Work Stealing Scheduler for Automatic Parallelization in FAUST

Linux Audio Conference

S. Letz, Y. Orlarey, D. Fober

GRAME
Centre national de création musicale

Utrecht, May 2 2010

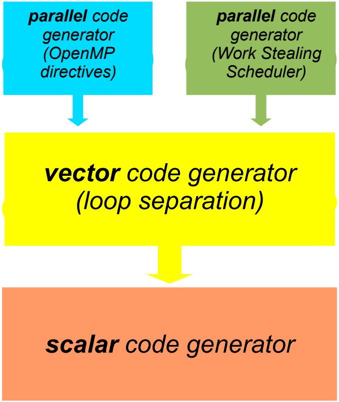
Outline

1. Work Stealing Scheduler

2. Demo

FAUST compilation stack

New Work Stealing Scheduler



FAUST compilation stack

How is the code generated?

- Scalar code is compiled as a unique big loop
- Vectorized code is compiled as separated smaller loops communicating with vectors
- Parallel code executes the graph of loops (= tasks) in parallel

How is the code generated?

- Scalar code is compiled as a unique big loop
- Vectorized code is compiled as separated smaller loops communicating with vectors
- Parallel code executes the graph of loops (= tasks) in parallel

How is the code generated?

- Scalar code is compiled as a unique big loop
- Vectorized code is compiled as separated smaller loops communicating with vectors
- Parallel code executes the graph of loops (= tasks) in parallel

How is the code generated?

- Scalar code is compiled as a unique big loop
- Vectorized code is compiled as separated smaller loops communicating with vectors
- Parallel code executes the graph of loops (= tasks) in parallel

FAUST compilation stack

How is the code generated?

- Scalar code is compiled as a unique big loop
- Vectorized code is compiled as separated smaller loops communicating with vectors
- Parallel code executes the graph of loops (= tasks) in parallel

Parallelizing the DAG

The computation DAG

- Tasks are organized as a Direct Acyclic Graph
- The graph is executed on a "vector size" that can be less or equal to callback buffer size
- Input buffers are consumed and output buffers are produced

Parallelizing the DAG

The computation DAG

- Tasks are organized as a Direct Acyclic Graph
- The graph is executed on a "vector size" that can be less or equal to callback buffer size
- Input buffers are consumed and output buffers are produced

Parallelizing the DAG

The computation DAG

- Tasks are organized as a Direct Acyclic Graph
- The graph is executed on a "vector size" that can be less or equal to callback buffer size
- Input buffers are consumed and output buffers are produced

Parallelizing the DAG

The computation DAG

- Tasks are organized as a Direct Acyclic Graph
- The graph is executed on a "vector size" that can be less or equal to callback buffer size
- Input buffers are consumed and output buffers are produced

Parallelizing the DAG

The computation DAG

- Tasks are organized as a Direct Acyclic Graph
- The graph is executed on a "vector size" that can be less or equal to callback buffer size
- Input buffers are consumed and output buffers are produced

Parallelizing the DAG

Using OpenMP

- The DAG is sorted to express a sequence of parallel group of tasks
- OpenMP pragmas are then added at appropriate location
- Synchronization points between parallel sections

Parallelizing the DAG

Using OpenMP

- The DAG is sorted to express a sequence of parallel group of tasks
- OpenMP pragmas are then added at appropriate location
- Synchronization points between parallel sections

Parallelizing the DAG

Using OpenMP

- The DAG is sorted to express a sequence of parallel group of tasks
- OpenMP pragmas are then added at appropriate location
- Synchronization points between parallel sections

Parallelizing the DAG

Using OpenMP

- The DAG is sorted to express a sequence of parallel group of tasks
- OpenMP pragmas are then added at appropriate location
- Synchronization points between parallel sections

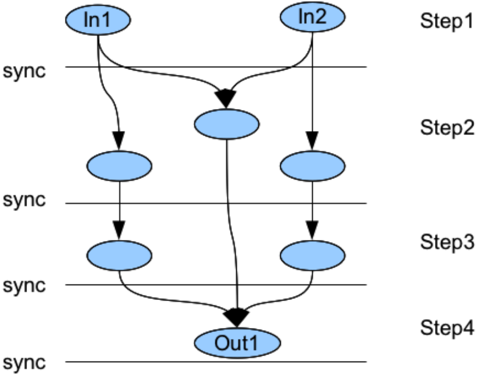
Parallelizing the DAG

Using OpenMP

- The DAG is sorted to express a sequence of parallel group of tasks
- OpenMP pragmas are then added at appropriate location
- Synchronization points between parallel sections

Parallelizing the DAG

OpenMP model



Parallelizing the DAG

```
#pragma omp parallel
for (int index = 0; index < fullcount; index += 32) {
    int count = min (32, fullcount-index);
    FAUSTFLOAT* input0 = &input[0][index];
    FAUSTFLOAT* input1 = &input[1][index];
    FAUSTFLOAT* output0 = &output[0][index];
    FAUSTFLOAT* output1 = &output[1][index];


    // SECTION : 1
    #pragma omp sections
    {
        #pragma omp section
        {
            // DSP code
        }
        #pragma omp section
        {
            // DSP code
        }
    }

    // SECTION : 2
    #pragma omp sections
    {
        #pragma omp section
        {
            // DSP code
        }
        #pragma omp section
        {
            // DSP code
        }
        #pragma omp section
        {
            // DSP code
        }
    }

    // SECTION : 3
    #pragma omp sections
    {
        #pragma omp section
        {
            // DSP code
        }
        #pragma omp section
        {
            // DSP code
        }
    }

    // SECTION : 4
    #pragma omp single
    {
        // DSP code
    }

}
```

Parallelizing the DAG

OpenMP performances

- Works quite well with Intel icc compiler
- But not so well with gcc... (even not at all on OSX)
- Expressed parallelism is not optimal (too much synchronization points...)

Parallelizing the DAG

OpenMP performances

- Works quite well with Intel icc compiler
- But not so well with gcc... (even not at all on OSX)
- Expressed parallelism is not optimal (too much synchronization points...)

OpenMP performances

- Works quite well with Intel icc compiler
- But not so well with gcc... (even not at all on OSX)
- Expressed parallelism is not optimal (too much synchronization points...)

Parallelizing the DAG

OpenMP performances

- Works quite well with Intel icc compiler
- But not so well with gcc... (even not at all on OSX)
- Expressed parallelism is not optimal (too much synchronization points...)

Parallelizing the DAG

OpenMP performances

- Works quite well with Intel icc compiler
- But not so well with gcc... (even not at all on OSX)
- Expressed parallelism is not optimal (too much synchronization points...)

Explicit graph scheduling

Data-flow model

- Input tasks are ready to be executed
- Activations go from input to output following data dependencies links
- A given task can be executed when it's inputs have been executed
- We want to minimize the global execution time

Explicit graph scheduling

Data-flow model

- Input tasks are ready to be executed
- Activations go from input to output following data dependencies links
- A given task can be executed when it's inputs have been executed
- We want to minimize the global execution time

Explicit graph scheduling

Data-flow model

- Input tasks are ready to be executed
- Activations go from input to output following data dependencies links
- A given task can be executed when it's inputs have been executed
- We want to minimize the global execution time

Explicit graph scheduling

Data-flow model

- Input tasks are ready to be executed
- Activations go from input to output following data dependencies links
- A given task can be executed when it's inputs have been executed
- We want to minimize the global execution time

Explicit graph scheduling

Data-flow model

- Input tasks are ready to be executed
- Activations go from input to output following data dependencies links
- A given task can be executed when it's inputs have been executed
- We want to minimize the global execution time

Explicit graph scheduling

Data-flow model

- Input tasks are ready to be executed
- Activations go from input to output following data dependencies links
- A given task can be executed when it's inputs have been executed
- We want to minimize the global execution time

Static versus dynamic graph scheduling

Scheduling models

- A lot has been written on the subject
- Basically two different approaches:
- - static scheduling: finding a "mapping" of tasks on the set of cores before actual execution
- - dynamic scheduling: doing the "mapping" at runtime

Static versus dynamic graph scheduling

Scheduling models

- A lot has been written on the subject
- Basically two different approaches:
- - static scheduling: finding a "mapping" of tasks on the set of cores before actual execution
- - dynamic scheduling: doing the "mapping" at runtime

Static versus dynamic graph scheduling

Scheduling models

- A lot has been written on the subject
- Basically two different approaches:
- - static scheduling: finding a "mapping" of tasks on the set of cores before actual execution
- - dynamic scheduling: doing the "mapping" at runtime

Static versus dynamic graph scheduling

Scheduling models

- A lot has been written on the subject
- Basically two different approaches:
- - static scheduling: finding a "mapping" of tasks on the set of cores before actual execution
- - dynamic scheduling: doing the "mapping" at runtime

Static versus dynamic graph scheduling

Scheduling models

- A lot has been written on the subject
- Basically two different approaches:
- - static scheduling: finding a "mapping" of tasks on the set of cores before actual execution
- - dynamic scheduling: doing the "mapping" at runtime

Static versus dynamic graph scheduling

Scheduling models

- A lot has been written on the subject
- Basically two different approaches:
- - static scheduling: finding a "mapping" of tasks on the set of cores before actual execution
- - dynamic scheduling: doing the "mapping" at runtime

Static scheduling

Static model

- Usually requires that the cost of task execution and communication time in known in advance
- More of theoretical interest

Static model

- Usually requires that the cost of task execution and communication time in known in advance
- More of theoretical interest

Static scheduling

Static model

- Usually requires that the cost of task execution and communication time in known in advance
- More of theoretical interest

Static scheduling

Static model

- Usually requires that the cost of task execution and communication time in known in advance
- More of theoretical interest

Dynamic scheduling

Dynamic model

- Choice of task execution done at runtime
- A set of worker threads to execute tasks
- Worker threads have to find ready tasks, execute them and propagate "activations"

Dynamic scheduling

Dynamic model

- Choice of task execution done at runtime
- A set of worker threads to execute tasks
- Worker threads have to find ready tasks, execute them and propagate "activations"

Dynamic scheduling

Dynamic model

- Choice of task execution done at runtime
- A set of worker threads to execute tasks
- Worker threads have to find ready tasks, execute them and propagate "activations"

Dynamic scheduling

Dynamic model

- Choice of task execution done at runtime
- A set of worker threads to execute tasks
- Worker threads have to find ready tasks, execute them and propagate "activations"

Dynamic scheduling

Dynamic model

- Choice of task execution done at runtime
- A set of worker threads to execute tasks
- Worker threads have to find ready tasks, execute them and propagate "activations"

Simple "one global queue" model

Simple "one queue" model

- One global shared queue of "ready" tasks
- When a task is executed, possibly push ready output tasks in the queue
- All idle threads try to Pop ready tasks from the queue
- Needs lock-free access, a lot of contention on the global queue...

Simple "one global queue" model

Simple "one queue" model

- One global shared queue of "ready" tasks
- When a task is executed, possibly push ready output tasks in the queue
- All idle threads try to Pop ready tasks from the queue
- Needs lock-free access, a lot of contention on the global queue...

Simple "one queue" model

- One global shared queue of "ready" tasks
- When a task is executed, possibly push ready output tasks in the queue
- All idle threads try to Pop ready tasks from the queue
- Needs lock-free access, a lot of contention on the global queue...

Simple "one global queue" model

Simple "one queue" model

- One global shared queue of "ready" tasks
- When a task is executed, possibly push ready output tasks in the queue
- All idle threads try to Pop ready tasks from the queue
- Needs lock-free access, a lot of contention on the global queue...

Simple "one global queue" model

Simple "one queue" model

- One global shared queue of "ready" tasks
- When a task is executed, possibly push ready output tasks in the queue
- All idle threads try to Pop ready tasks from the queue
- Needs lock-free access, a lot of contention on the global queue...

Simple "one global queue" model

Simple "one queue" model

- One global shared queue of "ready" tasks
- When a task is executed, possibly push ready output tasks in the queue
- All idle threads try to Pop ready tasks from the queue
- Needs lock-free access, a lot of contention on the global queue...

Work Stealing Scheduler (1)

Principle

- A well known algorithm used for instance in Cilk++ (an extension to C/C++ for multithreaded parallel programs)
- Aims at improving the simple model previously described
- Has some interesting properties useful for fined-grained parallelism

Work Stealing Scheduler (1)

Principle

- A well known algorithm used for instance in Cilk++ (an extension to C/C++ for multithreaded parallel programs)
- Aims at improving the simple model previously described
- Has some interesting properties useful for fined-grained parallelism

Work Stealing Scheduler (1)

Principle

- A well known algorithm used for instance in Cilk++ (an extension to C/C++ for multithreaded parallel programs)
- Aims at improving the simple model previously described
- Has some interesting properties useful for fined-grained parallelism

Principle

- A well known algorithm used for instance in Cilk++ (an extension to C/C++ for multithreaded parallel programs)
- Aims at improving the simple model previously described
- Has some interesting properties useful for fined-grained parallelism

Work Stealing Scheduler (1)

Principle

- A well known algorithm used for instance in Cilk++ (an extension to C/C++ for multithreaded parallel programs)
- Aims at improving the simple model previously described
- Has some interesting properties useful for fined-grained parallelism

Work Stealing Scheduler (2)

Operations

- One queue by thread (considered as "private")
- The Work Stealing Queue has a "private" Push, LIFO Pop and a "public" FIFO Pop operations
- Each thread push ready tasks in it's private queue
- It get ready tasks from it's private queue until empty
- It can then "steal" tasks from other threads using their FIFO Pop operation

Work Stealing Scheduler (2)

Operations

- One queue by thread (considered as "private")
- The Work Stealing Queue has a "private" Push, LIFO Pop and a "public" FIFO Pop operations
- Each thread push ready tasks in it's private queue
- It get ready tasks from it's private queue until empty
- It can then "steal" tasks from other threads using their FIFO Pop operation

Work Stealing Scheduler (2)

Operations

- One queue by thread (considered as "private")
- The Work Stealing Queue has a "private" Push, LIFO Pop and a "public" FIFO Pop operations
- Each thread push ready tasks in it's private queue
- It get ready tasks from it's private queue until empty
- It can then "steal" tasks from other threads using their FIFO Pop operation

Work Stealing Scheduler (2)

Operations

- One queue by thread (considered as "private")
- The Work Stealing Queue has a "private" Push, LIFO Pop and a "public" FIFO Pop operations
- Each thread push ready tasks in it's private queue
- It get ready tasks from it's private queue until empty
- It can then "steal" tasks from other threads using their FIFO Pop operation

Work Stealing Scheduler (2)

Operations

- One queue by thread (considered as "private")
- The Work Stealing Queue has a "private" Push, LIFO Pop and a "public" FIFO Pop operations
- Each thread push ready tasks in it's private queue
- It get ready tasks from it's private queue until empty
- It can then "steal" tasks from other threads using their FIFO Pop operation

Work Stealing Scheduler (2)

Operations

- One queue by thread (considered as "private")
- The Work Stealing Queue has a "private" Push, LIFO Pop and a "public" FIFO Pop operations
- Each thread push ready tasks in it's private queue
- It get ready tasks from it's private queue until empty
- It can then "steal" tasks from other threads using their FIFO Pop operation

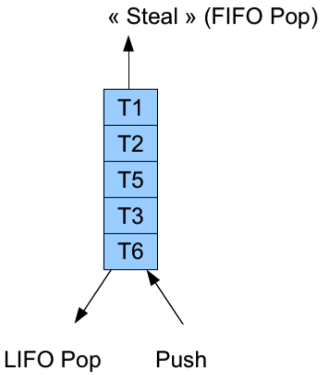
Work Stealing Scheduler (2)

Operations

- One queue by thread (considered as "private")
- The Work Stealing Queue has a "private" Push, LIFO Pop and a "public" FIFO Pop operations
- Each thread push ready tasks in it's private queue
- It get ready tasks from it's private queue until empty
- It can then "steal" tasks from other threads using their FIFO Pop operation

Work Stealing Queue

Operations on the WSQ



Work Stealing Scheduler (3)

Properties

- Less contention since each thread has its own queue
- Each thread can follow a "computation path" until its end, improving cache behaviour
- The "stolen" tasks are the ones pushed first, they are "near the inputs", thus they usually correspond to longer computation path

Work Stealing Scheduler (3)

Properties

- Less contention since each thread has its own queue
- Each thread can follow a "computation path" until its end, improving cache behaviour
- The "stolen" tasks are the ones pushed first, they are "near the inputs", thus they usually correspond to longer computation path

Properties

- Less contention since each thread has its own queue
- Each thread can follow a "computation path" until its end, improving cache behaviour
- The "stolen" tasks are the ones pushed first, they are "near the inputs", thus they usually correspond to longer computation path

Work Stealing Scheduler (3)

Properties

- Less contention since each thread has its own queue
- Each thread can follow a "computation path" until its end, improving cache behaviour
- The "stolen" tasks are the ones pushed first, they are "near the inputs", thus they usually correspond to longer computation path

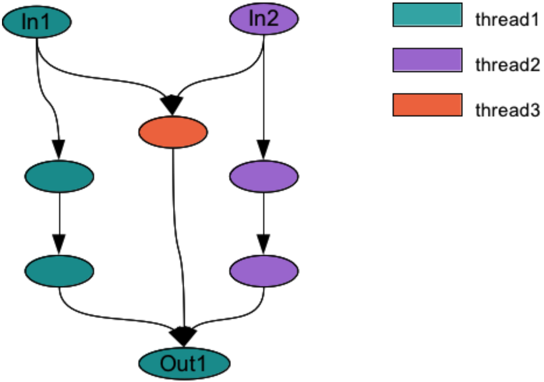
Work Stealing Scheduler (3)

Properties

- Less contention since each thread has its own queue
- Each thread can follow a "computation path" until its end, improving cache behaviour
- The "stolen" tasks are the ones pushed first, they are "near the inputs", thus they usually correspond to longer computation path

Work Stealing Scheduler (4)

Computation path



Compilation of the Work Stealing Scheduler

Why?

- Since the graph is know in at compilation time, the WSS can be compiled and "embedded" in the generated code
- For each task after DSP computation, the code to propagate activations only depends of the graph topology

Compilation of the Work Stealing Scheduler

Why?

- Since the graph is know in at compilation time, the WSS can be compiled and "embedded" in the generated code
- For each task after DSP computation, the code to propagate activations only depends of the graph topology

Compilation of the Work Stealing Scheduler

Why?

- Since the graph is know in at compilation time, the WSS can be compiled and "embedded" in the generated code
- For each task after DSP computation, the code to propagate activations only depends of the graph topology

Why?

- Since the graph is know in at compilation time, the WSS can be compiled and "embedded" in the generated code
- For each task after DSP computation, the code to propagate activations only depends of the graph topology

ComputeThread method

- Tasks are numbered
- For a given task, its "activation" value (number of inputs) is prepared
- The DAG is compiled as a big switch/case block to be executed by each thread
- Each sub-block contains the actual DSP code and the "propagate activations" code
- Two additional tasks are added: a "work stealing" task and an "end task"
- Before entering the switch/case block, ready input tasks are distributed among worker threads

Compilation of the Work Stealing Scheduler (2)

ComputeThread method

- Tasks are numbered
- For a given task, its "activation" value (number of inputs) is prepared
- The DAG is compiled as a big switch/case block to be executed by each thread
- Each sub-block contains the actual DSP code and the "propagate activations" code
- Two additional tasks are added: a "work stealing" task and an "end task"
- Before entering the switch/case block, ready input tasks are distributed among worker threads

ComputeThread method

- Tasks are numbered
- For a given task, its "activation" value (number of inputs) is prepared
- The DAG is compiled as a big switch/case block to be executed by each thread
- Each sub-block contains the actual DSP code and the "propagate activations" code
- Two additional tasks are added: a "work stealing" task and an "end task"
- Before entering the switch/case block, ready input tasks are distributed among worker threads

Compilation of the Work Stealing Scheduler (2)

ComputeThread method

- Tasks are numbered
- For a given task, its "activation" value (number of inputs) is prepared
- The DAG is compiled as a big switch/case block to be executed by each thread
- Each sub-block contains the actual DSP code and the "propagate activations" code
- Two additional tasks are added: a "work stealing" task and an "end task"
- Before entering the switch/case block, ready input tasks are distributed among worker threads

Compilation of the Work Stealing Scheduler (2)

ComputeThread method

- Tasks are numbered
- For a given task, its "activation" value (number of inputs) is prepared
- The DAG is compiled as a big switch/case block to be executed by each thread
- Each sub-block contains the actual DSP code and the "propagate activations" code
- Two additional tasks are added: a "work stealing" task and an "end task"
- Before entering the switch/case block, ready input tasks are distributed among worker threads

ComputeThread method

- Tasks are numbered
- For a given task, its "activation" value (number of inputs) is prepared
- The DAG is compiled as a big switch/case block to be executed by each thread
- Each sub-block contains the actual DSP code and the "propagate activations" code
- Two additional tasks are added: a "work stealing" task and an "end task"
- Before entering the switch/case block, ready input tasks are distributed among worker threads

Compilation of the Work Stealing Scheduler (2)

ComputeThread method

- Tasks are numbered
- For a given task, its "activation" value (number of inputs) is prepared
- The DAG is compiled as a big switch/case block to be executed by each thread
- Each sub-block contains the actual DSP code and the "propagate activations" code
- Two additional tasks are added: a "work stealing" task and an "end task"
- Before entering the switch/case block, ready input tasks are distributed among worker threads

Compilation of the Work Stealing Scheduler (2)

ComputeThread method

- Tasks are numbered
- For a given task, its "activation" value (number of inputs) is prepared
- The DAG is compiled as a big switch/case block to be executed by each thread
- Each sub-block contains the actual DSP code and the "propagate activations" code
- Two additional tasks are added: a "work stealing" task and an "end task"
- Before entering the switch/case block, ready input tasks are distributed among worker threads

ComputeThread method

```
void computeThread(int cur_thread) {
{
    TaskQueue taskqueue(cur_thread);
    int tasknum = -1;
    int count = fFullCount;
    // Init input and output
    FAUSTFLOAT* output0 = &output[0][fIndex];
    FAUSTFLOAT* output1 = &output[1][fIndex];
    int task_list_size = 2;
    int task_list[2] = {2,3};
    taskqueue.InitTaskList(task_list_size, task_list, fDynamicNumThreads, cur_thread, tasknum);
    while (!fIsFinished) {
        switch (tasknum) {

            case 2: {
                // TASK code
            }
            case 3: {
                // TASK code
            }
            ....
        }
    }
}
```

Compilation of the Work Stealing Scheduler (3)

Activation code for each connection type

- When possible a task is chosen at the "direct" output
- Ready tasks are Pushed into private WSQ
- Atomic decrement the activation counter of output tasks with several inputs (possibly getting one to execute...)
- Otherwise WORK_STEALING_INDEX is returned and Work Stealing task will be executed

Compilation of the Work Stealing Scheduler (3)

Activation code for each connection type

- When possible a task is chosen at the "direct" output
- Ready tasks are Pushed into private WSQ
- Atomic decrement the activation counter of output tasks with several inputs (possibly getting one to execute...)
- Otherwise WORK_STEALING_INDEX is returned and Work Stealing task will be executed

Compilation of the Work Stealing Scheduler (3)

Activation code for each connection type

- When possible a task is chosen at the "direct" output
- Ready tasks are Pushed into private WSQ
- Atomic decrement the activation counter of output tasks with several inputs (possibly getting one to execute...)
- Otherwise WORK_STEALING_INDEX is returned and Work Stealing task will be executed

Compilation of the Work Stealing Scheduler (3)

Activation code for each connection type

- When possible a task is chosen at the "direct" output
- Ready tasks are Pushed into private WSQ
- Atomic decrement the activation counter of output tasks with several inputs (possibly getting one to execute...)
- Otherwise WORK_STEALING_INDEX is returned and Work Stealing task will be executed

Compilation of the Work Stealing Scheduler (3)

Activation code for each connection type

- When possible a task is chosen at the "direct" output
- Ready tasks are Pushed into private WSQ
- Atomic decrement the activation counter of output tasks with several inputs (possibly getting one to execute...)
- Otherwise WORK_STEALING_INDEX is returned and Work Stealing task will be executed

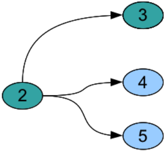
Compilation of the Work Stealing Scheduler (3)

Activation code for each connection type

- When possible a task is chosen at the "direct" output
- Ready tasks are Pushed into private WSQ
- Atomic decrement the activation counter of output tasks with several inputs (possibly getting one to execute...)
- Otherwise WORK_STEALING_INDEX is returned and Work Stealing task will be executed

Compiling tasks activation code

Several outputs tasks (without other inputs)



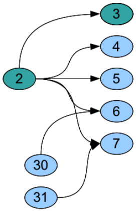

```
case 2: {

    // DSP code

    // Output tasks activation code
    taskqueue.PushHead(4);
    taskqueue.PushHead(5);
    tasknum = 3;
    break;
}
```

Compiling tasks activation code

Several outputs tasks (some without other inputs, some with other inputs)



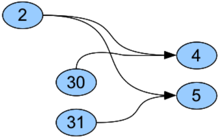

```
case 2: {

    // DSP code

    // Output tasks activation code
    fGraph.ActivateOutputTask(taskqueue, 6);
    fGraph.ActivateOutputTask(taskqueue, 7);
    taskqueue.PushHead(4);
    taskqueue.PushHead(5);
    tasknum = 3;
    break;
}
```

Compiling tasks activation code

Several outputs tasks (with other inputs)




```
case 2: {

    // DSP code

    // Output tasks activation code
    tasknum = WORK_STEALING_INDEX;
    fGraph.ActivateOutputTask(taskqueue, 4, tasknum);
    fGraph.ActivateOutputTask(taskqueue, 5, tasknum);
    fGraph.GetReadyTask(taskqueue, tasknum);
    break;
}
```

Compilation of the Work Stealing Scheduler (4)

Special tasks

- Work Stealing task aims to find a ready task in other threads (possibly "busy-looping")
- All output of the DAG are connected to the "end task"
- When executed, end task returns from the thread

Compilation of the Work Stealing Scheduler (4)

Special tasks

- Work Stealing task aims to find a ready task in other threads (possibly "busy-looping")
- All output of the DAG are connected to the "end task"
- When executed, end task returns from the thread

Compilation of the Work Stealing Scheduler (4)

Special tasks

- Work Stealing task aims to find a ready task in other threads (possibly "busy-looping")
- All output of the DAG are connected to the "end task"
- When executed, end task returns from the thread

Compilation of the Work Stealing Scheduler (4)

Special tasks

- Work Stealing task aims to find a ready task in other threads (possibly "busy-looping")
- All output of the DAG are connected to the "end task"
- When executed, end task returns from the thread

Compilation of the Work Stealing Scheduler (4)

Special tasks

- Work Stealing task aims to find a ready task in other threads (possibly "busy-looping")
- All output of the DAG are connected to the "end task"
- When executed, end task returns from the thread

Special tasks

Work stealing and end tasks

```
case WORK_STEALING_INDEX: {
    tasknum = TaskQueue::GetNextTask(cur_thread, fDynamicNumThreads);
    break;
}

case LAST_TASK_INDEX: {
    fIsFinished = true;
    break;
}
```

Compilation of the Work Stealing Scheduler (5)

Compute method

- Called by "master thread"
- Init graph state (activations)
- Wakes up worker threads, also participates
- After computation, synchronization code to wait for all worker threads to finish

Compilation of the Work Stealing Scheduler (5)

Compute method

- Called by "master thread"
- Init graph state (activations)
- Wakes up worker threads, also participates
- After computation, synchronization code to wait for all worker threads to finish

Compute method

- Called by "master thread"
- Init graph state (activations)
- Wakes up worker threads, also participates
- After computation, synchronization code to wait for all worker threads to finish

Compilation of the Work Stealing Scheduler (5)

Compute method

- Called by "master thread"
- Init graph state (activations)
- Wakes up worker threads, also participates
- After computation, synchronization code to wait for all worker threads to finish

Compilation of the Work Stealing Scheduler (5)

Compute method

- Called by "master thread"
- Init graph state (activations)
- Wakes up worker threads, also participates
- After computation, synchronization code to wait for all worker threads to finish

Compute method

- Called by "master thread"
- Init graph state (activations)
- Wakes up worker threads, also participates
- After computation, synchronization code to wait for all worker threads to finish

Compute method

Called by "master " thread

```
virtual void compute (int fullcount, FAUSTFLOAT** input, FAUSTFLOAT** output) {
    this->input = input;
    this->output = output;
    for (fIndex = 0; fIndex < fullcount; fIndex += 32) {
        fFullCount = min (32, fullcount-fIndex);
        TaskQueue::Init();
        // Initialize end task, if more than one input
        fGraph.InitTask(1,2);
        // Only initialize taks with more than one input
        fGraph.InitTask(19,8);
        fGraph.InitTask(20,8);
        fIsFinished = false;
        fThreadPool->SignalAll(fDynamicNumThreads - 1, this);
        computeThread(0);
        while (!fThreadPool->IsFinished()) {}
    }
}
```

Init method

- Creates worker threads, put them in sleep mode
- Worker threads will inherit "compute method" thread scheduling properties and priorities

Compilation of the Work Stealing Scheduler (6)

Init method

- Creates worker threads, put them in sleep mode
- Worker threads will inherit "compute method" thread scheduling properties and priorities

Compilation of the Work Stealing Scheduler (6)

Init method

- Creates worker threads, put them in sleep mode
- Worker threads will inherit "compute method" thread scheduling properties and priorities

Compilation of the Work Stealing Scheduler (6)

Init method

- Creates worker threads, put them in sleep mode
- Worker threads will inherit "compute method" thread scheduling properties and priorities

More parallelism (1)?

Pipelining

- Some graph are sequential by nature
- Pipelining idea : duplicating each task several times
- Connecting with the appropriate outputs
- Each "sub-task" to be run on a slice of the buffer
- Recursive and non-recursive tasks are treated differently
- Still to be tested...

More parallelism (1)?

Pipelining

- Some graph are sequential by nature
- Pipelining idea : duplicating each task several times
- Connecting with the appropriate outputs
- Each "sub-task" to be run on a slice of the buffer
- Recursive and non-recursive tasks are treated differently
- Still to be tested...

More parallelism (1)?

Pipelining

- Some graph are sequential by nature
- Pipelining idea : duplicating each task several times
- Connecting with the appropriate outputs
- Each "sub-task" to be run on a slice of the buffer
- Recursive and non-recursive tasks are treated differently
- Still to be tested...

More parallelism (1)?

Pipelining

- Some graph are sequential by nature
- Pipelining idea : duplicating each task several times
- Connecting with the appropriate outputs
- Each "sub-task" to be run on a slice of the buffer
- Recursive and non-recursive tasks are treated differently
- Still to be tested...

More parallelism (1)?

Pipelining

- Some graph are sequential by nature
- Pipelining idea : duplicating each task several times
- Connecting with the appropriate outputs
- Each "sub-task" to be run on a slice of the buffer
- Recursive and non-recursive tasks are treated differently
- Still to be tested...

More parallelism (1)?

Pipelining

- Some graph are sequential by nature
- Pipelining idea : duplicating each task several times
- Connecting with the appropriate outputs
- Each "sub-task" to be run on a slice of the buffer
- Recursive and non-recursive tasks are treated differently
- Still to be tested...

More parallelism (1)?

Pipelining

- Some graph are sequential by nature
- Pipelining idea : duplicating each task several times
- Connecting with the appropriate outputs
- Each "sub-task" to be run on a slice of the buffer
- Recursive and non-recursive tasks are treated differently
- Still to be tested...

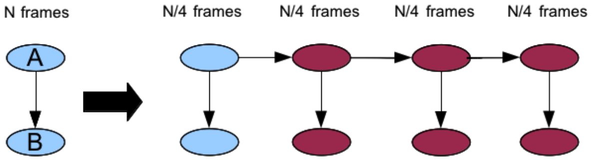
More parallelism (1)?

Pipelining

- Some graph are sequential by nature
- Pipelining idea : duplicating each task several times
- Connecting with the appropriate outputs
- Each "sub-task" to be run on a slice of the buffer
- Recursive and non-recursive tasks are treated differently
- Still to be tested...

Pipeling the graph

Example of graph rewriting



More parallelism (2)?

Bottleneck-task duplication ?

- Some tasks have "bottleneck" behaviour
- Could be interesting to just duplicate them, (executing them several times in different threads...)
- Less synchronization points, thus better global results
- Need to find a proper a method to find out those tasks
- More generally we need to explore "graph rewriting" techniques

More parallelism (2)?

Bottleneck-task duplication ?

- Some tasks have "bottleneck" behaviour
- Could be interesting to just duplicate them, (executing them several times in different threads...)
- Less synchronization points, thus better global results
- Need to find a proper a method to find out those tasks
- More generally we need to explore "graph rewriting" techniques

More parallelism (2)?

Bottleneck-task duplication ?

- Some tasks have "bottleneck" behaviour
- Could be interesting to just duplicate them, (executing them several times in different threads...)
- Less synchronization points, thus better global results
- Need to find a proper a method to find out those tasks
- More generally we need to explore "graph rewriting" techniques

More parallelism (2)?

Bottleneck-task duplication ?

- Some tasks have "bottleneck" behaviour
- Could be interesting to just duplicate them, (executing them several times in different threads...)
- Less synchronization points, thus better global results
- Need to find a proper a method to find out those tasks
- More generally we need to explore "graph rewriting" techniques

More parallelism (2)?

Bottleneck-task duplication ?

- Some tasks have "bottleneck" behaviour
- Could be interesting to just duplicate them, (executing them several times in different threads...)
- Less synchronization points, thus better global results
- Need to find a proper a method to find out those tasks
- More generally we need to explore "graph rewriting" techniques

More parallelism (2)?

Bottleneck-task duplication ?

- Some tasks have "bottleneck" behaviour
- Could be interesting to just duplicate them, (executing them several times in different threads...)
- Less synchronization points, thus better global results
- Need to find a proper a method to find out those tasks
- More generally we need to explore "graph rewriting" techniques

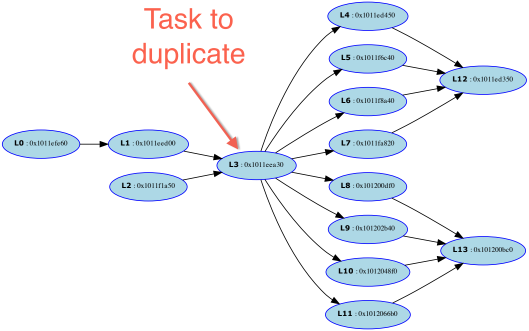
More parallelism (2)?

Bottleneck-task duplication ?

- Some tasks have "bottleneck" behaviour
- Could be interesting to just duplicate them, (executing them several times in different threads...)
- Less synchronization points, thus better global results
- Need to find a proper a method to find out those tasks
- More generally we need to explore "graph rewriting" techniques

Bottleneck task duplication

Example of Karplus8 graph



Benchmarks (1)

Finding where time is spent

- Estimating "busy-loop" cost (in the order of 30-50 usec on Sampo Combo organ run with 4 threads)
- Possibly yiedling if waiting for too long (with a configurable parameter, similar to icc OpenMP KMP_BLOCKTIME)
- Estimating worker threads wake up time (in the order of 10-30 usec on 2 Ghz 4 cores OSX machine)

Finding where time is spent

- Estimating "busy-loop" cost (in the order of 30-50 usec on Sampo Combo organ run with 4 threads)
- Possibly yiedling if waiting for too long (with a configurable parameter, similar to icc OpenMP KMP_BLOCKTIME)
- Estimating worker threads wake up time (in the order of 10-30 usec on 2 Ghz 4 cores OSX machine)

Benchmarks (1)

Finding where time is spent

- Estimating "busy-loop" cost (in the order of 30-50 usec on Sampo Combo organ run with 4 threads)
- Possibly yiedling if waiting for too long (with a configurable parameter, similar to icc OpenMP KMP_BLOCKTIME)
- Estimating worker threads wake up time (in the order of 10-30 usec on 2 Ghz 4 cores OSX machine)

Benchmarks (1)

Finding where time is spent

- Estimating "busy-loop" cost (in the order of 30-50 usec on Sampo Combo organ run with 4 threads)
- Possibly yiedling if waiting for too long (with a configurable parameter, similar to icc OpenMP KMP_BLOCKTIME)
- Estimating worker threads wake up time (in the order of 10-30 usec on 2 Ghz 4 cores OSX machine)

Benchmarks (1)

Finding where time is spent

- Estimating "busy-loop" cost (in the order of 30-50 usec on Sampo Combo organ run with 4 threads)
- Possibly yiedling if waiting for too long (with a configurable parameter, similar to icc OpenMP KMP_BLOCKTIME)
- Estimating worker threads wake up time (in the order of 10-30 usec on 2 Ghz 4 cores OSX machine)

Benchmarks (2)

Simple FAUST examples

- Usually do not benefit from parallelization or marginally
- See Yann workshop from yesterday...

Benchmarks (2)

Simple FAUST examples

- Usually do not benefit from parallelization or marginally
- See Yann workshop from yesterday...

Simple FAUST examples

- Usually do not benefit from parallelization or marginally
- See Yann workshop from yesterday...

Benchmarks (2)

Simple FAUST examples

- Usually do not benefit from parallelization or marginally
- See Yann workshop from yesterday...

Benchmarks (3)

WSS versus OpenMP

- Comparable with icc OpenMP, even better in some cases
- Much better than gcc OpenMP
- Better, finer control of threading behaviour especially in RT context (starting/stopping threads, maximum busy-time value...)

Benchmarks (3)

WSS versus OpenMP

- Comparable with icc OpenMP, even better in some cases
- Much better than gcc OpenMP
- Better, finer control of threading behaviour especially in RT context (starting/stopping threads, maximum busy-time value...)

Benchmarks (3)

WSS versus OpenMP

- Comparable with icc OpenMP, even better in some cases
- Much better than gcc OpenMP
- Better, finer control of threading behaviour especially in RT context (starting/stopping threads, maximum busy-time value...)

Benchmarks (3)

WSS versus OpenMP

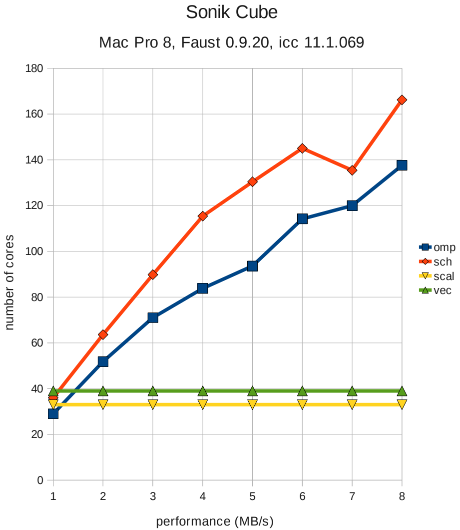
- Comparable with icc OpenMP, even better in some cases
- Much better than gcc OpenMP
- Better, finer control of threading behaviour especially in RT context (starting/stopping threads, maximum busy-time value...)

Benchmarks (3)

WSS versus OpenMP

- Comparable with icc OpenMP, even better in some cases
- Much better than gcc OpenMP
- Better, finer control of threading behaviour especially in RT context (starting/stopping threads, maximum busy-time value...)

Sonik Cube


Sonik Cube
Mac Pro 8, Faust 0.9.20, icc 11.1.069
number of cores
performance (MB/s)
0
20
40
60
80
100
120
140
160
180
1
2
3
4
5
6
7
8
omp
sch
scal
vec

Benchmarks (4)

One well working example : the famous 8 min compilation time, 50% CPU usage...

- Sampo YC20 Combo Organ
- Compiled with llvm-g++-4.2 on OSX MacPro 4 cores 2 GHz machine (gcc 4.2 compilation is way too slow...)

One well working example : the famous 8 min compilation time, 50% CPU usage...

- Sampo YC20 Combo Organ
- Compiled with llvm-g++-4.2 on OSX MacPro 4 cores 2 GHz machine (gcc 4.2 compilation is way too slow...)

One well working example : the famous 8 min compilation time, 50% CPU usage...

- Sampo YC20 Combo Organ
- Compiled with llvm-g++-4.2 on OSX MacPro 4 cores 2 GHz machine (gcc 4.2 compilation is way too slow...)

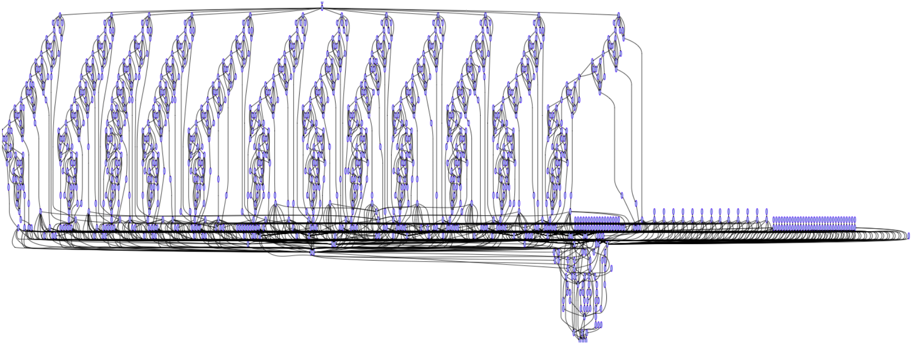
Benchmarks (4)

One well working example : the famous 8 min compilation time, 50% CPU usage...

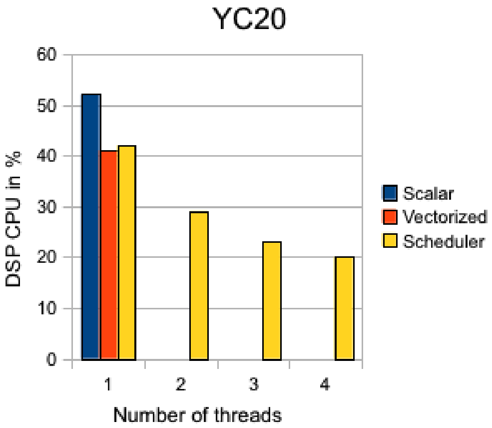
- Sampo YC20 Combo Organ
- Compiled with llvm-g++-4.2 on OSX MacPro 4 cores 2 GHz machine (gcc 4.2 compilation is way too slow...)

YC20

Graph of 991 tasks...



At 128 frames, 48 kHz with JACK : with 4 threads, 2 times faster than vectorized mode, 2.5 faster than scalar mode



Known issues (1)

Limits of the current approach

- Code size, compilers may fail to compile it...
- Threading issues: too much threads for the available cores, combining FAUST parallel modules...

Known issues (1)

Limits of the current approach

- Code size, compilers may fail to compile it...
- Threading issues: too much threads for the available cores, combining FAUST parallel modules...

Known issues (1)

Limits of the current approach

- Code size, compilers may fail to compile it...
- Threading issues: too much threads for the available cores, combining FAUST parallel modules...

Known issues (1)

Limits of the current approach

- Code size, compilers may fail to compile it...
- Threading issues: too much threads for the available cores, combining FAUST parallel modules...

Known issues (2)

Some idea of solutions

- Re-organising the task code in separated methods helps compilers
- Still need to improve task code sharing...
- Using threading libraries like "libdispatch" (part of OSX Grand Central Dispatch), but not yet adapted for RT fined grained code
- Using inter-process audio frameworks (like JACK...) to share context between audio RT applications (like for instance the adequate RT threads number to be used, depending of graph topology...)
- Dynamic adaptation: continuously measuring CPU use and starting/stopping threads accordingly...

Known issues (2)

Some idea of solutions

- Re-organising the task code in separated methods helps compilers
- Still need to improve task code sharing...
- Using threading libraries like "libdispatch" (part of OSX Grand Central Dispatch), but not yet adapted for RT fined grained code
- Using inter-process audio frameworks (like JACK...) to share context between audio RT applications (like for instance the adequate RT threads number to be used, depending of graph topology...)
- Dynamic adaptation: continuously measuring CPU use and starting/stopping threads accordingly...

Known issues (2)

Some idea of solutions

- Re-organising the task code in separated methods helps compilers
- Still need to improve task code sharing...
- Using threading libraries like "libdispatch" (part of OSX Grand Central Dispatch), but not yet adapted for RT fined grained code
- Using inter-process audio frameworks (like JACK...) to share context between audio RT applications (like for instance the adequate RT threads number to be used, depending of graph topology...)
- Dynamic adaptation: continuously measuring CPU use and starting/stopping threads accordingly...

Known issues (2)

Some idea of solutions

- Re-organising the task code in separated methods helps compilers
- Still need to improve task code sharing...
- Using threading libraries like "libdispatch" (part of OSX Grand Central Dispatch), but not yet adapted for RT fined grained code
- Using inter-process audio frameworks (like JACK...) to share context between audio RT applications (like for instance the adequate RT threads number to be used, depending of graph topology...)
- Dynamic adaptation: continuously measuring CPU use and starting/stopping threads accordingly...

Known issues (2)

Some idea of solutions

- Re-organising the task code in separated methods helps compilers
- Still need to improve task code sharing...
- Using threading libraries like "libdispatch" (part of OSX Grand Central Dispatch), but not yet adapted for RT fined grained code
- Using inter-process audio frameworks (like JACK...) to share context between audio RT applications (like for instance the adequate RT threads number to be used, depending of graph topology...)
- Dynamic adaptation: continuously measuring CPU use and starting/stopping threads accordingly...

Known issues (2)

Some idea of solutions

- Re-organising the task code in separated methods helps compilers
- Still need to improve task code sharing...
- Using threading libraries like "libdispatch" (part of OSX Grand Central Dispatch), but not yet adapted for RT fined grained code
- Using inter-process audio frameworks (like JACK...) to share context between audio RT applications (like for instance the adequate RT threads number to be used, depending of graph topology...)
- Dynamic adaptation: continuously measuring CPU use and starting/stopping threads accordingly...

Known issues (2)

Some idea of solutions

- Re-organising the task code in separated methods helps compilers
- Still need to improve task code sharing...
- Using threading libraries like "libdispatch" (part of OSX Grand Central Dispatch), but not yet adapted for RT fined grained code
- Using inter-process audio frameworks (like JACK...) to share context between audio RT applications (like for instance the adequate RT threads number to be used, depending of graph topology...)
- Dynamic adaptation: continuously measuring CPU use and starting/stopping threads accordingly...

Conclusion

Possible reuse

- Work Stealing Scheduler is quite efficient
- Easy to implement in the FAUST case (knowledge at compilation time...)
- Could be of interest for other graph based audio languages or environments : SuperCollider, PD...etc...

Possible reuse

- Work Stealing Scheduler is quite efficient
- Easy to implement in the FAUST case (knowledge at compilation time...)
- Could be of interest for other graph based audio languages or environments : SuperCollider, PD...etc...

Conclusion

Possible reuse

- Work Stealing Scheduler is quite efficient
- Easy to implement in the FAUST case (knowledge at compilation time...)
- Could be of interest for other graph based audio languages or environments : SuperCollider, PD...etc...

Conclusion

Possible reuse

- Work Stealing Scheduler is quite efficient
- Easy to implement in the FAUST case (knowledge at compilation time...)
- Could be of interest for other graph based audio languages or environments : SuperCollider, PD...etc...

Conclusion

Possible reuse

- Work Stealing Scheduler is quite efficient
- Easy to implement in the FAUST case (knowledge at compilation time...)
- Could be of interest for other graph based audio languages or environments : SuperCollider, PD...etc...

Outline

1. Work Stealing Scheduler
2. Demo