A Component-Based Framework for the Cell Broadband Engine

Timothy D. R. Hartley, Umit V. Catalyurek
Department of Biomedical Informatics
Department of Electrical and Computer Engineering
The Ohio State University, Columbus, OH, USA
hartleyt@ece.osu.edu, umit@bmi.osu.edu

Outline

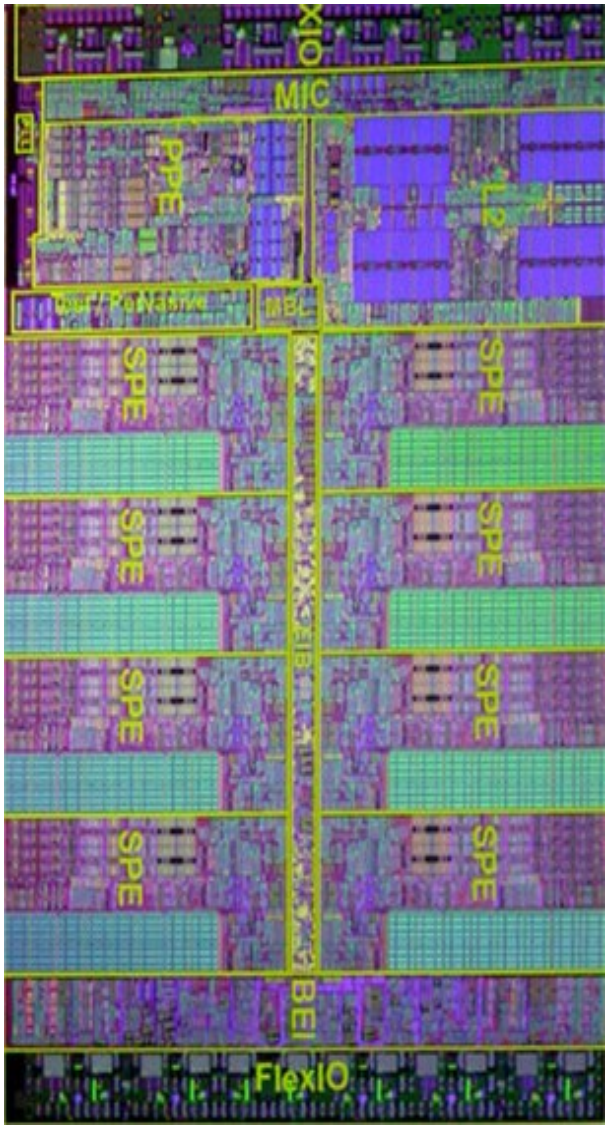
- Motivation
- Contributions
  - CBE Intercore Messaging Library
  - DataCutter-Lite
- Experimental Results
- Conclusions and Future Work

Motivation

- Programming the Cell requires expertise
  - Parallel programming
    - Data decomposition
    - Parallel algorithms
  - Streaming programming
    - Small scratch-pad memories
    - Double buffering
  - Cell peculiarities
    - DMA commands
    - SPE optimizations – not addressed here
- Component-based streaming frameworks are natural fits for heterogeneous, parallel processors

Cell Broadband Engine

- Cell Broadband Engine
  - Designed jointly by Sony, Toshiba, and IBM
  - 9-core heterogeneous microprocessor
  - Integrated high-bandwidth ring bus for on-chip communication
  - Quick Specs
    - >200 GFLOP/s floating-point arithmetic
    - >200 GB/s internal bus bandwidth



Cell Programming Peculiarities

- DMA commands
  - Simple in concept, difficult in practice
  - Fence, barrier, lists, alignments, etc.
- SPE code optimizations*
  - 25 GFLOP/s only reached on SPE when using SIMD FMA commands
  - Static dual-issue scheduling
  - Branch hints

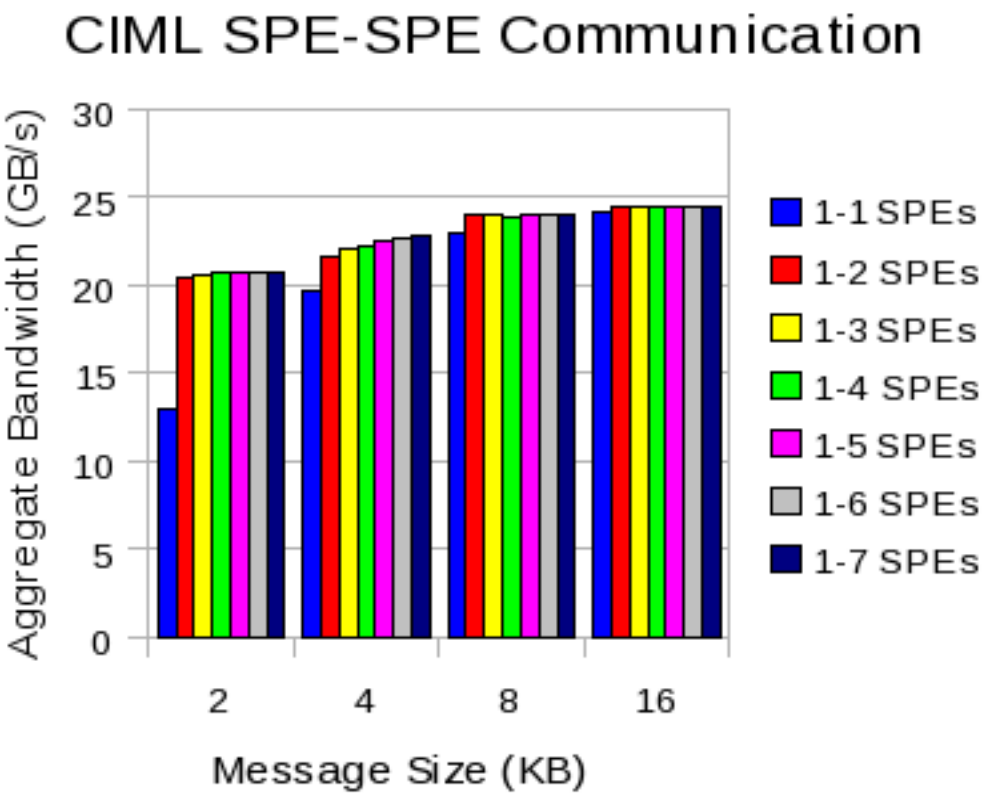
Contributions

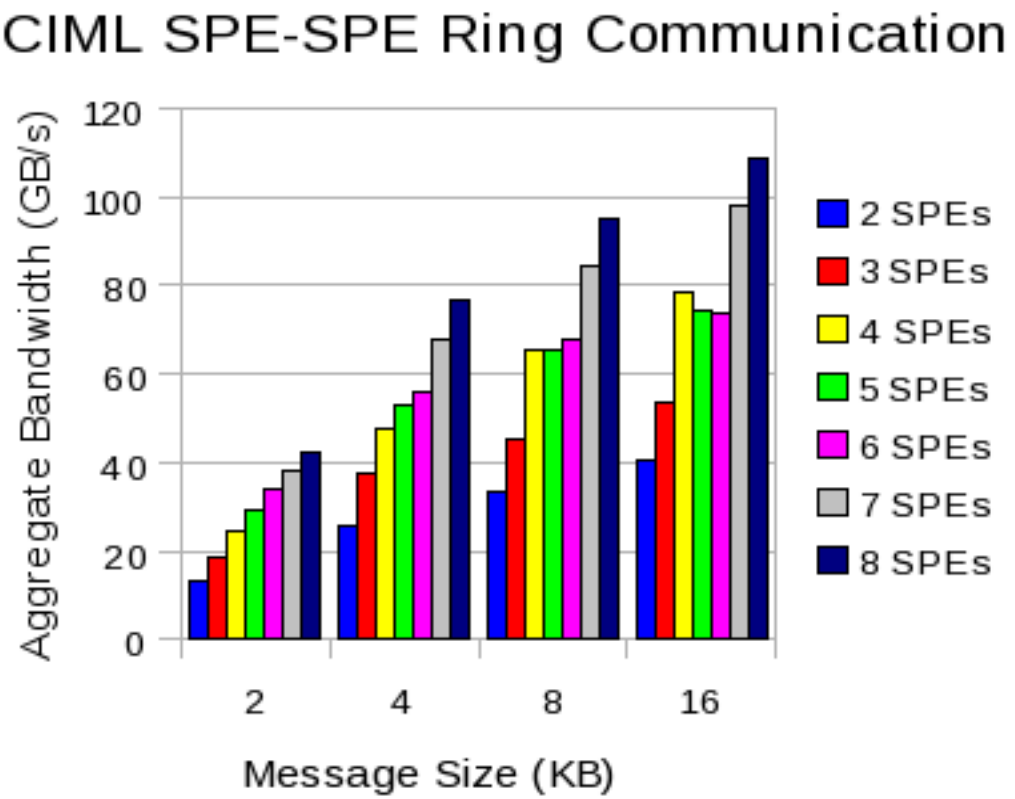
- Cell Broadband Engine Intercore Messaging Library (CIML)
  - Two-sided communication library
- DataCutter-Lite for Cell Broadband Engine
  - Filter-stream programming framework and runtime engine
  - Uses CIML for intercore communication

CBE Intercore Messaging Library (CIML)

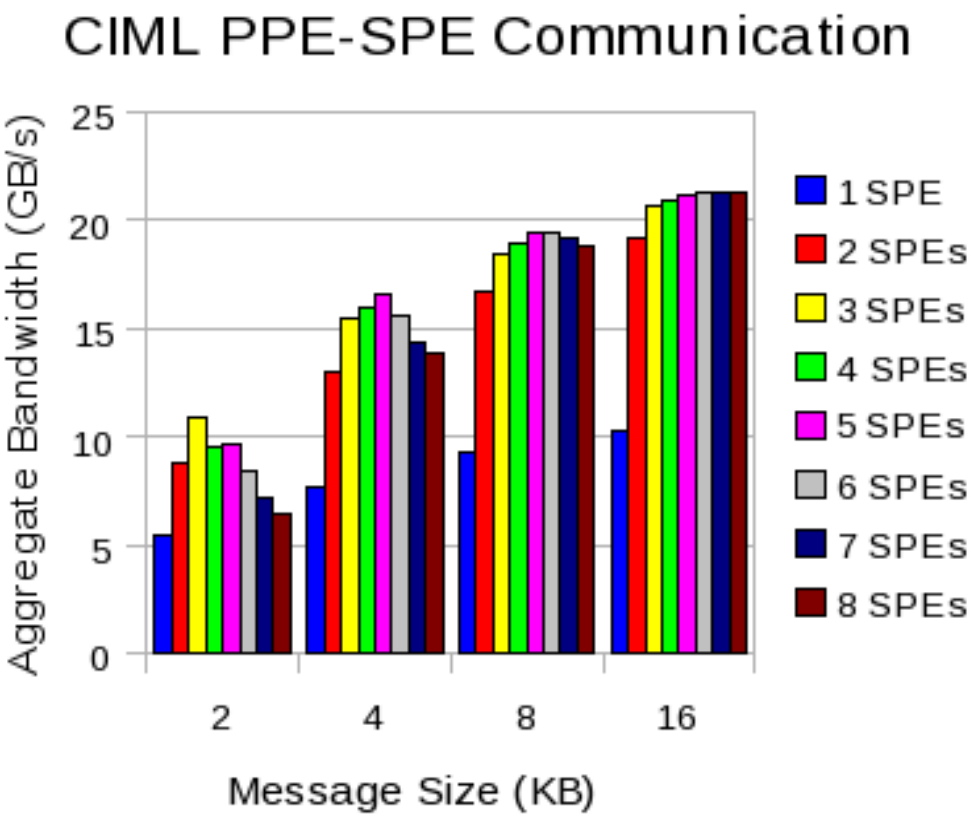
- Two-sided communication library
- Allows two-sided communication between all processors in Cell (PPU and SPU)
- Different from LANL's Cell Messaging Layer
  - CML uses a receiver-initiated protocol
  - Not suitable for streaming frameworks
    - Sender unknown
- Good performance for larger message sizes

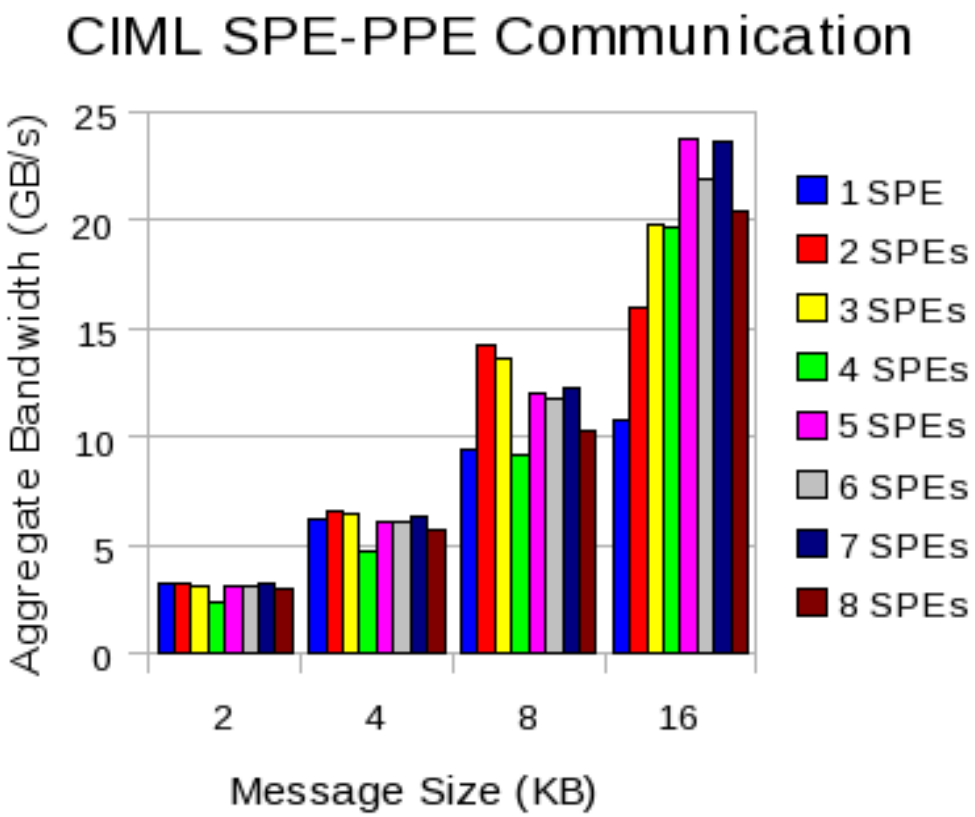
CIML Performance





CIML Performance





Component-based Programming Frameworks

- Application is decomposed into a natural task-graph
  - Task graph performs computation
  - Individual tasks perform single function
  - Tasks are independent, with well-defined interfaces
  - Higher-level programming abstraction

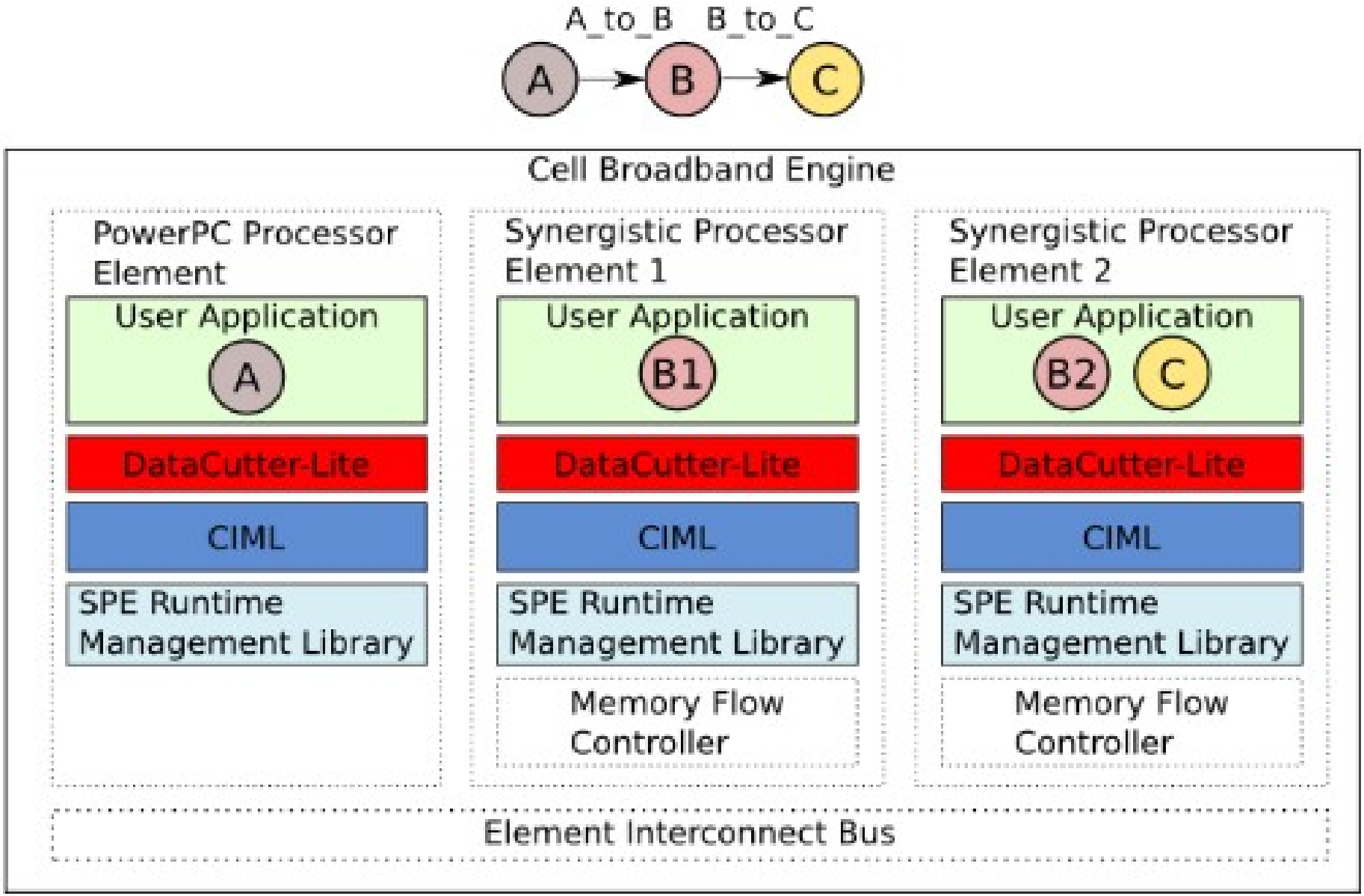
DataCutter

- DataCutter
  - Coarse-grained filter-stream framework
  - OSU/Maryland-bred component-based framework
  - Third-generation runtime uses MPI for high-bandwidth network support

DataCutter-Lite (DCL)

- Component-based, filter-stream programming framework
  - Define computation as task-graph
  - Tasks are *filters*, which are functions which compute
  - Data flows along *streams* to/from filters along pre-defined paths
  - Automatic multi-buffering of buffers
  - Automatic PPE-SPE, inter-SPE communication
- DCL is event-based
  - Arrival of stream *buffer* (a quantum of data in the application) triggers filter execution

Sample DCL Application Layout

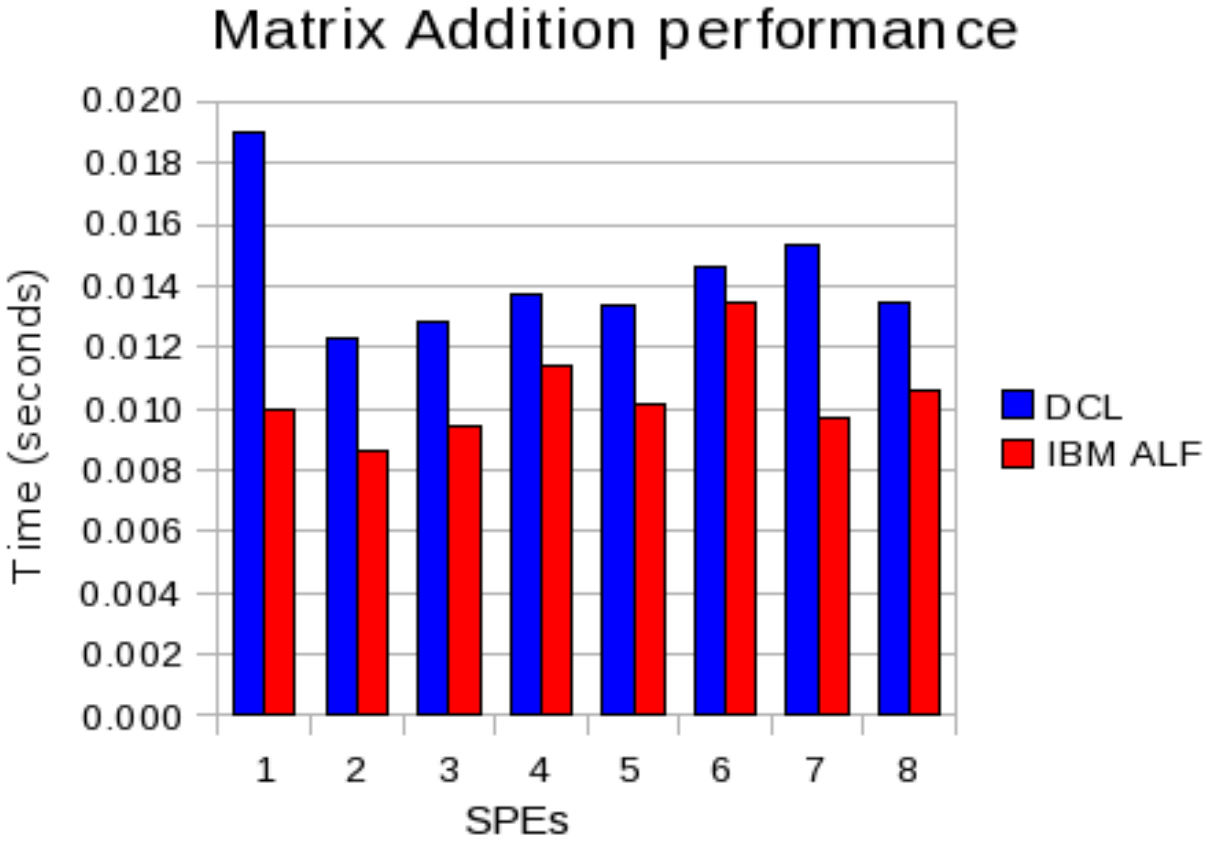


Experimental Results

- Use three applications
  - Variety of Communication-to-Computation Ratios (CCR)
- Matrix addition
  - High CCR
  - Compare with IBM's Accelerated Library Framework (ALF) example
- Image color-space transformation
  - Low CCR
  - Compare with custom-coded IBM SDK-based baseline
- Biomedical image analysis application
  - Medium CCR
  - Three-stage pipeline

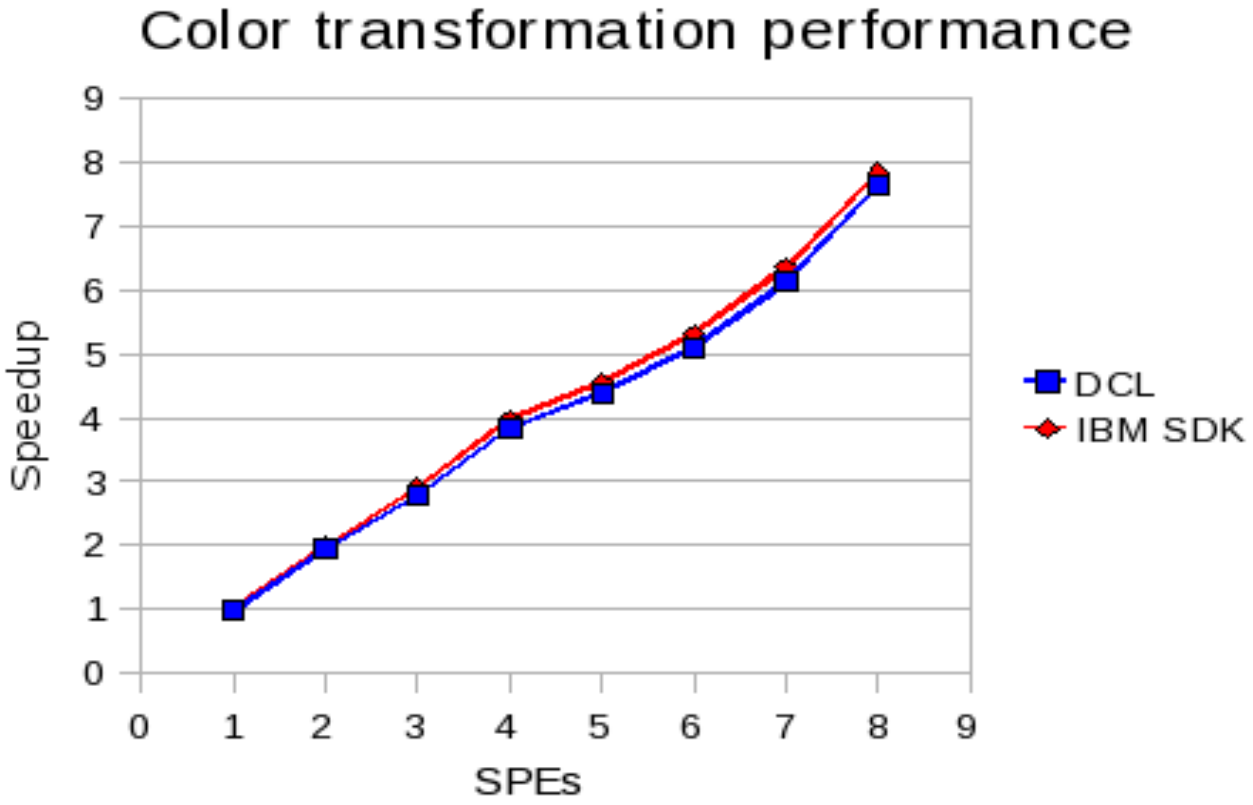
Matrix Addition Performance

- Compare against IBM ALF example
- 1024 x 512 matrix
- DCL has 8–91% longer execution time

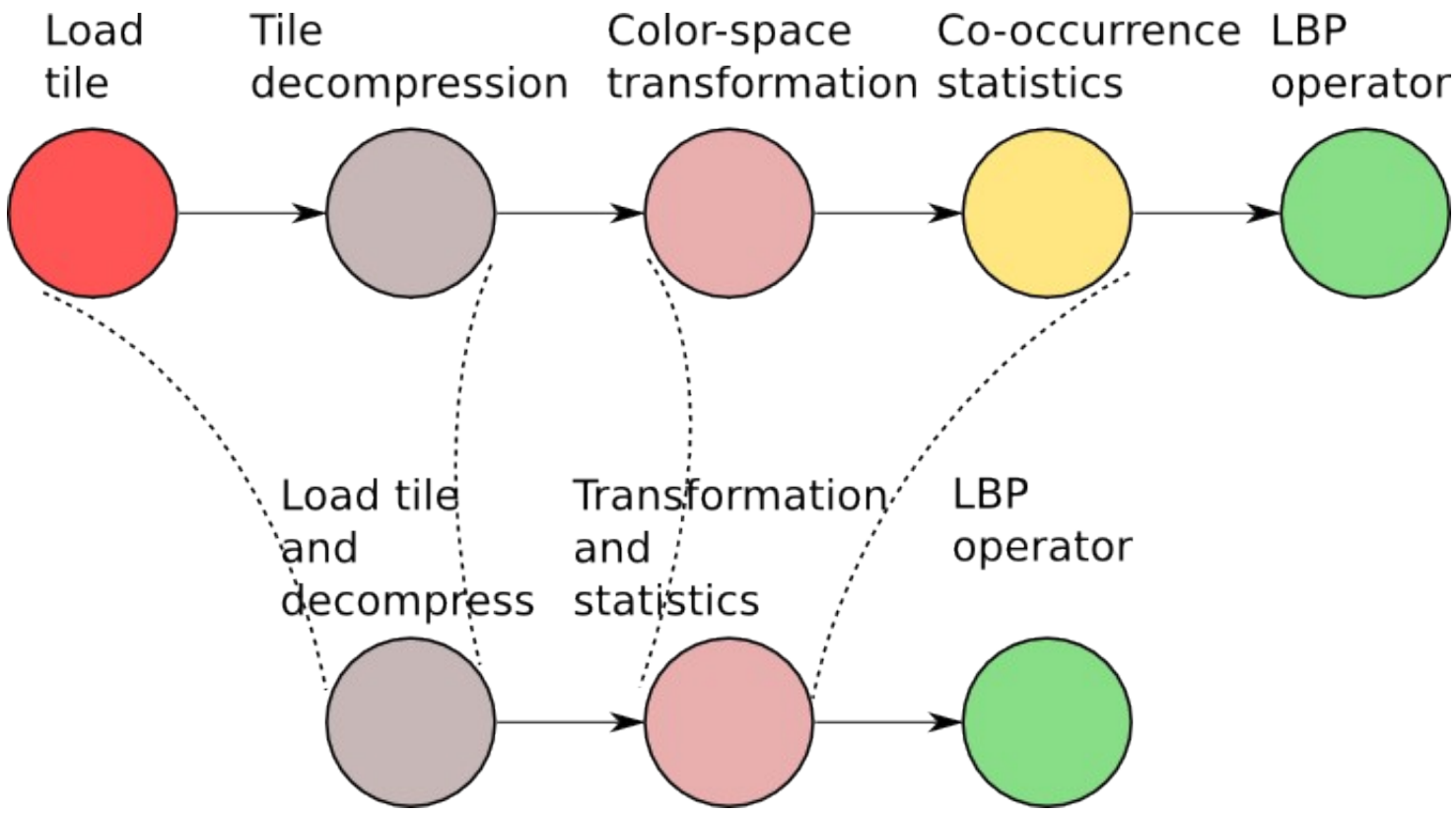


Color-Space Transform Performance

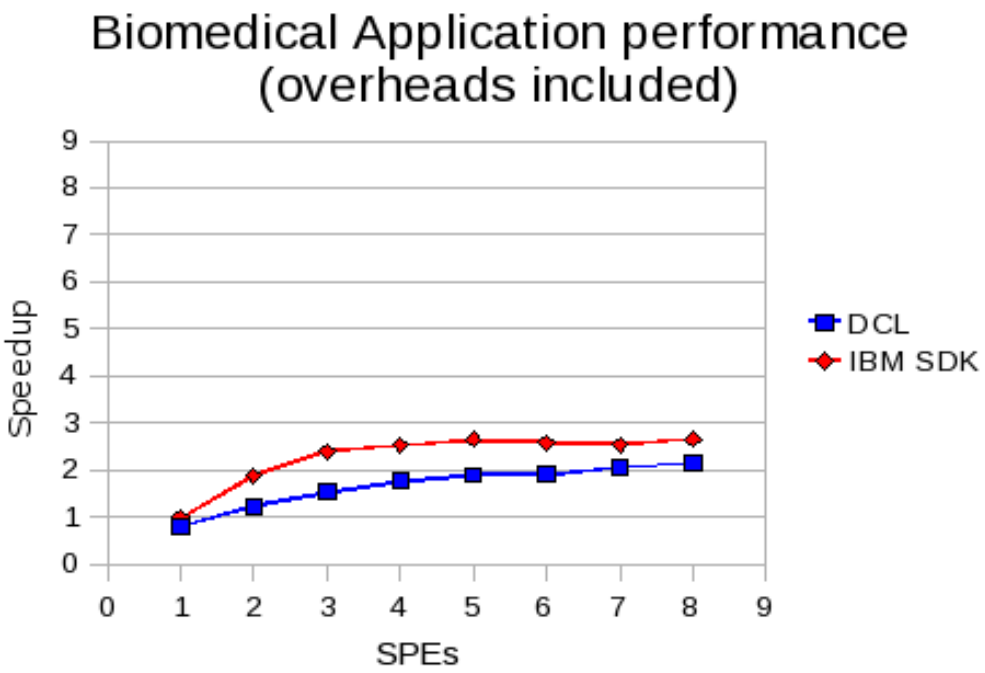
- Compare against custom IBM SDK version
- 32 1Kx1K image tiles
- DCL has 2-4% longer execution time

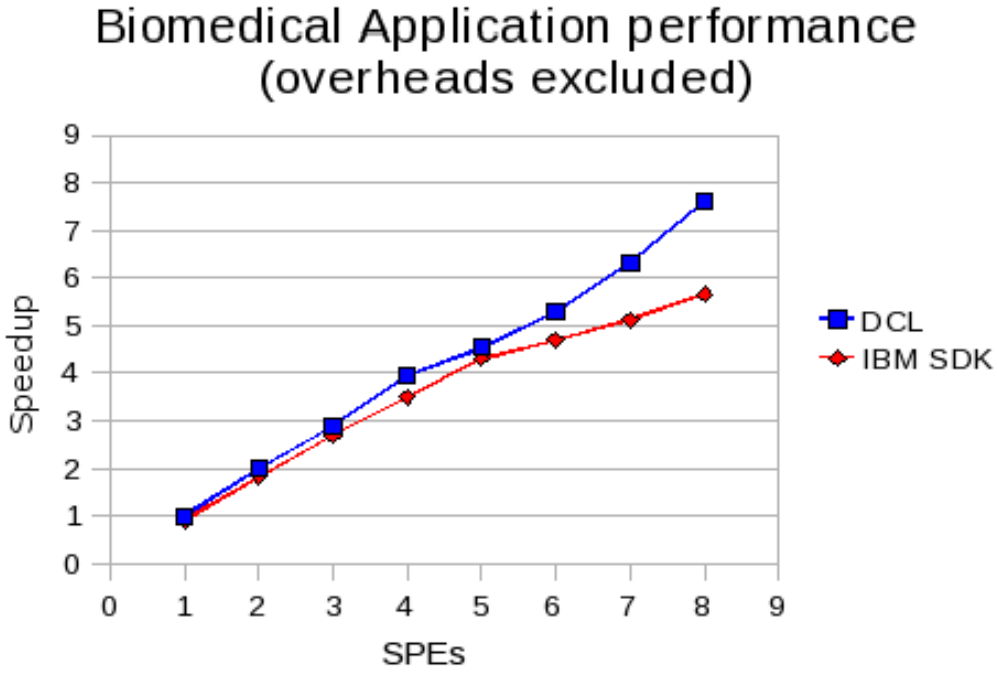


Biomedical Application Filter Layout



Biomedical Application Performance

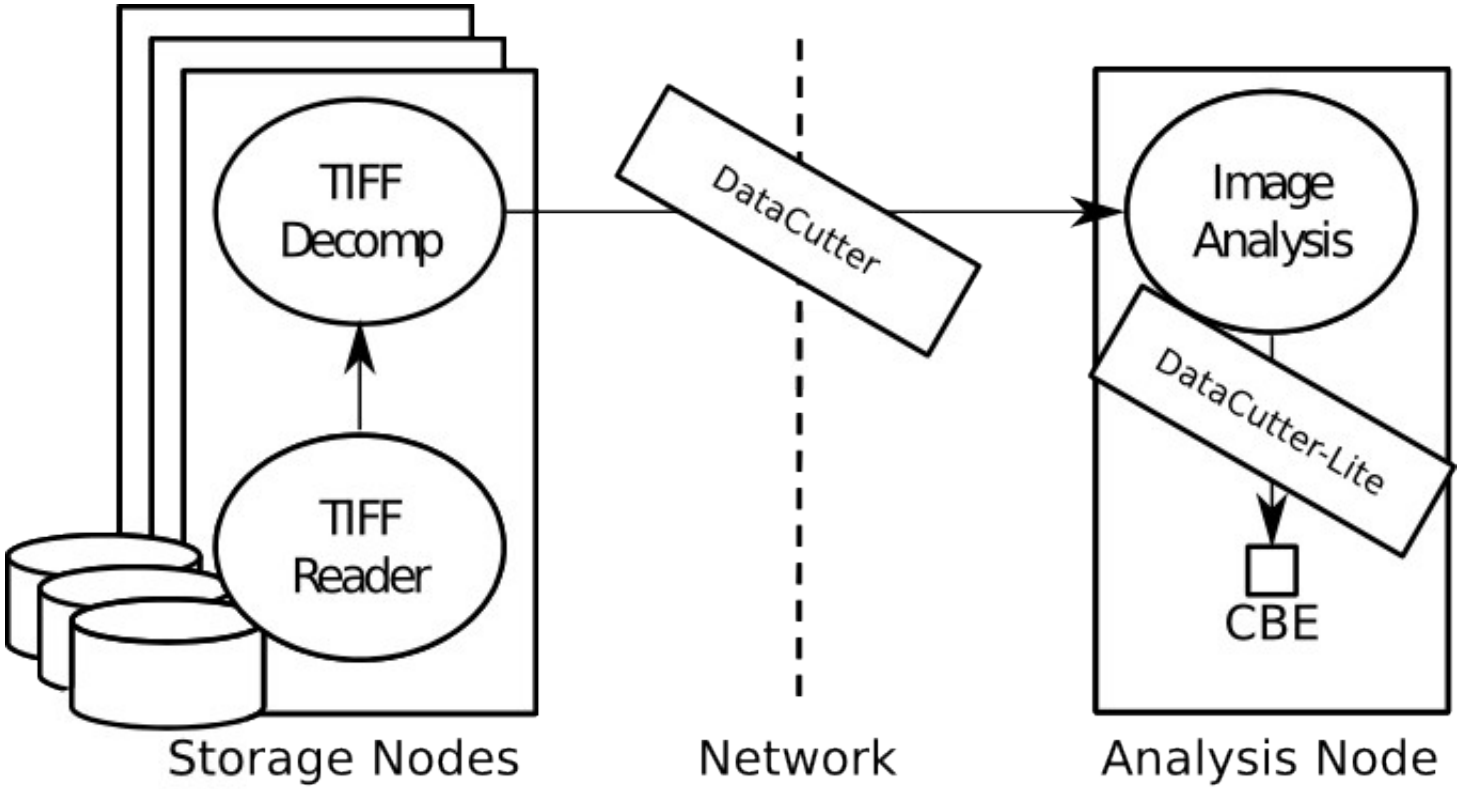




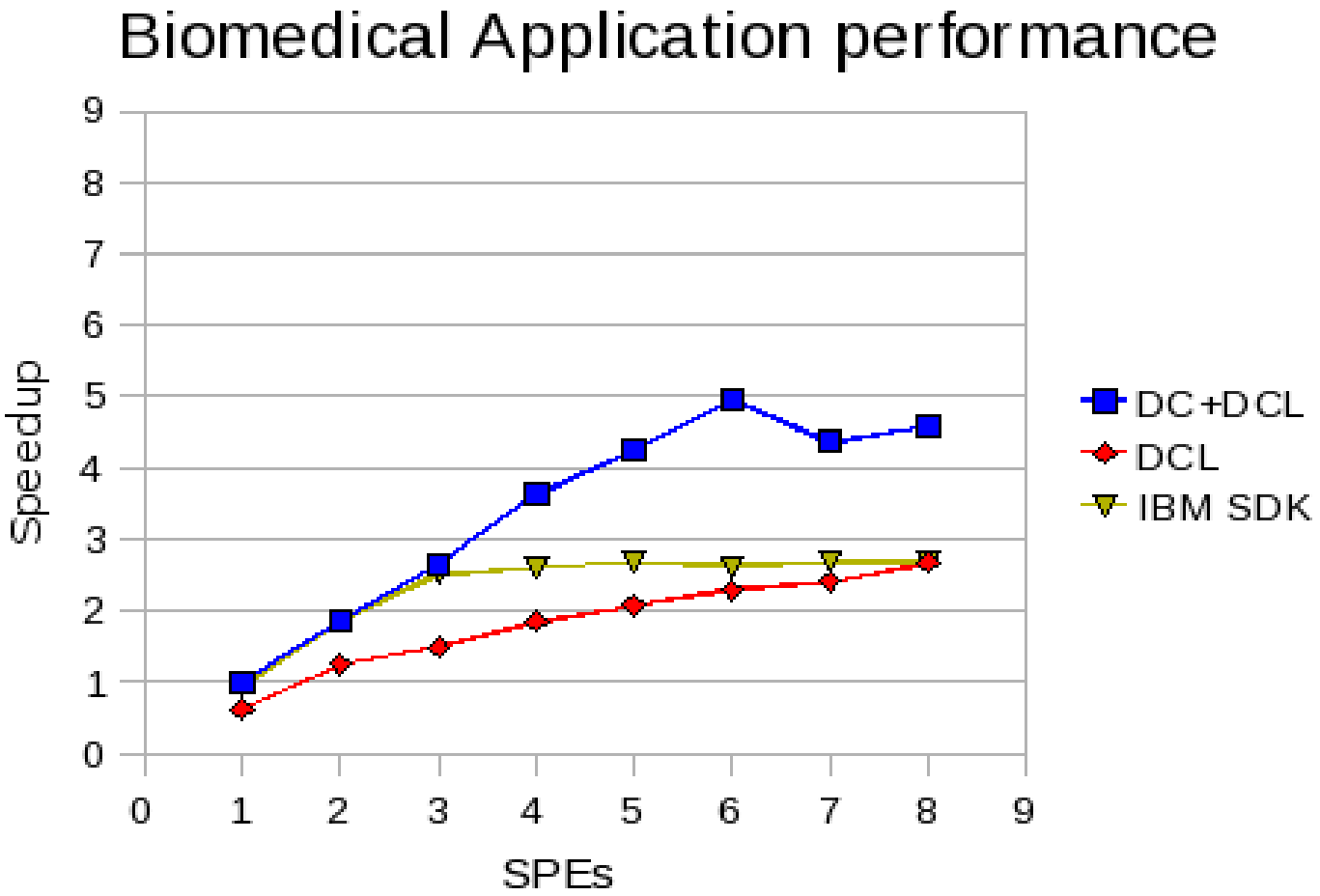
- Compared against custom IBM SDK version
- 32 1Kx1K image tiles
- Overheads included: DCL takes 23-57% longer
- Overheads excluded: SDK takes 5-26% longer

Mixed Granularity DataCutter Example

- DataCutter for coarse-grained parallelism
- DCL for fine-grained parallelism



Biomedical Application Performance (2)



- 1024 1Kx1K image tiles
- DC+DCL has up 42% shorter execution time

Conclusions Future Work

- Contributions
  - Two-sided communication library (CIML)
  - Filter-stream programming framework and runtime engine (DataCutter-Lite)
- Conclusions
  - CIML and DCL give good performance with easier programming than raw IBM SDK
- Future work
  - Extend fine-grained filter-stream framework to CMP, GPU
  - Automate trial-and-error fine-tuning
  - Simplify placement/sizing of filter instances with performance modeling

Related Work

- MPI-like
  - MPI u-tasks – IBM Research
  - Cell Messaging Layer (CML) - LANL
- Block-based
  - BlockLib
  - Sequoia - Stanford
  - Charm++ - UIUC
  - Accelerated Library Framework (ALF) – IBM SDK
- Source compilers
  - CellSs - BSC
- Streaming frameworks
  - StreamIt – MIT

DCL Code Examples

- PPE Code
  - main()
  - setup_application()
  - filter function
- SPE Code
  - filter function

```
// Omitted: Set up Matrices A, B, pointers, a_ptr,
// b_ptr, constants
int main(int argc, char ** argv) {
 init_dcl();

 for (i = 0; i < NUM_ROWS; i++) {
  DCLBuffer * buffer =
   create_buffer("raw_data",
   BUF_SIZE);

  append_array(buffer, a_ptr,
   NUM_COLS * sizeof(float));
  append_array(buffer, b_ptr,
   NUM_COLS * sizeof(float));

  stream_write(buffer);
  // Omitted: increment pointers a_ptr, b_ptr
 }
 finish_dcl();
 return 0;
}
```

DCL Code Examples

- PPE Code
  - main()
  - setup_application()
  - filter function
- SPE Code
  - filter function

```
// PPE setup and filter code
// Called by init_dcl()
void setup_application(Placement * p) {
 Filter * console = get_console(p);
 Filter * fadded = place_ppu_filter(p,
       "added_data");
 Filter * fadder = place_filter(p, 0, "add_values");

 Stream * sraw = add_stream(p, "raw_data");
 add_source(p, sraw, console);
 add_sink(p, sraw, fadder);

 Stream * sadded = add_stream(p,
      "added_matrix");
 add_source(p, sadded, fadder);
 add_sink(p, sadded, fadded);
}
```

DCL Code Examples

- PPE Code
  - main()
  - setup_application()
  - filter function
- SPE Code
  - filter function

```
// When receving a buffer from SPE
void added_data(DCLBuffer * buffer) {
  // Omitted: Deal with added matrix data
}

EVENT_PROVIDE1(added_data);
```

DCL Code Examples

- PPE Code
  - main()
  - setup_application()
  - filter function
- SPE Code
  - filter function

```
// Omitted: Set up constants
void add_values(DCLBuffer * buffer) {
 DCLBuffer * out_buffer = create_buffer(
      "added_matrix", BUF_SIZE);

 float * a = get_float_data_pointer(buffer);
 increment_extract_pointer(buffer,
      num_values * sizeof(float));
 float * b = get_float_data_pointer(buffer);
 float * c = get_float_data_pointer(out_buffer);

 for (i = 0; i < NUM_COLS; i++)
  c[i] = a[i] + b[i];

 stream_write(out_buffer);
}

EVENT_PROVIDE1(add_values);
```