Data-Centric Parallel Programming
Torsten Hoefler, Keynote at AsHES @ IPDPS’19, Rio, Brazil

Alexandros Ziogas, Tal Ben-Nun, Guillermo Indalecio, Timo Schneider, Mathieu Luisier, and Johannes de Fine Licht and the whole DAPP team @ SPCL

EuroMPI’19
September 11-13 2019
Zurich, Switzerland
https://eurompi19.inf.ethz.ch
Changing hardware constraints and the physics of computing

How to address locality challenges on standard architectures and programming? D. Unat et al.: “Trends in Data Locality Abstractions for HPC Systems”

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."

Source: Mark Horowitz, ISSC’14

Control Localities

Very Low

High

Source: Mark Horowitz, ISSC’14
Single Instruction Multiple Data/Threads (SIMD - Vector CPU, SIMT - GPU)

High Performance Computing really became a data management challenge

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

- Legion [1] is taking a good direction towards data-centric
  - Tasking relies on data placement but not really dependencies (not visible to tool-chain)
  - But it is still control-centric in the tasks – not (performance) portable between devices!

- Let’s go a step further towards an explicitly data-centric viewpoint
  - For performance engineers at least!

---

[1]: Bauer et al.: “Legion: expressing locality and independence with logical regions”, SC12, 2012

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.”
Performance Portability with DataCentric (DaCe) Parallel Programming

**Domain Scientist**

- Problem Formulation
  \[
  \frac{\partial u}{\partial t} - \alpha \nabla^2 u = 0
  \]
  
- Python / NumPy
- TensorFlow
- SDFG Builder API
- High-Level Program

**Performance Engineer**

- Data-Centric Intermediate Representation (SDFG, §3)
  
- Graph Transformations (API, Interactive, §4)

**System**

- Hardware Information
- SDFG Compiler
- CPU Binary
- GPU Binary
- FPGA Modules

**Runtime**

- Thin Runtime Infrastructure

**Preprint (arXiv): Ben-Nun, de Fine Licht, Ziogas, TH: Stateful Dataflow Multigraphs: A Data-Centric Model for High-Performance Parallel Programs**
# DAPP – Data Centric Programming 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
A first example 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](chart.png)
DIODE User Interface

Source Code

```
@dapp.program(dapp.float64[M,N], dapp.float64[N,K], dapp.float64[K,1]):
  def sum(a,b): return a+b
```

Transformations

SDFG (malleable)

Generated Code

```
// SDFG for DIODE

// Tasklet code (multiplication)
// out = (in_A + in_B)
```

Performance
Performance for matrix multiplication on x86

SDFG

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

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

SDFG

LoopReorder
MapReduceFusion

Performance [GFlop/s]

Problem Size

Naive

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

SDFG

BlockTiling
LoopReorder
MapReduceFusion

Naive
Performance for matrix multiplication on x86

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

Performance [GFlop/s] vs Problem Size

- LocalStorage
- RegisterTiling
- BlockTiling
- LoopReorder
- MapReduceFusion

Naive
Performance for matrix multiplication on x86

PromoteTransient
LocalStorage
RegisterTiling
BlockTiling
LoopReorder
MapReduceFusion

Naïve

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

But do we really care about MMM on x86 CPUs?
Hardware Mapping: 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**

```c
void _program_gemm(int sym_0, int sym_1, int sym_2, double * __restrict__ re
// State s0
 for (int tile_k = 0; tile_k < sym_2; tile_k += 128) {
    #pragma omp parallel for
    for (int tile_i = 0; tile_i < sym_0; tile_i += 64) {
        for (int tile_j = 0; tile_j < sym_1; tile_j += 240) {
            vec<double, 4> local_B_s0_0[128 * 3];
            Global2Stack_2D_FixedWidth<4, double, 3>({88[t_loca

            for (int regtile_i = 0; regtile_i < (min(64, +
              vec<double, 4> regtile_C_s0_1[4 * 3];
              for (int i = 0; i < 4; i += 1) {
                  for (int j = 0; j < 3; j += 1) {
                      double in_A = A[(1 + regtile_i_i +
                      vec<double, 4> in_B = local_B_s0_0
                      // Tasklet code (mult)
                      auto out = in_A * in_B;
                      regtile_C_s0_1[1*3 + j] = out;

                     }
               for (int k = 1; k < (min(128, sym_2 - 1))
               // ...
```
Hardware Mapping: Pipelined Architectures

- **Module generation with HDL and HLS**
  - Integration with Xilinx SDAccel
  - 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)

8.12x faster 98.6% of MKL 2.5x faster 82.7% of Halide 99.9% of MKL
Performance Evaluation: Fundamental Kernels (GPU, FPGA)

90% of CUTLASS

GPU

19.5x of Spatial

FPGA

309,000x
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 the promise of DAPP – on to a real application!

**Domain Scientist**

- Problem Formulation: \( \frac{\partial u}{\partial t} - \alpha \nabla^2 u = 0 \)
- Python / NumPy
- TensorFlow
- SDFG Builder API
- High-Level Program

**Performance Engineer**

- Data-Centric Intermediate Representation (SDFG, §3)
- Graph Transformations (API, Interactive, §4)

**System**

- Hardware Information
- SDFG Compiler
- CPU Binary
- GPU Binary
- FPGA Modules
- Thin Runtime Infrastructure

**Transformation**

- Transformed Dataflow
- Performance Results
Next-Generation Transistors need to be cooler – addressing self-heating

Features:
- $w \leq 7$ nm
- $h \geq 40$ nm
- $L \leq 100$ nm

(b) due to the large $h/w$ ratio, the $z$-direction can be modeled as periodic and represented by momentum points.

(c) Atomic structure of Si channel (in red: atoms, in black: bonds).

(d) Atomically resolved temperature within the FinFET in (a-b), when a gate-to-source and drain-to-source voltage is applied.
Quantum Transport Simulations with OMEN

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

\[
\begin{align*}
\text{NEGF} & : \quad \text{SSE } \Sigma[\mathcal{G}(E + \hbar \omega, k_z - q_z) \mathcal{D}(\omega, q_z)](E, k_z) \\
\text{Electrons } \mathcal{G}(E, k_z) & : \quad (E \cdot s - H - \Sigma^R) \cdot G^R = I \\
& \quad G^< = G^R \cdot \Sigma^< \cdot G^A \\
\text{Phonons } \mathcal{D}(\omega, q_z) & : \quad (\omega^2 - \Phi - \Pi^R) \cdot D^R = I \\
& \quad D^< = D^R \cdot \Pi^< \cdot D^A \\
\text{GF} & : \quad \text{SSE } \Pi[\mathcal{G}(E, k_z) \mathcal{G}(E + \hbar \omega, k_z + q_z)](\omega, q_z)
\end{align*}
\]
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>T/lop</td>
<td>Time [s]</td>
<td>% Peak</td>
</tr>
<tr>
<td>OMEN</td>
<td>174.0</td>
<td>144.14</td>
<td>23.2%</td>
</tr>
<tr>
<td>Python</td>
<td>174.0</td>
<td>1,342.77</td>
<td>2.5%</td>
</tr>
<tr>
<td>DaCe</td>
<td>174.0</td>
<td>111.25</td>
<td>30.1%</td>
</tr>
</tbody>
</table>

Piz Daint single node (P100)

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

<table>
<thead>
<tr>
<th>Variant</th>
<th>Phase</th>
<th>GF</th>
<th>SSE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T/lop</td>
<td>Time [s]</td>
<td>% Peak</td>
</tr>
<tr>
<td>OMEN</td>
<td>174.0</td>
<td>144.14</td>
<td>23.2%</td>
</tr>
<tr>
<td>Python</td>
<td>174.0</td>
<td>1,342.77</td>
<td>2.5%</td>
</tr>
<tr>
<td>DaCe</td>
<td>174.0</td>
<td>111.25</td>
<td>30.1%</td>
</tr>
</tbody>
</table>

Basic operation in SSE (many very small MMMs)

5k atoms
10,240 atoms on 27,360 V100 GPUs (full-scale Summit)

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

<table>
<thead>
<tr>
<th>Variant</th>
<th>$N_a$</th>
<th>Time [s]</th>
<th>Time/Atom [s]</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMEN</td>
<td>1,064</td>
<td>4695.70</td>
<td>4.413</td>
<td>1.0x</td>
</tr>
<tr>
<td>DaCe</td>
<td>10,240</td>
<td>489.83</td>
<td>0.048</td>
<td>92.3x</td>
</tr>
</tbody>
</table>

$P = 6,840, N_b = 34, N_{orb} = 12, N_E = 1,220, N_{\omega} = 70.$

Already ~100x speedup on 25% of Summit – the original OMEN does not scale further!

Communication time reduced by 417x on Piz Daint!

Volume on full-scale Summit from 12 PB/iter $\rightarrow$ 87 TB/iter
An example of fine-grained data-centric optimization (i.e., how to vectorize)

\[ \sum_{q_2} \sum_{q_2} \int d\omega [\nabla H \cdot G(E + \hbar \omega, k_2 - q_2) \cdot \nabla H \cdot D(\omega, q_2)] \]
Overview and wrap-up

This project has received funding from the European Research Council (ERC) under grant agreement "DAPP (PI: T. Hoefler)".