *Design of future micro-processors*

- **Now working on large-scale:**
  - Deep Learning (transformers)
  - Climate (COSMO, icon, fv3)
  - Green’s functions solvers
  - ... your project?

[http://spcl.inf.ethz.ch/DAPP](http://spcl.inf.ethz.ch/DAPP)
Case Study: Weather and Climate Code FV3

[1] COSMO 1.1 km
2018-05-29 00:00 UTC+2

Weather service is currently GPU-powered [2]

[1] Institute for Atmospheric and Climate Science and Computer Graphics Laboratory, ETH Zürich [https://vimeo.com/389292423]
[2] Swiss National Supercomputing Center (CSCS) [https://www.cscs.ch/computers/arolla-tsa-meteoswiss]
The Pace Project – Rewriting FV3 in Python

- **FV3** is a highly optimized atmospheric model in Fortran
  - Rewrite in Python to run it at scale on modern supercomputers
  - No FORTRAN involved – move to 21st century programming + devops + package management (with similar syntax!)
- **Full dynamical core**: 12,450 Python LoC across 36 modules
  vs. 29,458 in the baseline FORTRAN implementation

Usage: python -m pace.driver.run [OPTIONS] CONFIG_PATH
Run the driver.
CONFIG_PATH is the path to a DriverConfig yaml file.
Options: ...

https://github.com/ai2cm/pace

Declarative Abstraction (GT4Py)
Unit Tests
Dynamical Core
Acoustics
Remapping
Horizontal Stencil
Vertical Solver

Orchestration (DaCe)
Full-Program Optimization
Transfer Tuning
Local Optimization
Backend
Halo Exchange
Callbacks

J. Dahm et al., “Pace v0.1: A Python-based Performance-Portable Implementation of the FV3 Dynamical Core”. EGUSphere’22
Declarative Abstraction (GT4Py)

Acoustics
Tracer Advection
Dynamical Core
Remapping
Unit Tests
Halo Exchange

Horizontal Stencil
Vertical Solver

Local Optimization
Transfer Tuning
Full-Program Optimization

Callbacks
Backend
Orchestration (DaCe)

Ben-Nun et al.: Productive Performance Engineering for Weather and Climate Modeling with Python, SC22
class HyperdiffusionDamping:
    # ...
    def __call__(self, qdel: FloatField, cd: float):
        # ...
        for n in range(self._ntimes):
            nt = self._ntimes - (n + 1)
            self._corner_fill(qdel, self._q)
            if nt > 0:
                self._copy_corners_x(self._q)
                self._compute_zonal_flux[n](
                    self._fx, self._q, self._del6_v)
            # ...

def dycore_loop(state, dycore, time_steps):
    # ...
    for _ in range(time_steps):
        dycore.step_dynamics(state)
        # ...
        state = initialize_state(...)  # Data loading
        dycore = fv_dynamics.DynamicalCore(...)  
        # Invoke function
        dycore_loop(state, dycore, T)
        validate(state)
        plot_on_map(state.x_wind)
**Dynamical Core (fv_dynamics)**

**Stencil calls per timestep:** 18,978

**Legend**

- **Component with Multiple Stencils**

---

```python
class HyperdiffusionDamping:
    # ...
    def __call__(self, qdel: FloatField, cd: float):
        # ...
        for n in range(self._ntimes):
            nt = self._ntimes - (n + 1)
            self.corner_fill(qdel, self._q)
            if nt > 0:
                self.copy_corners_x(self._q)
            self.compute_zonal_flux[n](self.fx, self._q, self._del6_v)
        # ...

def dycore_loop(state, dycore, time_steps):
    for _ in range(time_steps):
        dycore.step_dynamics(state)
    # ...

del2cubed.py

https://github.com/ai2cm/pace/blob/main/examples/notebooks/stencil_definition.ipynb

Ben-Nun et al.: Productive Performance Engineering for Weather and Climate Modeling with Python, SC22
Declarative Abstraction (GT4Py)

- Acoustics
- Remapping
- Unit Tests
- Dynamical Core
- Halo Exchange
- Horizontal Stencil
- Vertical Solver
- Callbacks
- Backend
- Local Optimization
- Transfer Tuning
- Full-Program Optimization
- Orchestration (DaCe)
GridTools for Python (GT4Py)

- Domain Specific Language (DSL) for Weather and Climate

- A declarative approach to define stencils (“what”, not “how”)
  - 3D stencils and vertical solvers

- Computation domain is abstracted
  - Relative indexing
  - Automatic iteration ranges and halo regions

- Implementation concerns are delegated to backends
  - Execution schedules
  - Memory allocation
  - Target language

```python
@gtscript.stencil(backend='dace:gpu')
def q_j_stencil(q: FloatField, area: FloatFieldIJ, x_area_flux: FloatField, fx2: FloatField, q_j: FloatField):
    with computation(PARALLEL), interval(...):
        fx1 = x_area_flux * fx2
        area_with_x_flux = area + x_area_flux - x_area_flux[1, 0, 0]
        q_j = (q * area + fx1 - fx1[1, 0, 0]) / area_with_x_flux
```

Ben-Nun et al.: Productive Performance Engineering for Weather and Climate Modeling with Python, SC22
```python
@gtscript.stencil(backend='dace:gpu')
def q_j_stencil(q: FloatField, area: FloatFieldIJ,  
x_area_flux: FloatField, fx2: FloatField,  
q_j: FloatField):
    with computation(PARALLEL), interval(...):
        fx1 = x_area_flux * fx2
        area_with_x_flux = area + x_area_flux - x_area_flux[1, 0, 0]
        q_j = (q * area + fx1 - fx1[1, 0, 0]) / area_with_x_flux
```

Stencil Implementations

GT4Py Backend
Stencil Implementations

Orchestration and Global Optimization

GT4Py Backend

Ben-Nun et al.: Productive Performance Engineering for Weather and Climate Modeling with Python, SC22
Characterizing the optimization space

**Within each stencil**
- Computational layout
- Data layout
- Other rescheduling passes in GT4Py (e.g., branch → predication)

**Between stencils**
- Fusion
- Macro scheduling
- Pre-allocation (memory pool, static)
- Data layout “path”

---

Ben-Nun et al.: Productive Performance Engineering for Weather and Climate Modeling with Python, SC22
Initial Heuristics

Interval, Operation, K, J, I

J, I, Interval, Operation, K
Initial Heuristics

Aligned addresses

Pre-padding \((o)\)

0

\(a\)

Shape: \((I + 2h, J + 2h, K)\)

Start offset: \(o = a - h\)

Strides:

\(s_i = 1\)

\(s_j = a \left\lceil \frac{I + 2h}{a} \right\rceil\)

\(s_k = s_j \cdot (J + 2h)\)
Initial Heuristics

Module-Based Autotuning

Transfer-Tune to Full Application

Exhaustive tuning on graph cutouts

Initial Heuristics

Module-Based Autotuning

Transfer-Tune to Full Application

Subgraph Fusion

On-The-Fly Fusion

Store top-k patterns

Exhaustive tuning on graph cutouts

Without transfer tuning:

\[ \geq 30,302,185 \text{ configurations} \]

With transfer tuning:

603

2:42 hours on Piz Daint

8:24 hours

Truemper et al.: Performance Embeddings: A Similarity-based Transfer Tuning Approach to Performance Optimization, ICS’23
Initial Heuristics

Module-Based Autotuning

Transfer-Tune to Full Application

Benchmark, Generate Perf. Model

Horizontal stencil

Vertical solver

Truemper et al.: Performance Embeddings: A Similarity-based Transfer Tuning Approach to Performance Optimization, ICS’23
Initial Heuristics - Module-Based Autotuning

Transfer-Tune to Full Application

Benchmark, Generate Perf. Model

Suboptimal Kernel Inspection

---

with computation(PARALLEL), interval(...):

    vort = dt * (delpc ** 2.0 + vort ** 2.0) ** 0.5

---

Truemper et al.: Performance Embeddings: A Similarity-based Transfer Tuning Approach to Performance Optimization, ICS'23
Initial Heuristics
Module-Based Autotuning
Transfer-Tune to Full Application
Benchmark, Generate Perf. Model

Fine Tuning
Suboptimal Kernel Inspection

Truemper et al.: Performance Embeddings: A Similarity-based Transfer Tuning Approach to Performance Optimization, ICS’23
Initial Heuristics

Module-Based Autotuning

Transfer-Tune to Full Application

Benchmark, Generate Perf. Model

Fine Tuning

Suboptimal Kernel Inspection

Watch the full “Transfer Tuning” talk tomorrow @ ICS’23, 4:20pm!

Truemper et al.: Performance Embeddings: A Similarity-based Transfer Tuning Approach to Performance Optimization, ICS’23
Evaluated Systems

Piz Daint:
- GPU: 1 x NVIDIA Tesla P100 / Node
- CPU: Intel Xeon E5-2690 v3 (12 cores)

JUWELS Booster:
- GPU: 4 x NVIDIA Tesla A100 / Node
- CPU: AMD EPYC 7402 (2 sockets, 24 cores)

Domain size: 192x192x80
Memory Bounds

Initial Heuristics → Module-Based Autotuning → Transfer-Tune to Full Application → Benchmark, Generate Perf. Model

Fine Tuning → Suboptimal Kernel Inspection

43.77 GB/s

501.1 GB/s

Potential Speedup ≤ 11.45x
Representative Vertical Solver
Riemann Solver (riem_solver_c)

Semi-implicit solver for nonhydrostatic terms of vertical velocity and pressure perturbation

<table>
<thead>
<tr>
<th>Domain Size (relative size)</th>
<th>FORTRAN Time [ms]</th>
<th>FORTRAN Scaling</th>
<th>GT4Py+DaCe Time [ms]</th>
<th>GT4Py+DaCe Scaling</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 × 128 × 80 (1x)</td>
<td>12.27</td>
<td>—</td>
<td>1.85</td>
<td>2.08</td>
<td>6.63×</td>
</tr>
<tr>
<td>192 × 192 × 80 (2.25x)</td>
<td>27.94</td>
<td>2.28</td>
<td>3.86</td>
<td>3.76</td>
<td>7.25×</td>
</tr>
<tr>
<td>256 × 256 × 80 (4x)</td>
<td>52.40</td>
<td>4.27</td>
<td>6.96</td>
<td>7.53×</td>
<td></td>
</tr>
<tr>
<td>384 × 384 × 80 (9x)</td>
<td>121.80</td>
<td>9.92</td>
<td>15.31</td>
<td>8.26</td>
<td>7.96×</td>
</tr>
</tbody>
</table>

- CPU cache runs out, data layout not ideal
- Not enough parallelism

Ben-Nun et al.: Productive Performance Engineering for Weather and Climate Modeling with Python, SC22
Representative Horizontal Stencil
Finite Volume Transport (fv_tp_2d)

FORTRAN runs on a single slice, GT4Py/DaCe runs on entire 3D domain

<table>
<thead>
<tr>
<th>Domain Size (relative size)</th>
<th>FORTRAN</th>
<th>GT4Py+DaCe</th>
</tr>
</thead>
<tbody>
<tr>
<td>128x128x80 (1x)</td>
<td>3.41 ms</td>
<td>1.81 ms</td>
</tr>
<tr>
<td>192x192x80 (2.25x)</td>
<td>12.31 ms</td>
<td>3.41 ms</td>
</tr>
<tr>
<td>256x256x80 (4x)</td>
<td>35.79 ms</td>
<td>5.67 ms</td>
</tr>
<tr>
<td>384x384x80 (9x)</td>
<td>106.66 ms</td>
<td>13.10 ms</td>
</tr>
</tbody>
</table>

0.13% of load/stores are L3 misses

Closing gap to ideal memory bandwidth factor

Ben-Nun et al.: Productive Performance Engineering for Weather and Climate Modeling with Python, SC22
Weak Scaling

Simulation throughput of **0.12 SYPD** at 2.6 km grid spacing

FV3 Summary:
- 6 weeks of work
- 10 optimization revisions
- 4 performance engineers
- 3.92 – 8.48x speedup vs. FORTRAN
- 0 model changes

Ben-Nun et al.: Productive Performance Engineering for Weather and Climate Modeling with Python, SC22
Key Points and Conclusions

- Scientific Python has a long history of optimization for high-performance computing, with contributions from various domains.
- Spatial Devices are the Future!
- More on Mapping with some Theory

Fortran to DaCe coming soon!

Spatial Devices are the Future!

More on Mapping with some Theory

Friday, 10am @ HPDC’23

De Matteis et al.: “Streaming Task Graph Scheduling for Dataflow Architectures”

More of SPCL’s research:

- youtube.com/@spcl
- twitter.com/spcl_eth
- github.com/spcl

Want to join our efforts? We’re looking for excellent Postdocs, PhD students, and Visitors. Talk to me!
Generating hardware descriptions

Stateful DataFlow multiGraph (SDFG)

Performance Engineer

Parametric Dataflow Graphs (SDFG)

Graph Transformations

Code generated for CPU, GPU, FPGA.

All data movement is explicit

Pure dataflow sections.
Example transformation #0: Offload to FPGA

Graph Transformations

```
fpga_x.CopyFromHost(0, N, x);
fpga_y.CopyFromHost(0, N, y);
fpga_w.CopyFromHost(0, N, w);
__dace_runkernel_axpydot(x_device, y_device, w_device, N);
fpga_result.CopyToHost(0, N, result);
```
Example transformation #1: Stream memory accesses

```c
void module_read_x(float const *x_device_in,
                    dace::FIFO<float, 1, P> &x_pipe,
                    int N) {
  for (int i = 0; i < N; i += 1) {
    #pragma HLS PIPELINE II=1
    float from_memory = x_device_in[i];
    ///////////////////
    // Tasklet code (read_x)
    x_pipe.push(from_memory);
    ///////////////////
  }
}
```

Memory accessed from separate processing elements
Example transformation #2: Stream between operators

Stateful DataFlow multiGraph (SDFG)

Graph Transformations

Performance Engineer

Parametric Dataflow Graphs (SDFG)

Code generated for CPU, GPU, FPGA.

for (int n0 = 0; n0 < (N / P); n0 += 1) {
    dace::vec<float, 4>* C_buffer[(M / 4)];
    for (int k = 0; k < K; k += 1) {
        float A_reg;
        for (int n1 = 0; n1 < P; n1 += 1) {
            #pragma HLS PIPELINE II=1
            #pragma HLS LOOP_FLATTEN
            float a_in = (A_pipe[p]).pop();
            // Tasklet code (buffer_a)
            if ((n1 == ((P - p) - 1))) {
                A_reg = a_in;
            } else {
                A_pipe[(p + 1)].push(a_in);
            }
        }
        for (int m = 0; m < (M / 4); m += 1) {
            #pragma HLS PIPELINE II=1
            #pragma HLS LOOP_FLATTEN
            float a_in = A_reg;
            dace::vec<float, 4>* b_in = (B_pipe[p]).pop();
            dace::vec<float, 4>* c_in = C_buffer[m];
            ...
        }
    }
}

Code generation

for (int n0 = 0; n0 < (N / P); n0 += 1) {
dace::vec<float, 4> C_buffer[(M / 4)];
for (int k = 0; k < K; k += 1) {
  float A_reg;
  for (int n1 = 0; n1 < P; n1 += 1) {
    #pragma HLS PIPELINE II=1
    #pragma HLS LOOP_FLATTEN
    
    float a_in = (A_pipe[p]).pop();

    // Tasklet code (buffer_a)
    if ((n1 == ((P - p) - 1))) {
      A_reg = a_in;
    }
    if ((p < (P - 1))) {
      A_pipe[(p + 1)].push(a_in);
    }
  }
  #pragma HLS PIPELINE II=1
  #pragma HLS LOOP_FLATTEN
  
  float a_in = A_reg;
  dace::vec<float, 4> b_in = (B_pipe[p]).pop();
  dace::vec<float, 4> c_in = C_buffer[m];
  ...
}

for (int n0 = 0; n0 < (N / P); n0 += 1) {
  float4 C_buffer[(M / 4)];
  #pragma ivdep
  for (int k = 0; k < K; k += 1) {
    float A_reg;
    #pragma ivdep
    
    float a_in = read_channel_intel(A_pipe[p]);
    float *a_reg = &A_reg;
    #define a_out A_pipe[(p + 1)] // God save us
    
    // Tasklet code (buffer_a)
    if ((n1 == ((P - p) - 1))) {
      *a_reg = a_in;
    }
    if ((p < (P - 1))) {
      write_channel_intel(a_out, a_in);
    }
  }
  #undef a_out
  
  #pragma loop_coalesce
  #pragma ivdep
  for (int m = 0; m < (M / 4); m += 1) {
    float4 b_in = read_channel_intel(B_pipe[p]);
    float4 c_in = C_buffer[m];
    ...
  }

Cross-vendor support

AMD

Vivado HLS

Intel

OpenCL

Cross-vendor support
Boilerplate code elimination

```cpp
int __dace_init_intel_fpga(float *__restrict__ A, dace::vec<float, 4> *__restrict__ B,
                          dace::vec<float, 4> *__restrict__ C, int K, int N) {
  dace::fpga::_context = new dace::fpga::Context();
  dace::fpga::_context->Get().MakeProgram(
    DACE_BINARY_DIR "gemm_fpga_systolic_vectorized_4_NxKx128.aocx");
  return 0;
}

void __dace_runkernel_gemm_0(hlslib::ocl::Buffer<float> &A_device,
                             hlslib::ocl::Buffer<dace::vec<float, 4>> &B_device,
                             hlslib::ocl::Buffer<dace::vec<float, 4>> &C_device,
                             int K, int N) {
  hlslib::ocl::Program program = dace::fpga::_context->Get().CurrentlyLoadedProgram();
  std::vector<hlslib::ocl::Kernel> kernels;
  kernels.emplace_back(program.MakeKernel("mod_0_read_A", A_device, K, N));
  kernels.emplace_back(program.MakeKernel("mod_0_read_B", B_device, K, N));
  kernels.emplace_back(program.MakeKernel("mod_0_gemm_0", K, N));
  kernels.emplace_back(program.MakeKernel("mod_0_gemm_1", K, N));
  kernels.emplace_back(program.MakeKernel("mod_0_gemm_2", K, N));
  kernels.emplace_back(program.MakeKernel("mod_0_gemm_3", K, N));
  kernels.emplace_back(program.MakeKernel("mod_0_write_C", C_device, K, N));
  std::vector<cl::Event> events;
  for (auto &k : kernels) {
    events.emplace_back(k.ExecuteTaskFork());
  }
  cl::Event::waitForEvents(events);
}
```
void __program_gemm_fpga_systolic_vectorized_4_NxKx128_internal(float *__restrict__ A,
    dace::vec<float, 4> *__restrict__ B,
    dace::vec<float, 4> *__restrict__ C, int K, int N) {
    hlslib::ocl::Buffer<float, hlslib::ocl::Access::readWrite> A_device =
        dace::fpga::_context->Get().MakeBuffer<float, hlslib::ocl::Access::readWrite>((K * N));
    hlslib::ocl::Buffer<dace::vec<float, 4>, hlslib::ocl::Access::readWrite> B_device =
        dace::fpga::_context->Get().MakeBuffer<dace::vec<float, 4>, hlslib::ocl::Access::readWrite>((K * M) / 4));
    hlslib::ocl::Buffer<dace::vec<float, 4>, hlslib::ocl::Access::readWrite> C_device =
        dace::fpga::_context->Get().MakeBuffer<dace::vec<float, 4>, hlslib::ocl::Access::readWrite>((M * N) / 4));
    A_device.CopyFromHost(0, N * K, A);
    B_device.CopyFromHost(0, K * (M / 4), B);
    C_device.CopyFromHost(0, N * (M / 4), C);
    __dace_runkernel_gemm_0(A_device, B_device, C_device, K, N);
    C_device.CopyToHost(0, N * (M / 4), C);
}
N = dace.symbol("N")
K = dace.symbol("K")
M = dace.symbol("M")
P = dace.symbol("P")
W = dace.symbol("W")

def make_sdfg():
    # ...do stuff...
    return sdfg

if __name__ == "__main__":
    P.set(8)  # 8 processing elements
    W.set(4)  # 4-way vectorization
    # Matrix of size 1024x1024x1024
    M.set(1024)
    K.set(1024)
    N.set(1024)

gemm = make_sdfg()

# Initialize arrays: Randomize A and B, zero C
A = np.ndarray([N.get(), K.get()], dtype=np.float32)
B = np.ndarray([K.get(), M.get()], dtype=np.float32)
C = np.ndarray([N.get(), M.get()], dtype=np.float32)
A[:] = np.random.rand(N.get(), K.get()).astype(np.float32)
B[:] = np.random.rand(K.get(), M.get()).astype(np.float32)
C[:] = np.random.rand(N.get(), M.get()).astype(np.float32)

gemm(A=A, B=B, C=C, N=N, K=K)  # M is fixed at compile-time

DaCe successfully runs 30/30 Polybench applications directly from NumPy (Xilinx + Intel)!

DaCe programs are exposed as Python functions

SDFGs defined or loaded using Python API

Interfaces with NumPy arrays

DaCe successfully runs 30/30 Polybench applications directly from NumPy (Xilinx + Intel)!

DaCe programs are exposed as Python functions

SDFGs defined or loaded using Python API

Interfaces with NumPy arrays
Case study: Weather simulation at MeteoSwiss

[1] COSMO 1.1 km
2018-05-29 00:00 UTC+2

Weather service is currently GPU-powered [2]

[1] Institute for Atmospheric and Climate Science and Computer Graphics Laboratory, ETH Zürich [https://vimeo.com/389292423]
[2] Swiss National Supercomputing Center (CSCS) [https://www.cscs.ch/computers/arolla-tera-meteo-swiss]
Weather stencil programs

Idea: Pipeline dependencies in a fine-grained manner on a spatial computing architecture.

Gysi et al.: MODESTO: Data-centric Analytic Optimization of Complex Stencil Programs on Heterogeneous Architectures, 29th International Conference on Supercomputing (ICS'15)
Map the **entire** stencil program to a **fully-pipelined parallel** streaming hardware architecture.

Transparently expose this mapping from a high-level scientific **DSL**.
Stencil Operation

```c
for (int i = 1; i < N - 1; ++i) {
    for (int j = 1; j < M - 1; ++j) {
        c[i, j] = (a[i-1, j] + a[i+1, j]) - (b[i, j-1] + b[i, j+1]);
    }
}
```
A stencil program is a DAG of stencil operations working on the same grid.

Gysi et al.: MODESTO: Data-centric Analytic Optimization of Complex Stencil Programs on Heterogeneous Architectures, 29th International Conference on Supercomputing (ICS'15)
Prototype: The StencilFlow Stack

```
{
  "dimensions": [4096, 4096],
  "vectorization": 8,
  "outputs": ["d"],
  "inputs": {
    "a": {
      "data_type": "float32",
      "input_dims": ["j", "k"]
    },
    "c0": {
      "data_type": "float32",
      "input_dims": []
    },
    "c1": {
      "data_type": "float32",
      "input_dims": []
    },
    "c2": {
      "data_type": "float32",
      "input_dims": []
    },
    "c3": {
      "data_type": "float32",
      "input_dims": []
    },
    "c4": {
      "data_type": "float32",
      "input_dims": []
    }
  },
  "program": {
    "b": {
      "data_type": "float32",
      "boundary": {"a": {
        "type": "constant",
        "value": 0
      }},
      "computation": "b[i] = c0*a[j, k] + c1*a[j-1, k] + c2*a[j+1, k] + c3*a[j, k-1] + c4*a[j, k+1]
    },
    "d": {
      "data_type": "float32",
      "boundary": {"b": {
        "type": "constant",
        "value": 0
      }},
      "computation": "c[i] = c0*b[j, k] + c1*b[j-1, k] + c2*b[j+1, k] + c3*b[j, k-1] + c4*b[j, k+1]
    }
  }
}
```

High-level Python interface

```
% bin/run_program.py diffusion.json hardware
```

Frontend

```
{
  "dimensions": [4096, 4096],
  "vectorization": 8,
  "outputs": ["d"],
  "inputs": {
    "a": {
      "data_type": "float32",
      "input_dims": ["j", "k"]
    },
    "c0": {
      "data_type": "float32",
      "input_dims": []
    },
    "c1": {
      "data_type": "float32",
      "input_dims": []
    },
    "c2": {
      "data_type": "float32",
      "input_dims": []
    },
    "c3": {
      "data_type": "float32",
      "input_dims": []
    },
    "c4": {
      "data_type": "float32",
      "input_dims": []
    }
  },
  "program": {
    "b": {
      "data_type": "float32",
      "boundary": {
        "a": {
          "type": "constant",
          "value": 0
        }
      },
      "computation": "b = c0*a[j,k] + c1*a[j-1,k] + c2*a[j+1,k] + c3*a[j,k-1] + c4*a[j,k+1]"
    },
    "d": {
      "data_type": "float32",
      "boundary": {
        "b": {
          "type": "constant",
          "value": 0
        }
      },
      "computation": "c = c0*b[j,k] + c1*b[j-1,k] + c2*b[j+1,k] + c3*b[j,k-1] + c4*b[j,k+1]"
    }
  }
}
```

JSON-based domain-specific frontend

Stencil DAG

High-level Python interface

```
% bin/run_program.py diffusion.json hardware
```
The StencilFlow Stack

JSON-based domain-specific frontend

High-level Python interface

% bin/run_program.py diffusion.json hardware

Stencil DAG

DaCe Dataflow Graph

Expanded Graph

Abstract Hardware Mapping

Generated code

Intel FPGA OpenCL

Reference C++

…
Two classes of buffers

We exploit all available reuse, such that by design, every memory location is read exactly once.

**Internal buffers** maximize data reuse within each stencil operation.  
**Delay buffers** maximize data reuse between stencil operations.
Internal buffers

Every cell used by a stencil operation is only read once.
Global reuse

We can exploit this reuse by **streaming** data through on-chip memory.
Delay buffers

Intermediate buffers represent physical hardware

Sizes are fixed at compile time

Design can **deadlock** when path is full.

\[
C[0] = B[0] + A[0]
\]

\[
\]

\[
A[0] = \ldots
\]
Delay buffers

No deadlock. Both paths stream simultaneously.

Delay buffers are inserted throughout the design.

Path delay is decided by internal buffer size...

...and pipeline depth.
The StencilFlow Stack - Results

```
{
  "dimensions": [4096, 4096],
  "vectorization": 8,
  "outputs": ["d"],
  "inputs": {
    "a": {
      "data_type": "float32",
      "input_dims": ["j", "k"],
    },
    "c0": {
      "data_type": "float32",
      "input_dims": [],
    },
    "c1": {
      "data_type": "float32",
      "input_dims": [],
    },
    "c2": {
      "data_type": "float32",
      "input_dims": [],
    },
    "c3": {
      "data_type": "float32",
      "input_dims": [],
    },
    "c4": {
      "data_type": "float32",
      "input_dims": [],
    }
  },
  "program": {
    "b": {
      "data_type": "float32",
      "boundary": {
        "a": {
          "type": "constant",
          "value": 0,
        }
      },
      "computation": "b[j, k] = c0*a[j, k] + c1*a[j-1, k] + c2*a[j+1, k] + c3*a[j, k-1] + c4*a[j, k+1]"
    },
    "d": {
      "data_type": "float32",
      "boundary": {
        "b": {
          "type": "constant",
          "value": 0,
        }
      },
      "computation": "d[j, k] = c0*b[j, k] + c1*b[j-1, k] + c2*b[j+1, k] + c3*b[j, k-1] + c4*b[j, k+1]"
    }
  }
}
```

High-level Python interface

% bin/run_program.py diffusion.json

Stencil DAG → Abstract Hardware Mapping → Generated code

DaCe Dataflow Graph → Expanded Graph → Intel FPGA OpenCL

References:
Simple iterative stencils

Communication implemented with SMI [1].

\[ FP32, W = 4, 24 \text{ Op/Stencil, } 2^{15} \times 32 \times 32 \text{ domain.} \]

Chain more stencils (cf. time tiling) to utilize spatial capacity

Simple iterative stencils

% bin/run_program.py diffusion_3d_vec8.json hardware
Taming the weather

Horizontal diffusion program obtained from generic weather frontend.
Taming the weather

Program is recovered as StencilFlow **DSL**.

```
"inputs": {
  "pp_in": {
    "data": "pp_in_128x64x128_float64.dat",
    "data_type": "float64",
    "dimensions": [
      "i",
      "j",
      "k"
    ]
  },
  "crlato": {
    "data": "crlato_128_float64.dat",
    "data_type": "float64",
    "dimensions": [
      "i"
    ]
  },
  "crlatu": {
    "data": "crlatu_128_float64.dat",
    "data_type": "float64",
    "dimensions": [
      "i"
    ]
  },
  "hdmask": {
    "data": "hdmask_128x64x128_float64.dat",
    "data_type": "float64",
    "dimensions": [
      "i",
      "j",
      "k"
    ]
  },
  "w_in": {
    "data": "w_in_128x64x128_float64.dat",
    "data_type": "float64",
    "dimensions": [
      "i",
      "j",
      "k"
    ]
  }
},
```
Taming the weather

We can now generate and process the stencil DAG.

- Performs $130$ floating point operations per grid point (including sqrt, min, and max operations).
- 20 data-dependent branches
- Arithmetic intensity of $65/18 \text{ Op/Byte}$. 
Taming the weather

CPU and GPU results are highly optimized codes produced by the DAWN framework [1].

<table>
<thead>
<tr>
<th>Stratix 10</th>
<th>Runtime</th>
<th>Performance</th>
<th>Peak BW.</th>
<th>%Roof.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1,178 μs</td>
<td>145 GOP/s</td>
<td>77 GB/s</td>
<td>52%</td>
</tr>
</tbody>
</table>

318 MHz at 48% DSP utilization

all memory bound

The Stratix 10 is held back by insufficient bandwidth.

FPGAs are good at deterministically **exploiting bandwidth**, but require a lot of **pipeline parallelism**.

Reproducibility artifact

We would love it if you **break it**!

https://github.com/spcl/stencilflow
Maybe we can get a full weather/climate code on FPGAs

New Study: Scientists Create Earth's Highly-Accurate 'Digital Twin' to Project Climate Event

By Precious Smith Mar 01, 2021 11:09 AM EST

Computer scientists at ETH (Swiss Federal Institute of Technology, Zurich) are trying to create a detailed digital twin of our Big Blue Marble in a kind of pixelated cloning experiment intended to serve as a model of experiment for Earth's climatic changes.

GPU: The Most Encouraging Option

Presently, researchers have the believe that super computers based on graphics processing units (GPU) seem to be the most encouraging option for the creation of their digital earth.

They appraise that running a full-scale digital twin, matching hardware with advanced algorithms, would need a system operating approximately 20,000 GPUs and consuming nearly 20MW of total power.
Despite all data-movement optimizations, our frequencies were far from the target (often <200 MHz)

```
int i, k;
for(i=0; i<k; ++i)
    n += n*obj;
    k = n*n
```
Another trick: temporal vectorization aka. automatic multi-pumping

How can temporal vectorization be implemented?

```python
@dace
def vadd(x: dace.float32[N], y: dace.float32[N]):
    return x + y
```

# Performance of temporal vectorization on a Xilinx AMD Alveo U280

<table>
<thead>
<tr>
<th></th>
<th>32 PEs</th>
<th>S=8</th>
<th>S=8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CA [10] O</td>
<td></td>
<td>O</td>
</tr>
<tr>
<td>Freq CL0 [MHz]</td>
<td>250</td>
<td>268</td>
<td>261.4</td>
</tr>
<tr>
<td>Freq CL1 [MHz]</td>
<td>-</td>
<td>-</td>
<td>452.8</td>
</tr>
<tr>
<td>Perf [GOp/s]</td>
<td>253.2</td>
<td>256.1</td>
<td>219.1</td>
</tr>
<tr>
<td>LUT Logic [%]</td>
<td>43.9</td>
<td>44.8</td>
<td>32.1</td>
</tr>
<tr>
<td>LUT Memory [%]</td>
<td>6.9</td>
<td>13</td>
<td>10.1</td>
</tr>
<tr>
<td>Registers [%]</td>
<td>44.5</td>
<td>44.3</td>
<td>36.6</td>
</tr>
<tr>
<td>BRAM [%]</td>
<td>81.4</td>
<td>80.3</td>
<td>47</td>
</tr>
<tr>
<td>DSP [%]</td>
<td>88.9</td>
<td>90</td>
<td>45.6</td>
</tr>
<tr>
<td>MOp/s per DSP</td>
<td>98.9</td>
<td>98.8</td>
<td>167.0</td>
</tr>
</tbody>
</table>

Matrix Multiplication

<table>
<thead>
<tr>
<th></th>
<th>S=8</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Freq CL0 [MHz]</td>
<td>307.6</td>
<td>322.4</td>
</tr>
<tr>
<td>Freq CL1 [MHz]</td>
<td>-</td>
<td>510.4</td>
</tr>
<tr>
<td>Perf [GOp/s]</td>
<td>101.4</td>
<td>96.9</td>
</tr>
<tr>
<td>LUT Logic [%]</td>
<td>20.25</td>
<td>14.2</td>
</tr>
<tr>
<td>LUT Memory [%]</td>
<td>6.21</td>
<td>6.89</td>
</tr>
<tr>
<td>Registers [%]</td>
<td>22.48</td>
<td>19.14</td>
</tr>
<tr>
<td>BRAM [%]</td>
<td>15.33</td>
<td>10.57</td>
</tr>
<tr>
<td>DSP [%]</td>
<td>28.89</td>
<td>14.44</td>
</tr>
<tr>
<td>MOp/s per DSP</td>
<td>121.9</td>
<td>232.8</td>
</tr>
</tbody>
</table>

Jacobi 3D stencil

<table>
<thead>
<tr>
<th></th>
<th>S=8</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Freq CL0 [MHz]</td>
<td>309.1</td>
<td>329.4</td>
</tr>
<tr>
<td>Freq CL1 [MHz]</td>
<td>-</td>
<td>537.3</td>
</tr>
<tr>
<td>Perf [GOp/s]</td>
<td>110.4</td>
<td>102.8</td>
</tr>
<tr>
<td>LUT Logic [%]</td>
<td>16.55</td>
<td>12.08</td>
</tr>
<tr>
<td>LUT Memory [%]</td>
<td>4.85</td>
<td>5.27</td>
</tr>
<tr>
<td>Registers [%]</td>
<td>18.25</td>
<td>15.88</td>
</tr>
<tr>
<td>BRAM [%]</td>
<td>10.57</td>
<td>8.18</td>
</tr>
<tr>
<td>DSP [%]</td>
<td>31.67</td>
<td>16.67</td>
</tr>
<tr>
<td>MOp/s per DSP</td>
<td>121.0</td>
<td>214.2</td>
</tr>
</tbody>
</table>

Diffusion 3D stencil

DaCe is a versatile platform

- DaCeML (PyTorch/ONNX)
- C (C99)
- GridTools (Weather & Climate)
- Fortran (in planning)
- NumPy
- NVIDIA GPU
- AMD GPU
- x86 CPU
- ARM SVE
- Intel FPGA
- Xilinx AMD FPGA
- RTL (soon)

Your favorite language/DSL (through SDFG builder)
Your favorite processor (through C++ codegen)

Overview and wrap-up

This project has received funding from the European Research Council (ERC) under grant agreements "DAPP and PSAP (PI: T. Hoeffer)".

https://www.youtube.com/c/ScalableParallelComputingLabETHZurich
First Example: 2D Stencil

**Initialization (read input)**

**State s0**

- \([y=0:H,x=0:W]\)
- Initialize
- \([y=0:H,x=0:W]\)
- \(B[y,x]\)
- \([y=0:H,x=0:W]\)
- \(B[0:H,0:W]\)
- \(B\)

**State s1**

- \([y=0:H, x=0:W]\)
- \(A[0:H,0:W]\)
- \(t < T; t++\)
- \(t \geq T\)
- **Iteration (solver)**
- **Exit (converged)**

**Initialization**

- \([y=0:H,x=0:W]\)
- \(A[y-1,x]\)
- \(A[y+1,x]\)
- \(A[y,x-1]\)
- \(A[y,x+1]\)

**Jacobi**

- \(B[y-1,x]\)
- \(B[y,x]\)
- \(B[y+1,x]\)

**Iteration**

- \(t < T; t++\)

**Exit**

- Converged

---

Ben-Nun, de Fine Licht, Ziogas, TH: Stateful Dataflow Multigraphs: A Data-Centric Model for High-Performance Parallel Programs, SC19
Second Example: MatMul

```python
@dace.program
def gemm(A, B, C):
    # Transient variable
tmp = dace.define_local([M, N, K], dtype=A.dtype)

@dace.map
def multiplication(i: _[0:M], j: _[0:N], k: _[0:K]):
    in_A << A[i,k]
in_B << B[k,j]
out >> tmp[i,j,k]

out = in_A * in_B
dace.reduce(lambda a, b: a + b, tmp, C, axis=2)
```

DaCe-Python Explicit (minimal) side-effect code

\[ N^3 \text{ size dataflow temporary!} \]
Second Example: MatMul

```python
@dace.program
def gemm(A, B, C):
    # Transient variable
tmp = dace.define_local([M, N, K], dtype=A.dtype)

@dace.map
def multiplication(i: [0:M], j: [0:N], k: [0:K]):
    in_A << A[i,k]
in_B << B[k,j]
out >> tmp[i,j,k]

out = in_A * in_B
dace.reduce(lambda a, b: a + b, tmp, C, axis=2)
```

MapReduce Fusion transformation to $N^2$ size
Optimizing Transformer Deep Neural Networks

Back to our first example – Laplace in DaCe Python

```python
@dace.program
def Laplace(A: dace.float64[2,N],
            T: dace.uint32):
    for t in range(T):
        for i in dace.map[1:N-1]:
            # Data dependencies
            in_l << A[t%2, i-1]
            in_c << A[t%2, i]
            in_r << A[t%2, i+1]
            out >> A[(t+1)%2, i]
            # Computation
            out = in_l - 2*in_c + in_r
```

![Diagram of Laplace computation in DaCe]
DIODE User Interface (moving into vscode)

Source Code

Transformations

SDFG (malleable)

Generated Code

Performance

SDFG
Performance for matrix multiplication on x86

SDFG

Ben-Nun, de Fine Licht, Ziogas, TH: Stateful Dataflow Multigraphs: A Data-Centric Model for High-Performance Parallel Programs, SC19
Performance for matrix multiplication on x86

SDFG

MapReduceFusion (27 SLOC)

Ben-Nun, de Fine Licht, Ziqas, TH: Stateful Dataflow Multigraphs: A Data-Centric Model for High-Performance Parallel Programs, SC19
Performance for matrix multiplication on x86

SDFG

LoopReorder (27 SLOC)
MapReduceFusion (27 SLOC)

Ben-Nun, de Fine Licht, Ziogas, TH: Stateful Dataflow Multigraphs: A Data-Centric Model for High-Performance Parallel Programs, SC19
Performance for matrix multiplication on x86

SDFG

BlockTiling (39 SLOC)
LoopReorder (27 SLOC)
MapReduceFusion (27 SLOC)

Ben-Nun, de Fine Licht, Zioagas, TH: Stateful Dataflow Multigraphs: A Data-Centric Model for High-Performance Parallel Programs, SC19
Performance for matrix multiplication on x86

- RegisterTiling (47 SLOC)
- BlockTiling (39 SLOC)
- LoopReorder (27 SLOC)
- MapReduceFusion (27 SLOC)

Ben-Nun, de Fine Licht, Ziogas, TH: Stateful Dataflow Multigraphs: A Data-Centric Model for High-Performance Parallel Programs, SC19
Performance for matrix multiplication on x86

- Naïve
- LocalStorage (50 SLOC)
- RegisterTiling (47 SLOC)
- BlockTiling (39 SLOC)
- LoopReorder (27 SLOC)
- MapReduceFusion (27 SLOC)
Performance for matrix multiplication on x86

PromoteTransient (51 SLOC)
LocalStorage (50 SLOC)
RegisterTiling (47 SLOC)
BlockTiling (39 SLOC)
LoopReorder (27 SLOC)
MapReduceFusion (27 SLOC)
Performance for matrix multiplication on x86

With more tuning: 98.6% of MKL for specific inputs (587 SLOC)

But do we really care about MatMul on x86 CPUs?

Ben-Nun, de Fine Licht, Ziogas, TH: Stateful Dataflow Multigraphs: A Data-Centric Model for High-Performance Parallel Programs, SC19
Code Generation for Load/Store Architectures

- **Recursive code generation (C++, CUDA)**
  - **Control flow:** Construct detection and gotos

- **Parallelism**
  - **Multi-core CPU:** OpenMP, atomics, and threads
  - **GPU:** CUDA kernels and streams
  - Connected components run concurrently

- **Memory and interaction with accelerators**
  - Array-array edges create intra-/inter-device copies
  - Memory access validation on compilation
  - Automatic CPU SDFG to GPU transformation

- **Tasklet code immutable**
Code Generation for Pipelined Architectures

- **Module generation with HDL and HLS**
  - Integration with Xilinx SDAccel or OpenCL (RTL in development)
  - Nested SDFGs become FPGA state machines

- **Parallelism**
  - Exploiting temporal locality: Pipelines
  - Exploiting spatial locality: Vectorization, replication

- **Replication**
  - Enables parametric systolic array generation

- **Memory access**
  - Burst memory access, vectorization
  - Streams for inter-PE communication
Performance (Portability) Evaluation

- **Three platforms:**
  - Intel Xeon E5-2650 v4 CPU (2.20 GHz, no HT)
  - Tesla P100 GPU
  - Xilinx VCU1525 hosting an XCVU9P FPGA

- **Compilers and frameworks:**
  - Compilers:
    - GCC 8.2.0
    - Clang 6.0
    - icc 18.0.3
  - Polyhedral optimizing compilers:
    - Polly 6.0
    - Pluto 0.11.4
    - PPCG 0.8
  - GPU and FPGA compilers:
    - CUDA nvcc 9.2
    - Xilinx SDAccel 2018.2
  - Frameworks and optimized libraries:
    - HPX
    - Halide
    - Intel MKL
    - NVIDIA CUBLAS, CUSPARSE, CUTLASS
    - NVIDIA CUB
Performance Evaluation: Fundamental Kernels (CPU)

- **Database Query**: roughly 50% of a 67,108,864 column
- **Matrix Multiplication (MM)**: 2048x2048x2048
- **Histogram**: 8192x8192
- **Jacobi stencil**: 2048x2048 for T=1024
- **Sparse Matrix-Vector Multiplication (SpMV)**: 8192x8192 CSR matrix (nnz=33,554,432)

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Performance</th>
<th>Baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database Query</td>
<td>8.12x faster</td>
<td>99.9% of MKL</td>
</tr>
<tr>
<td>MM</td>
<td>2.5x faster</td>
<td>98.6% of MKL</td>
</tr>
<tr>
<td>Histogram</td>
<td>82.7% of Halide</td>
<td>2.5x faster</td>
</tr>
<tr>
<td>SpMV</td>
<td>99.9% of MKL</td>
<td>2.5x faster</td>
</tr>
</tbody>
</table>

Ben-Nun, de Fine Licht, Ziogas, TH: Stateful Dataflow Multigraphs: A Data-Centric Model for High-Performance Parallel Programs, SC19
Performance Evaluation: Fundamental Kernels (GPU, FPGA)

GPU

90% of CUTLASS

FPGA

19.5x of Spatial

309,000x

Ben-Nun, de Fine Licht, Ziogas, TH: Stateful Dataflow Multigraphs: A Data-Centric Model for High-Performance Parallel Programs, SC19
Performance Evaluation: Polybench (CPU)

- Polyhedral benchmark with 30 applications
- Without any transformations, achieves 1.43x (geometric mean) over general-purpose compilers
Performance Evaluation: Polybench (GPU, FPGA)

- Automatically transformed from CPU code

**GPU**

(1.12x geometric speedup)

**FPGA**

The first full set of placed-and-routed Polybench
Case Study: Parallel Breadth-First Search

- Compared with Galois and Gluon
  - State-of-the-art graph processing frameworks on CPU

- Graphs:
  - Road maps: USA, OSM-Europe
  - Social networks: Twitter, LiveJournal

Performance portability – fine, but who cares about microbenchmarks?
Remember: Scientific Software Engineering in the 21st century

\[
\frac{\partial u}{\partial t} - \alpha \nabla^2 u = 0
\]

Parametric Dataflow Graphs (SDFG)

**Domain Scientist**
- NumPy
- TensorFlow
- PyTorch
- MATLAB

**Applied Scientist**
- translate DSL into parametric dataflow graphs
- SDFG Builder API
  - Multi-Level Library Nodes

**Performance Engineer**
- Graph Transformations (API, Interactive)
- Graph Transformations (API, Interactive)
- Transformed Dataflow

**Performance Results**
- Specialized Code Generation
  - CPU Code
  - GPU Code
  - FPGA Code

**Runtime**
- C++ code generation/runtime

Ben-Nun, de Fine Licht, Ziogas, TH: Stateful Dataflow Multigraphs: A Data-Centric Model for High-Performance Parallel Programs, SC19
Next-Generation Transistors need to be cooler – addressing self-heating

Ziogas et al. “A Data-Centric Approach to Extreme-Scale Ab Initio Dissipative Quantum Transport Simulations”, SC19 (Gordon Bell Prize 2019).
Quantum Transport Simulations with OMEN

- OMEN Code (Luisier et al., Gordon Bell award finalist 2011 and 2015)
  - 90k SLOC, C, C++, CUDA, MPI, OpenMP, ...

\[
\text{GF} \quad \text{SSE}\quad \Sigma[G(E + \hbar\omega, k_z - q_z) D(\omega, q_z)](E, k_z)
\]

\[
\text{Electrons } G(E, k_z) \\
\left( E \cdot S - H - \Sigma^R \right) \cdot G^R = I \quad G^< = G^R \cdot \Sigma^< \cdot G^A
\]

\[
\text{Phonons } D(\omega, q_z) \\
\left( \omega^2 - \Phi - \Pi^R \right) \cdot D^R = I \quad D^< = D^R \cdot \Pi^< \cdot D^A
\]

\[
\text{SSE} \quad \Pi[G(E, k_z) G(E + \hbar\omega, k_z + q_z)](\omega, q_z)
\]

Ziogas et al. “A Data-Centric Approach to Extreme-Scale Ab Initio Dissipative Quantum Transport Simulations”, SC19 (Gordon Bell Prize 2019).
OMEN

\[ N_{kZ} \]

\[ N_{qZ} \]

\[ N_E \]

\[ N_\omega \]

Ziogas et al. “A Data-Centric Approach to Extreme-Scale Ab Initio Dissipative Quantum Transport Simulations”, SC19 (Gordon Bell Prize 2019).
All of OMEN (90k SLOC) in a single SDFG – (collapsed) tasklets contain more SDFGs
Zooming into SSE (large share of the runtime)

Between 100-250x less communication at scale! (from PB to TB)
Additional interesting performance insights

Python is slow! Ok, we knew that – but compiled can be fast!

<table>
<thead>
<tr>
<th>Variant</th>
<th>Phase</th>
<th>GF</th>
<th>SSE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Tilop</td>
<td>Time [s]</td>
</tr>
<tr>
<td>OMEN</td>
<td></td>
<td>174.0</td>
<td>144.14</td>
</tr>
<tr>
<td>Python</td>
<td></td>
<td>174.0</td>
<td>1,342.77</td>
</tr>
<tr>
<td>DaCe</td>
<td></td>
<td>174.0</td>
<td>111.25</td>
</tr>
</tbody>
</table>

Piz Daint single node (P100)

CuBLAS can be very inefficient (well, unless you floptimize)

<table>
<thead>
<tr>
<th></th>
<th>cuBLAS</th>
<th>DaCe (SBSMM)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GPU</td>
<td>Glop</td>
</tr>
<tr>
<td>P100</td>
<td>27.42</td>
<td>6.73 ms</td>
</tr>
<tr>
<td>V100</td>
<td>27.42</td>
<td>4.62 ms</td>
</tr>
</tbody>
</table>

Basic operation in SSE (many very small MMMs)

5k atoms
An example of fine-grained data-centric optimization (i.e., how to vectorize)

Ziogas et al. “A Data-Centric Approach to Extreme-Scale Ab Initio Dissipative Quantum Transport Simulations”, SC19 (Gordon Bell Prize 2019).
10,240 atoms on 27,360 V100 GPUs (full-scale Summit)

- 56 P flop/s with I/O (28% peak)

Communication time reduced by 417x on Piz Daint!
Volume on full-scale Summit from 12 PB/iter $\rightarrow$ 87 TB/iter

Ziogas et al. “A Data-Centric Approach to Extreme-Scale Ab Initio Dissipative Quantum Transport Simulations”, SC19 (Gordon Bell Prize 2019).