Phloem: Automatic Acceleration of Irregular Applications with Fine-Grain Pipeline Parallelism

Quan M. Nguyen and Daniel Sanchez
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

HPCA 2023
Wednesday, 1 March 2023
Irregular applications difficult to accelerate

• Data-dependent memory accesses and control flow
• Recent hardware support for irregular application pipelines
• **Problem: many ways to map pipelines**

Dynamic simulation
[Image](https://sparse-files-images.engr.tamu.edu/Um/2cubes_sphere_graph.gif)

Graph processing

Sparse deep learning

By Barrett Lyon / The Opte Project
Visualization of the routing paths of the Internet.
Spatial

Core → Core → Core → Core

Temporal

Core

Both!

Core → Core → Core → Core

Core

Core

Core → Core → Core → Core

Core
New tradeoffs call for a new compiler: Phloem

• Temporal pipelines have fundamentally different tradeoffs
• Phloem systematically creates efficient temporal pipelines
• Apply as static transformation or use profile-guided optimization
• Phloem improves performance by 1.7x gmean, 80% of manually tuned code
Agenda

Intro → Background → Phloem → Evaluation
The perils of irregularity

```python
for vtx in vertices:
    for ngh in neighbors[vtx]:
        work(vtx, ngh)
```
As a pipeline (in separate cores)

```python
for vtx in vertices:
    for ngh in neighbors[vtx]:
        work(vtx, ngh)
```

- Enumerate vertices
- Fetch neighbors
- work()
Pipeline

Enumerate vertices → Fetch neighbors → work()

Decoupling
Decoupling is not new, but...

(DAE [ISCA’82], PIPE [ISCA’85], ZS-1 [ASPLOS’87], ACRI-1 [HPCN’95], MT-DCAE [PACT’01], Raw [MICRO’02], Merrimac [SC’03], DSWP [PACT’04, MICRO’05, CGO’10], Outrider [ISCA’11], MPPA [HPEC’13], HELIX [CGO’12, ISCA’14], DeSC [MICRO’15], ...)
Spatial pipelines cause load imbalance

- high outdegree: more work
- low outdegree: less work

Enum vtxs → 1 2 → Fetch neighs → 37 714 61 7 252 → work()
Dynamic temporal pipelines effectively handle irregularity

- Time-division multiplex many stages onto same core or processing element
  - General-purpose core: compute, loads/stores, control flow
  - Decoupled communication between stages on different cores or same cores
  - Core dynamically selects stages to execute

- Tradeoffs for efficient stages fundamentally changes

![Multithreaded Core Diagram]

1. Enumerate
2. Fetch neighbors
3. work()
Making efficient dynamic \textit{temporal} pipelines

- Decouple all long-latency events
  - \textit{spatial: decoupling limited by load imbalance}
- Use dynamic temporal features of hardware to load balance
  - \textit{spatial: requires known communication rates}
    (StreamIt [CC’02])
- Result: many small stages
  - \textit{spatial: few large stages}
    (DSWP [PACT’04,MICRO’05], PS-DSWP [CGO’10], …)
- \textbf{Phloem makes these first-class considerations}
A representative architecture: Pipette [MICRO’20]

• Implements irregular applications as multithreaded programs
• Each stage runs on a thread of a multithreaded core
• Architectural support for cheap, fast inter-thread communication
  • enq(queue, value)
  • deq(queue)
• Reuse simultaneous multithreading to achieve load balance
• Reference accelerators further decouple memory accesses
• Change control flow through special values
Pipelines built manually

def bfs(src):
    ...
    for v in current frontier:
        start, end = offsets[v], offsets[v+1]
        for ngh in neighbors[start:end]:
            dist = distances[ngh]
            if dist is not set:
                set distance; add to next frontier
    ...

Process current frontier
Enumerate neighbors
Visit neighbors
Update data, next frontier
void bfs(Graph* g, int* cur_frontier, int* next_frontier, int root, int* distances)
{
    int cur_frontier_idx = 0, next_frontier_idx = 0;
    int cur_dist = 0;
    // Add root to frontier
    cur_frontier[cur_frontier_idx++] = root;
    distances[root] = 0;
    while (cur_frontier_idx != 0) {
        cur_dist++;
        // Process current frontier
        for (int i = 0; i < cur_frontier_idx; i++) {
            int v = cur_frontier[i];
            // Enumerate neighbors
            int edge_start = g->edges[v];
            int edge_end = g->nodes[v+1];
            for (int e = edge_start; e < edge_end; e++) {
                // Visit neighbor
                int ngh = g->edges[e];
                // If dist decreases, update it,
                // add ngh to next frontier
                int old_dist = distances[ngh];
                if (cur_dist < old_dist) {
                    distances[ngh] = cur_dist;
                    cur_frontier[cur_frontier_idx++] = ngh;
                }
            }
            swap(&cur_frontier, &next_frontier);
            cur_frontier_idx = next_frontier_idx;
        }
    }
}

void bfs_stage1(Graph* g, int* cur_frontier, int* next_frontier, int root, int* distances)
{
    int cur_frontier_idx = 0;
    int cur_dist = 0;
    // Add root to frontier
    cur_frontier[cur_frontier_idx++] = root;
    distances[root] = 0;
    while (cur_frontier_idx != 0) {
        cur_dist++;
        // Process current frontier
        for (int i = 0; i < cur_frontier_idx; i++) {
            int v = cur_frontier[i];
            enq(v, v);
            if (distances[v] >= distances[root]) {
                distances[v] = distances[root];
                enq(ctrl(v, NEXT));
                swap(&cur_frontier, &next_frontier);
                cur_frontier_idx = next_frontier_idx;
            }
        }
    }
}

void bfs_stage2(Graph* g, int* cur_frontier, int* next_frontier, int root, int* distances)
{
    int cur_frontier_idx = 0;
    int cur_dist = 0;
    setup_reference_accelerator(1, INDIRECT, g->nodes);
    setup_control_value_handler(1, &q1_handle_ctrl);
    while (true) {
        while (true) {
            // Enumerate neighbors
            int edge_start = deq(1);
            int edge_end = deq(1);
            for (int e = edge_start; e < edge_end; e++) {
                enq(2, e);
            }
        }
    }
}

void bfs_stage3(Graph* g, int* cur_frontier, int* next_frontier, int root, int* distances)
{
    setup_reference_accelerator(2, INDIRECT, g->nodes);
    setup_control_value_handler(2, &q2_handle_ctrl);
    while (true) {
        if (deq(2) == LAST) {
            enq(3, ngh);
            break;
        }
        enq(3, NEXT);
    }
}

void bfs_stage4(Graph* g, int* cur_frontier, int* next_frontier, int root, int* distances)
{
    setup_reference_accelerator(4, INDIRECT, g->nodes);
    setup_control_value_handler(3, &q3_handle_ctrl);
    while (true) {
        while (true) {
            int ngh = deq(3);
            // If dist decreases, update it,
            // add ngh to next frontier
            int old_dist = ngh;
            if (distances[ngh] > distances[root]) {
                distances[ngh] = distances[root];
                enq(4, ngh);
            }
        }
    }
}

void enq(int v, int e) {
    int dean, enq_status;
    dean = enq(1, v, e);
    enq_status = enq(2, dean);
    switch(enq_status) {
        case 1:
            break;
        case 2:
            break;
        default:
            break;
    }
}

void deq(int v, int e) {
    int dean, deq_status;
    dean = deq(1, v, e);
    deq_status = deq(2, dean);
    switch(deq_status) {
        case 1:
            break;
        case 2:
            break;
        default:
            break;
    }
}

int q1_handle_ctrl(int i, int ngh) {
    if (i == 3) {
        enq(3, ngh);
    }
    return 0;
}

int q2_handle_ctrl(int i, int ngh) {
    if (i == 4) {
        enq(4, ngh);
    }
    return 0;
}

int q3_handle_ctrl(int i, int ngh) {
    if (i == 5) {
        enq(5, ngh);
    }
    return 0;
}

int q4_handle_ctrl(int i, int ngh) {
    if (i == 6) {
        enq(6, ngh);
    }
    return 0;
}

void setup_reference_accelerator(int i, int accel_type, int* nodes) {
    // Setup accelerator
}

void setup_control_value_handler(int i, int* nodes) {
    // Setup control value handler
}

int deq(int i, int v, int e) {
    return 0;
}

int enq(int i, int v, int e) {
    return 0;
}
Agenda

Intro → Background → Phloem → Evaluation
for ngh in g->nodes[\text{edge}\_\text{start}:\text{edge}\_\text{end}]:
...  

```c
def (i = 0; i < cur_frontier_idx; i++) {
    v = cur_frontier[i];
    edge_start = g->nodes[v];
    edge_end = g->nodes[v+1];
    for (e = edge_start; e < edge_end; e++) {
        ngh = g->edges[e];
        ...
    }
}
```
... 

```c
for (i = 0; i < cur_frontier_idx; i++) {
    v = cur_frontier[i];
    edge_start = g->nodes[v];
    edge_end = g->nodes[v+1];
    for (e = edge_start; e < edge_end; e++) {
        ngh = g->edges[e];
        ...
    }
}
```
... 

```c
for (i = 0; i < cur_frontier_idx; i++) {
    edge_start = g->nodes[v];
    edge_end = g->nodes[v+1];
    for (e = edge_start; e < edge_end; e++) {
        ...
    }
}
```
... for (i = 0; i < cur_frontier_idx; i++) {
  edge_start = g->nodes[v];
  edge_end = g->nodes[v+1];
  for (e = edge_start; e < edge_end; e++) {
    ...
  }
}
for (i = 0; i < cur_frontier_idx; i++) {
    v = deq(); v_plus_1 = deq();
    edge_start = g->nodes[v];
    edge_end = g->nodes[v_plus_1];
    for (e = edge_start; e < edge_end; e++) {
        enq(e);
    }
    ...
... 

```c
for (i = 0; i < cur_frontier_idx; i++) {
    v = deq(); v_plus_1 = deq();
    edge_start = g->nodes[v];
    edge_end = g->nodes[v+1];
    for (e = edge_start; e < edge_end; e++) {
        enq(e);
    }

    ... 
```
... for (i = 0; i < cur_frontier_idx; i++) {
    v = deq();
    edge_start = g->nodes[v];
    edge_end = g->nodes[v+1];
    for (e = edge_start; e < edge_end; e++) {
        enq(e);
        ...
    }
}
setup_reference_accelerator();

for (i = 0; i < cur_frontier_idx; i++) {
    v = deq();
    edge_start = g->nodes[v];
    edge_end = g->nodes[v+1];
    for (e = edge_start; e < edge_end; e++) {
        enq(e);
        ...
    }
}
```c
setup_reference_accelerator();
for (i = 0; i < cur_frontier_idx; i++) {
    edge_start = deq();
    edge_end = deq();
    for (e = edge_start; e < edge_end; e++) {
        enq(e);
        ...
    }
}
```
setup_reference_accelerator();

while (true) {
    edge_start = deq();
    edge_end = deq();
    for (e = edge_start; e < edge_end; e++) {
        enq(e);
    } } enq_ctrl(NEXT);
```c
setup_reference_accelerator();
setup_control_value_handler(&&handle_ctrl);
while (true) {
    edge_start = deq();
    edge_end = deq();
    for (e = edge_start; e < edge_end; e++) {
        enq(e);
    }
}
handle_ctrl:
    deq();
enq_ctrl(NEXT);
```
Review of Phloem transformations

1. Add queues
2. Recompute values
3. Accelerate memory accesses
4. Use control values
5. Use control handlers
6. Inter-stage dead code elimination

(Handling races and aliasing)

(Reducing unnecessary communication)
Phloem pipeline pipeline

- **Static compilation flow**: single pipeline
- **Profile-guided optimization mode**: search for best pipeline

Decoupling points:

Score:

Load:

old_dist = distances[ edge ]
edge = g->edges[i]
edge_start = g->nodes[current]
edge_end = g->nodes[current+1]
...

Cost model

Identify irregular loads

C code

Generate candidate decouplings

Main passes

Evaluate

Training inputs

Select best pipeline

Evaluate

Evaluate

Evaluate
See paper for more

• Handling race conditions and aliasing
• Automatically offloading chains of memory accesses
• Static and profile-guided cost models
• Energy results and breakdowns
• Sweeps on pipeline length
Agenda

Intro → Background → Phloem → Evaluation
Methodology

- Event-driven cycle simulation based on ZSim
- 4-way simultaneous multithreaded OOO core with 6-wide issue (similar to Intel Skylake)
- Comparison systems:
  - Baseline: serial OOO core
  - Data-parallel 4-way multithreaded
  - Manually pipelined version
- Applications evaluated: BFS, Connected Components, PageRank-Delta, Radii estimation, SpMSpM
Performance similar to hand-tuned code

- Serial
- Data-parallel
- Phloem
- Manually pipelined

Speedup

Normalized cycles

BFS  CC  PRD  Radii  SpMSpM

Idle  Queue  full/empty  Stalls  Issued
#pragma phloem

```
int ngh = g->edges[e];
```

#pragma phloem distribute

```
int old_dist = distances[ngh];
```

// set up replica 0 frontier
// set up replica 1 frontier

...
```c
int ngh = g->edges[e];
#pragma phloem distribute
int old_dist = distances[ngh];
```
Extending domain-specific languages

\[ y_i = \sum_j A_{ij} \times x_j \]
\[ y(i) = A(i, j) \times x(j) \]

Tensor Algebra Compiler (taco)

C code

Phloem

![Diagram showing the process of extending domain-specific languages](image)

![Bar chart showing speedup improvements](chart)
Summary and conclusion

• Emerging hardware support for building fine-grain pipelines from irregular applications changes the tradeoffs for efficient pipelines
• Phloem systematizes compiling irregular applications into pipelines
• Fast static mode and comprehensive profile-guided mode
• Achieves gmean 1.7x speedup, 80% of manual performance
• Makes state-of-the-art hardware support accessible to all
Thank you!

Phloem: Automatic Acceleration of Irregular Applications with Fine-Grain Pipeline Parallelism

Quan M. Nguyen and Daniel Sanchez
qmn@csail.mit.edu and sanchez@csail.mit.edu

HPCA 2023
Wednesday, 1 March 2023