Building a Scalable Evaluation Engine for Presto

Florian Zitzelsberger
Pixar Animation Studios
I have opening credits. I heard its the hipster thing to do:

- **Florian, John, Matt** and **Ian** make up the Execution Team. They are the real heroes responsible for all this awesome work... and yes, there are two Florians on this team.

- Thanks you to **George, Alicia, Adam, Guido** and **Steve** for their mentorship and support
Here is what I will be talking about:
- (Click) First, I will give you a quick introduction to Presto
- (Click) Then, I will explain the architecture of Presto's execution system. The system was designed from the ground up with multithreading in mind, so I am going to highlight what makes this architecture amenable to parallelization.
- (Click) In the later half of the presentation, I am going to dive into some concrete approaches we use to evaluate our data flow networks on multiple threads
Here is what I will be talking about:
- First, I will give you a **quick introduction** to Presto
- Then, I will explain the **architecture** of Presto’s execution system. The system was designed from the ground up with **multithreading in mind**, so I am going to highlight what makes this architecture **amenable to parallelization**.
- In the later half of the presentation, I am going to dive into some **concrete approaches** we use to evaluate our data flow networks on multiple threads.
Here is what I will be talking about:

- (Click) First, I will give you a **quick introduction** to Presto
- (Click) Then, I will explain the **architecture** of Presto’s execution system. The system was designed from the ground up with **multithreading in mind**, so I am going to highlight what makes this architecture **amenable to parallelization**.
- (Click) In the later half of the presentation, I am going to dive into some **concrete approaches** we use to evaluate our data flow networks on multiple threads
Here is what I will be talking about:
- (Click) First, I will give you a quick introduction to Presto
- (Click) Then, I will explain the architecture of Presto’s execution system. The system was designed from the ground up with multithreading in mind, so I am going to highlight what makes this architecture amenable to parallelization.
- (Click) In the later half of the presentation, I am going to dive into some concrete approaches we use to evaluate our data flow networks on multiple threads
Presto Video

See vimeo.com/90687696 for another example.

- This is a recording of me interacting with Presto, so you can see what it looks like.
- Presto is a digital content creation (or DCC) application. It is Pixar’s proprietary animation tool.
- Our character rigs are built in Presto, and lots of layout and crowds as well as some simulation workflows, are also done in Presto.
- We want workflows in Presto to be interactive and allow for quick iteration in order to extend artistic reach. Subsequently, performance is always big concern.
Now that you have an idea what Presto is, let's talk about the system architecture.
- At the core of Presto is a powerful execution engine (click).
- This is a general purpose computation engine (click), which tracks dependencies between computations (click), using a data flow graph.
- (Click) The execution engine also sparsely invalidates computations as inputs into the network change. Values that are not affected by a changed input value will remain cached.
- (Click) We primarily use this engine for evaluation of character rigs, but lots of imaging system computations are also being implemented through this system.
Execution System

- General purpose computation engine

- At the core of presto is a powerful execution engine (click)
- This is a general purpose computation engine (click), which tracks dependencies between computations (click), using a data flow graph.
- (Click) The execution engine also sparsely invalidates computations as inputs into the network change. Values that are not affected by a changed input value will remain cached.
- (Click) We primarily use this engine for evaluation of character rigs, but lots of imaging system computations are also being implemented through this system.
Execution System

- General purpose computation engine
- Tracks dependencies between computations

- At the core of presto is a powerful execution engine (click)
- This is a general purpose computation engine (click), which tracks dependencies between computations (click), using a data flow graph.
- (Click) The execution engine also sparsely invalidates computations as inputs into the network change. Values that are not affected by a changed input value will remain cached.
- (Click) We primarily use this engine for evaluation of character rigs, but lots of imaging system computations are also being implemented through this system.
Execution System

- General purpose computation engine
- Tracks dependencies between computations
- Underpinned by a data flow graph

- At the core of presto is a powerful execution engine (click)
- This is a general purpose computation engine (click), which tracks dependencies between computations (click), using a data flow graph.
- (Click) The execution engine also sparsely invalidates computations as inputs into the network change. Values that are not affected by a changed input value will remain cached.
- (Click) We primarily use this engine for evaluation of character rigs, but lots of imaging system computations are also being implemented through this system.
At the core of Presto is a powerful execution engine, which is a general purpose computation engine that tracks dependencies between computations using a data flow graph. The execution engine also sparsely invalidates computations as inputs into the network change. Values that are not affected by a changed input value will remain cached.

We primarily use this engine for evaluation of character rigs, but lots of imaging system computations are also being implemented through this system.
At the core of Presto is a powerful execution engine, which is a general purpose computation engine that tracks dependencies between computations using a data flow graph. The execution engine also sparsely invalidates computations as inputs into the network change. Values that are not affected by a changed input value will remain cached. We primarily use this engine for evaluation of character rigs, but lots of imaging system computations are also being implemented through this system.
- There are 3 distinct phases of execution, which I will explain in greater detail:
  - (Click) First, **compilation** produces the **data flow network** from higher level concepts
  - (Click) Then, **scheduling** builds an **acceleration structure** for evaluation
  - (Click) Lastly, **evaluation** produces the **computed values**
- These phases are listed in order of **increasing frequency**, and order of **decreasing cost**
  - **Compilation is expensive**, but it happens relatively **infrequently**
  - **Evaluation** on the other hand is **relatively quick**, but it happens **frequently**
- In the context of multithreading, I will be talking mostly about evaluation. We parallelize work in many different subsystems in Presto, but **evaluation performance is especially critical for interactivity**.
There are 3 distinct phases of execution, which I will explain in greater detail:
- (Click) First, **compilation** produces the **data flow network** from higher level concepts
- (Click) Then, **scheduling** builds an **acceleration structure** for evaluation
- (Click) Lastly, **evaluation** produces the **computed values**

These phases are listed in order of **increasing frequency**, and order of **decreasing cost**
- **Compilation is expensive**, but it happens relatively **infrequently**
- **Evaluation** on the other hand is **relatively quick**, but it happens **frequently**

In the context of multithreading, I will be talking mostly about evaluation. We parallelize work in many different subsystems in Presto, but **evaluation performance is especially critical for interactivity**.
There are 3 distinct phases of execution, which I will explain in greater detail:

- (Click) First, compilation produces the data flow network from higher level concepts.
- (Click) Then, scheduling builds an acceleration structure for evaluation.
- (Click) Lastly, evaluation produces the computed values.

These phases are listed in order of increasing frequency, and order of decreasing cost:
- Compilation is expensive, but it happens relatively infrequently.
- Evaluation on the other hand is relatively quick, but it happens frequently.

In the context of multithreading, I will be talking mostly about evaluation. We parallelize work in many different subsystems in Presto, but evaluation performance is especially critical for interactivity.
There are 3 distinct phases of execution, which I will explain in greater detail:

- First, compilation produces the data flow network from higher level concepts
- Then, scheduling builds an acceleration structure for evaluation
- Lastly, evaluation produces the computed values

These phases are listed in order of increasing frequency, and order of decreasing cost:

- Compilation is expensive, but it happens relatively infrequently
- Evaluation on the other hand is relatively quick, but it happens frequently

In the context of multithreading, I will be talking mostly about evaluation. We parallelize work in many different subsystems in Presto, but evaluation performance is especially critical for interactivity.
During the first phase, a **compiler** digests high level concepts into a **data flow network**

- The high level concepts are essentially a **visual programming language**: 
  - **Schemas** provide templates for the primitives
  - **Scene Description** is where primitives are instantiated and values are assigned (example on next slide)
- The **product** of compilation is a **data flow network**
- A typical network for a character rig has between **70,000 - 100,000 nodes**
- A **challenge** with the compilation architecture is that it must **incrementally update** the network on topological scene changes
- It’s **relatively expensive** to compile a network. This is in the **order of seconds**, but scales with **scene complexity**. Initial compilation happens **infrequently**, for example during shot **load** time.
- Here is an example of **scene description** as displayed in Presto’s **graph editor**
- We have **instantiated** multiple **prims**
- And we can **wire up attributes**, and **assign values** to them
- And express **generic relationships** between different primitives
- The **second phase** of execution is **scheduling**
- The **client makes a request** for computed values
- Using the request, and the data flow network, the **scheduler** will build an **acceleration structure** for evaluation. This structure is optimized for the **data access patterns of evaluation**.
- The network is **expensive to traverse** and we do it often: Frame changes, input value changes (munging), etc.
- This phase is invoked **more frequently** than compilation, and it is significantly **cheaper**: On the order of dozens to hundreds of milliseconds.
The third and last phase of execution is evaluation. During this stage, the executor evaluates the data flow network by iterating over the schedule. For each requested value, it determines which input dependencies are unfulfilled, and it then invokes the corresponding node callbacks. It's important to note that the network does not store any state (topology only): All computed values are stored in the data manager.

- Data manager is owned by an executor
- Computed values are stored optimized for data access patterns
- The data managers are stateful in the sense that during successive rounds of evaluation, values that have previously been computed and not since been invalidated, can be picked up as cached values
static void Add(const VdfContext &ctx)
{
    const double result =
        ctx.GetInputValue<double>(in1Token) +
        ctx.GetInputValue<double>(in2Token);

    ctx.SetOutput(result);
}

- Here is what a node callback might look like. This is what the executor invokes.
- There are several interesting aspects to callbacks in our system:
- The callbacks are registered along with the execution behaviors in the schema.
- They are free standing C++ functions with no access to member variables
- They only take one argument, which is a context object through which input values are read, and output values are written.
- Accessible inputs are registered statically, and ahead of time in the schema.
- The data produced is owned by the data manager, which is responsible for lifetime management.
Thread Safety

- (Click) Because of the way that data access is restricted in node callbacks, we **can easily invoke them in parallel**
- (Click) There is some data access that we cannot easily prevent. For example, access to **globals and singletons** is dangerous.
- (Click) Sometimes access to globals is **not immediately obvious**, for example when calling into **third-party libraries**
- So, as you can see, plugin writers **can still shoot themselves in the foot**, despite the restrictions we put in place
Thread Safety

- Callbacks invoked concurrently

- (Click) Because of the way that data access is restricted in node callbacks, we can easily invoke them in parallel
- (Click) There is some data access that we cannot easily prevent. For example, access to **globals and singletons** is dangerous.
- (Click) Sometimes access to globals is not immediately obvious, for example when calling into **third-party libraries**
- So, as you can see, plugin writers can still shoot themselves in the foot, despite the restrictions we put in place
Thread Safety

- Callbacks invoked concurrently
- Globals and singletons

- (Click) Because of the way that data access is restricted in node callbacks, we **can easily invoke them in parallel**
- (Click) There is some data access that we cannot easily prevent. For example, access to **globals and singletons** is dangerous.
- (Click) Sometimes access to globals is **not immediately obvious**, for example when calling into **third-party libraries**
- So, as you can see, plugin writers **can still shoot themselves in the foot**, despite the restrictions we put in place
Thread Safety

- Callbacks invoked concurrently
- Globals and singletons
- Third-party library access

- (Click) Because of the way that data access is restricted in node callbacks, we can easily invoke them in parallel.
- (Click) There is some data access that we cannot easily prevent. For example, access to globals and singletons is dangerous.
- (Click) Sometimes access to globals is not immediately obvious, for example when calling into third-party libraries.
- So, as you can see, plugin writers can still shoot themselves in the foot, despite the restrictions we put in place.
- In the **context of parallel evaluation**, we are pretty happy with this architecture, because:
  - We have provided some good **layers of abstraction**
  - We have **compartmentalized** the data and responsibilities
  - We have optimized our **data for access patterns**
  - We have **separated constant state from mutable state**
  - And we have **reduced** the opportunity for user error
- Let’s talk about some \textit{concrete approaches} to evaluating data flow networks in parallel
The most obvious way to parallelize is to just multithread the work that is happening inside of a callback. For example, one could simply put a parallel for loop inside this callback to process vectorized data. I've named this "Node-Level Parallelism"
- The most obvious way to parallelize is to just multithread the work that is happening **inside of a callback**
- (Click) For example, one could simply put a **parallel for loop** inside this callback to process vectorized data
- I've named this **“Node-Level Parallelism”**
Another approach is what I call **Input-Value-Level Parallelism**

- Frequently, we want to **compute the same value** or values given a set of different input values
- For example, **time** is just an input to our network, and we often want to compute a particular value over a **frame range**
- (Click) What we can do is **instantiate an executor**, set some **input value to 1** in the data manager, and run the executor on a first task
- (Click) We do the same with a **second executor**, and a second input value on a second parallel task...
- (Click) ... and so on
- Since the executors only **mutate state within** their data managers, and these instances are each owned by a separate task, shared state is not mutated in parallel and no synchronization is required
Another approach is what I call “**Input-Value-Level Parallelism**”

- Frequently, we want to **compute the same value** or values given a set of different input values
- For example, **time** is just an input to our network, and we often want to compute a particular value over a **frame range**

  - (Click) What we can do is **instantiate an executor**, set some **input value to 1** in the data manager, and run the executor on a first task
  - (Click) We do the same with a **second executor**, and a second input value on a second parallel task…
  - (Click) … and so on

- Since the executors only **mutate state within** their data managers, and these instances are each owned by a separate task, shared state is not mutated in parallel and no synchronization is required
- Another approach is what I call “**Input-Value-Level Parallelism**”
- Frequently, we want to **compute the same value** or values given a set of different input values
- For example, *time* is just an input to our network, and we often want to compute a particular value over a **frame range**
- (Click) What we can do is **instantiate an executor**, set some **input value to 1** in the data manager, and run the executor on a first task
- (Click) We do the same with a **second executor**, and a second input value on a second parallel task…
- (Click) … and so on
- Since the executors only **mutate state within** their data managers, and these instances are each owned by a separate task, shared state is not mutated in parallel and no synchronization is required
- Another approach is what I call “Input-Value-Level Parallelism”
- Frequently, we want to compute the same value or values given a set of different input values
- For example, time is just an input to our network, and we often want to compute a particular value over a frame range
- (Click) What we can do is instantiate an executor, set some input value to 1 in the data manager, and run the executor on a first task
- (Click) We do the same with a second executor, and a second input value on a second parallel task…
- (Click) … and so on
- Since the executors only mutate state within their data managers, and these instances are each owned by a separate task, shared state is not mutated in parallel and no synchronization is required
- In Presto, we have been employing this solution to great success with a feature called Background Execution (click twice!)
- In this video, I am modifying the splines on the right, and then playing back a few frames. This is a common workflow for animators.
- (Click) If you watch the green bar on the bottom, you can see some of the frames changing to a darker shade of green as they are being invalidated during the interaction.
- Once the interaction ends, you can see these dark green frames asynchronously being re-computed. They will change back to the lighter shade of green.
- By the time the user hits the play button, and the playhead reaches the invalidated frames, the data will often already be cached, ensuring playback at interactive rates.
Parallelizing across time works great, but what if you want to multithread single frame evaluation. You can take advantage of parallelism where there are two or more parallel branches in the network. To illustrate this, let’s first assign some labels to these nodes. We can run nodes A, B, and C in that order, in one parallel task. The nodes will take various amounts of time to run. Nodes D, E, and F can run in order, in another parallel task. Once both tasks 1 and 2 have completed, Node G can run. This solution works well, if the topology of your networks allows for branch-level parallelism. If your networks don’t have many independent branches, you need to make changes in order to create branches. For example, you could restructure your rigs to deform different appendages on a character independently.
Parallelizing across time works great, but what if you want to multithread single frame evaluation?

- You can take advantage of parallelism where there are two or more parallel branches in the network.

(Click) To illustrate this, let's first assign some labels to these nodes.

(Click) We can run nodes A, B, and C in that order, in one parallel task. The nodes will take various amounts of time to run.

(Click) Nodes D, E, and F can run in order, in another parallel task.

(Click) Once both tasks 1 and 2 have completed, Node G can run.

This solution works well, if the topology of your networks allows for branch-level parallelism.

If your networks don't have many independent branches, you need to make changes in order to create branches.

For example, you could restructure your rigs to deform different appendages on a character independently.
Parallelizing across time works great, but what if you want to multithread single frame evaluation? You can take advantage of parallelism where there are two or more parallel branches in the network. (Click) To illustrate this, let's first assign some labels to these nodes. (Click) We can run nodes A, B, and C in that order, in one parallel task. The nodes will take various amounts of time to run. (Click) Nodes D, E, and F can run in order, in another parallel task. (Click) Once both tasks 1 and 2 have completed, Node G can run. This solution works well, if the topology of your networks allows for branch-level parallelism. If your networks don’t have many independent branches, you need to make changes in order to create branches. For example, you could restructure your rigs to deform different appendages on a character independently.
Parallelizing across time works great, **but what if** you want to multithread **single frame evaluation**?
- You can take advantage of parallelism where there are **two or more parallel branches** in the network.
- (Click) To illustrate this, let’s first assign some **labels** to these nodes.
- (Click) We can run nodes **A, B, and C** in that order, in one parallel task. The nodes will take various amounts of time to run.
- (Click) Nodes **D, E, and F** can run in order, in another parallel task.
- (Click) Once both tasks 1 and 2 have completed, Node **G** can run.
- **This solution works well** if the topology of your networks **allows for branch-level parallelism**.
- If your networks don’t have many independent branches, you need to make changes in order to create branches.
- For example, you could restructure your rigs to deform **different appendages** on a character independently.
Parallelizing across time works great, but what if you want to multithread single frame evaluation. You can take advantage of parallelism where there are two or more parallel branches in the network. To illustrate this, let's first assign some labels to these nodes. We can run nodes A, B, and C in that order, in one parallel task. The nodes will take various amounts of time to run. Nodes D, E, and F can run in order, in another parallel task. Once both tasks 1 and 2 have completed, Node G can run. This solution works well, if the topology of your networks allows for branch-level parallelism. If your networks don't have many independent branches, you need to make changes in order to create branches. For example, you could restructure your rigs to deform different appendages on a character independently.
- (Click) I created an idealized test (representative in the number of nodes and points deformed) to evaluate the performance of our implementation of branch-level multithreading.
- In this case, the network has 5 distinct branches.
- You can see that performance scales well up to 5 cores, but beyond that we don’t see an improvement.
- When we used branch multithreading on our character rigs, the performance improvements where limited to 10-15% because our rigs do not have many parallel branches.
- In fact we have **long, sequential deformer chains**
- We allocate a **contiguous buffer** at the top of the deformer chain and pass it through each deformer node
- **Not every deformer** node will modify **all the data** in this buffer
- In fact, most nodes will only **deform a subset**
- (Click) In order to make this work, we augment our networks with so called **masks**
- Masks are essentially **bit sets** indicating which elements in the buffer are being **read or modified**. They are part of the topology of the network.
- (Click) The deformer labeled **B** has an output mask of size 4, indicating that the point pool consists of 4 elements in this case
- (Click) The **green dots** here indicate elements set in the mask. In this case, deformer **B** is modifying the last 3 elements in the buffer
- In our networks, all data is semantically **treated as a vector** - even scalar values are basically buffers of size 1
- In fact we have **long, sequential deformer chains**
- We allocate a **contiguous buffer** at the top of the deformer chain and pass it through each deformer node
- **Not every deformer** node will modify **all the data** in this buffer
- In fact, most nodes will only **deform a subset**
- (Click) In order to make this work, we augment our networks with so called **masks**
- Masks are essentially **bit sets** indicating which elements in the buffer are being **read or modified**. They are part of the topology of the network.
- (Click) The deformer labeled **B** has an output mask of size 4, indicating that the point pool consists of 4 elements in this case
- (Click) The **green dots** here indicate elements set in the mask. In this case, deformer **B** is modifying the last 3 elements in the buffer
- In our networks, all data is semantically **treated as a vector** - even scalar values are basically buffers of size 1
- In fact we have **long, sequential deformer chains**
- We allocate a **contiguous buffer** at the top of the deformer chain and pass it through each deformer node
- **Not every deformer** node will modify **all the data** in this buffer
- In fact, most nodes will only **deform a subset**
- (Click) In order to make this work, we augment our networks with so called **masks**
- Masks are essentially **bit sets** indicating which elements in the buffer are being **read or modified**. They are part of the topology of the network.
- (Click) The deformer labeled **B** has an output mask of size 4, indicating that the point pool consists of 4 elements in this case
- (Click) The **green dots** here indicate elements set in the mask. In this case, deformer **B** is modifying the last 3 elements in the buffer
- In our networks, all data is semantically **treated as a vector** - even scalar values are basically buffers of size 1
- In fact we have **long, sequential deformer chains**
- We allocate a **contiguous buffer** at the top of the deformer chain and pass it through each deformer node
- **Not every deformer** node will modify **all the data** in this buffer
- In fact, most nodes will only **deform a subset**
- (Click) In order to make this work, we augment our networks with so called **masks**
- Masks are essentially **bit sets** indicating which elements in the buffer are being **read or modified**. They are part of the topology of the network.
- (Click) The deformer labeled **B** has an output mask of size 4, indicating that the point pool consists of 4 elements in this case
- (Click) The **green dots** here indicate elements set in the mask. In this case, deformer **B** is modifying the last 3 elements in the buffer
- In our networks, all data is semantically **treated as a vector** - even scalar values are basically buffers of size 1
Iterators

```cpp
static void Deform(const VdfContext &ctx)
{
    VdfReadWriteIterator<GfVec3d> it(
        ctx, outToken);

    for (; not it.IsAtEnd(); ++it) {
        *it = DeformPoint(*it);
    }
}
```

- In the callback, all vectorized data is accessed via **iterators**
- We don't just provide **raw pointers** to array data, since the data manager is free to choose a **memory layout** optimized for **access patterns**, **temporal cache locality** and **parallel access**
- When we invoke the callback, the **operative subset** of the iterator is limited by the mask
- The vectorized nature of our networks opens up another **opportunity for parallelization**: 
- Using masks, we can **divide vectors into partitions**…
- (Click) … and determine which nodes **overlap with these partitions**
- We determine **new dependencies** within each partition -&gt; **Strips**
- For illustrative purposes, the **partition grain size** here is 1
- In reality, choosing the grain size such that the data in one partition fits comfortably within the **L1 data cache**
- Now that we have these strips, (Click) we can actually invoke the respective node callbacks on **concurrent tasks** (Click) (Click) (Click)
- Note that we still have the **same number of nodes** in the network, we just invoke the node **callback multiple times**, each time restricting the **operative subset** to the partition
- Also note that **progress** along each strip can be made **independently** to each other strip
- This solutions is essentially a **superset of branch multithreading**. It has all the same benefits that branch multithreading provides, but it attempts to **generate additional parallelism** from **exploiting domain knowledge** contained in our vectorized data flow networks.
- The vectorized nature of our networks opens up another opportunity for parallelization:
- Using masks, we can divide vectors into partitions…
- (Click) … and determine which nodes overlap with these partitions
- We determine new dependencies within each partition -> Strips
- For illustrative purposes, the partition grain size here is 1
- In reality, choosing the grain size such that the data in one partition fits comfortably within the L1 data cache
- Now that we have these strips, (Click) we can actually invoke the respective node callbacks on concurrent tasks (Click) (Click) (Click)
- Note that we still have the same number of nodes in the network, we just invoke the node callback multiple times, each time restricting the operative subset to the partition
- Also note that progress along each strip can be made independently to each other strip
- This solutions is essentially a superset of branch multithreading. It has all the same benefits that branch multithreading provides, but it attempts to generate additional parallelism from exploiting domain knowledge contained in our vectorized data flow networks.
The vectorized nature of our networks opens up another opportunity for parallelization:

- Using masks, we can divide vectors into partitions...
- (Click) ... and determine which nodes overlap with these partitions
- We determine new dependencies within each partition -> Strips
- For illustrative purposes, the partition grain size here is 1
- In reality, choosing the grain size such that the data in one partition fits comfortably within the L1 data cache
- Now that we have these strips, (Click) we can actually invoke the respective node callbacks on concurrent tasks (Click) (Click) (Click)
- Note that we still have the same number of nodes in the network, we just invoke the node callback multiple times, each time restricting the operative subset to the partition
- Also note that progress along each strip can be made independently to each other strip
- This solutions is essentially a superset of branch multithreading. It has all the same benefits that branch multithreading provides, but it attempts to generate additional parallelism from exploiting domain knowledge contained in our vectorized data flow networks.
The vectorized nature of our networks opens up another opportunity for parallelization:
- Using masks, we can divide vectors into partitions…
- (Click) … and determine which nodes overlap with these partitions
- We determine new dependencies within each partition -> Strips
- For illustrative purposes, the partition grain size here is 1
- In reality, choosing the grain size such that the data in one partition fits comfortably within the L1 data cache
- Now that we have these strips, (Click) we can actually invoke the respective node callbacks on concurrent tasks (Click) (Click) (Click)
- Note that we still have the same number of nodes in the network, we just invoke the node callback multiple times, each time restricting the operative subset to the partition
- Also note that progress along each strip can be made independently to each other strip
- This solutions is essentially a superset of branch multithreading. It has all the same benefits that branch multithreading provides, but it attempts to generate additional parallelism from exploiting domain knowledge contained in our vectorized data flow networks.
The vectorized nature of our networks opens up another opportunity for parallelization:

- Using masks, we can divide vectors into partitions...
- (Click) ... and determine which nodes overlap with these partitions
- We determine new dependencies within each partition -> Strips
- For illustrative purposes, the partition grain size here is 1
- In reality, choosing the grain size such that the data in one partition fits comfortably within the L1 data cache
- Now that we have these strips, (Click) we can actually invoke the respective node callbacks on concurrent tasks (Click) (Click) (Click)
- Note that we still have the same number of nodes in the network, we just invoke the node callback multiple times, each time restricting the operative subset to the partition
- Also note that progress along each strip can be made independently to each other strip
- This solutions is essentially a superset of branch multithreading. It has all the same benefits that branch multithreading provides, but it attempts to generate additional parallelism from exploiting domain knowledge contained in our vectorized data flow networks.
- The vectorized nature of our networks opens up another opportunity for parallelization:
- Using masks, we can divide vectors into partitions…
- (Click) … and determine which nodes overlap with these partitions
- We determine new dependencies within each partition -> Strips
- For illustrative purposes, the partition grain size here is 1
- In reality, choosing the grain size such that the data in one partition fits comfortably within the L1 data cache
- Now that we have these strips, (Click) we can actually invoke the respective node callbacks on concurrent tasks (Click) (Click) (Click)
- Note that we still have the same number of nodes in the network, we just invoke the node callback multiple times, each time restricting the operative subset to the partition
- Also note that progress along each strip can be made independently to each other strip
- This solutions is essentially a superset of branch multithreading. It has all the same benefits that branch multithreading provides, but it attempts to generate additional parallelism from exploiting domain knowledge contained in our vectorized data flow networks.
I ran the **same idealized test case** I had earlier used to demonstrate the scalability of branch multithreading through the new strip mining algorithm.

It’s still the same network with **only 5 topological branches**, but it scales beyond 5 cores, which is **great news**.

In fact, this chart looks exactly the same after reducing the number of topological **branches to 1**. This is also great, because we expect strip mining to **scale** with the **number of points deformed**, rather than the topology of the network.

With 17 cores the relative **speedup** is about **12x**

(Click) Since we are talking about scalability: **Linear speedup** is a bit of a **myth** when it comes to modern **workstation CPUs**

(Click) You will find that as you increase utilization on your processor, the **clock rate** of each individual core will **decrease** so that the hardware can maintain **thermal and power envelopes**

- The yellow line here is the best we could possibly achieve on our hardware with a workload that is **purely computational**
- I am saying “purely computational” because this does not take into account any **memory traffic**. Remember that even though we have 18 cores available on this CPU, memory controllers and channels are a much more limited resource.
- Now that we’ve made ourselves **feel good** about how this algorithm performs, let’s look at some real world results
- I ran the same idealized test case I had earlier used to demonstrate the scalability of branch multithreading through the new strip mining algorithm.
- It’s still the same network with only 5 topological branches, but it scales beyond 5 cores, which is great news.
- In fact, this chart looks exactly the same after reducing the number of topological branches to 1. This is also great, because we expect strip mining to scale with the number of points deformed, rather than the topology of the network.
- With 17 cores the relative speedup is about 12x
- (Click) Since we are talking about scalability: Linear speedup is a bit of a myth when it comes to modern workstation CPUs
- (Click) You will find that as you increase utilization on your processor, the clock rate of each individual core will decrease so that the hardware can maintain thermal and power envelopes
- The yellow line here is the best we could possibly achieve on our hardware with a workload that is purely computational
- I am saying “purely computational” because this does not take into account any memory traffic. Remember that even though we have 18 cores available on this CPU, memory controllers and channels are a much more limited resource.
- Now that we’ve made ourselves feel good about how this algorithm performs, let’s look at some real world results
I ran the **same idealized test case** I had earlier used to demonstrate the scalability of branch multithreading through the new strip mining algorithm.

- It’s still the same network with **only 5 topological branches**, but it scales beyond 5 cores, which is **great news**.
- In fact, this chart looks exactly the same after **reducing** the number of topological **branches to 1**. This is also great, because we expect strip mining to **scale** with the **number of points deformed**, rather than the topology of the network.
- With 17 cores the relative **speedup** is about **12x**
- (Click) Since we are talking about scalability: **Linear speedup** is a bit of a **myth** when it comes to modern **workstation CPUs**
- (Click) You will find that as you increase utilization on your processor, the **clock rate** of each individual core will **decrease** so that the hardware can maintain **thermal and power envelopes**
- The yellow line here is the best we could possibly achieve on our hardware with a workload that is **purely computational**
- I am saying “purely computational” because this does not take into account any **memory traffic**. Remember that even though we have 18 cores available on this CPU, memory controllers and channels are a much more limited resource.
- Now that we’ve made ourselves **feel good** about how this algorithm performs, let’s look at some real world results
Here is some animation on a primary character from coco

- Without strip mining the cost of evaluating each frame is 49 milliseconds
- With strip mining the cost is now 16 milliseconds
- No changes have been made to the rig or any of the plugin code
- We do get some nice speedups right out of the gate, which our animators are very happy about
- On **average** we get a **3.3x speedup** out of the gate
- (Click) Now, if we plot this against the **12x speedup** we same in the test case from earlier we’re not getting the same dramatic speedup on the **real character rig**
- This is **interesting**, so let’s see if we can find out what’s going on here...
- On **average** we get a **3.3x speedup** out of the gate
- (Click) Now, if we plot this against the **12x** speedup we same in the test case from earlier we’re not getting the same dramatic speedup on the **real character rig**
- This is **interesting**, so let’s see if we can find out what’s going on here…
- On **average** we get a **3.3x speedup** out of the gate
- (Click) Now, if we plot this against the **12x** speedup we same in the test case from earlier we’re not getting the same dramatic speedup on the **real character rig**
- This is **interesting**, so let’s see if we can find out what’s going on here...
So let’s turn on our **instrumentation** and capture a profile of a **single round of evaluation**
- There is a **performance penalty** for having the instrumented profiler on, but we just want to know what all the **threads** are doing
- The **horizontal lines** are the different threads, and the **colorful bars** are where node callbacks are running
- Don’t worry about the colors, they don’t have a special meaning for the purpose of this presentation
- What **sticks out** here is that there are times where **utilization** is dropping. You can clearly see hole patterns in the profile
- (Click) Let me **highlight** those
- (Click) Turns out that there is some underutilization for about **20%** of the timeline
- Okay, that doesn’t sound toooooo bad…
So let’s turn on our **instrumentation** and capture a profile of a **single round of evaluation**

- There is a **performance penalty** for having the instrumented profiler on, but we just want to know what all the **threads** are doing
- The **horizontal lines** are the different threads, and the **colorful bars** are where node callbacks are running
- Don’t worry about the colors, they don’t have an special meaning for the purpose of this presentation
- What **sticks out** here is that there are times where **utilization** is dropping. You can clearly see hole patterns in the profile
- (Click) Let me **highlight** those
- (Click) Turns out that there is some underutilization for about **20%** of the timeline
- Okay, that doesn’t sound toooooo bad…
So let’s turn on our instrumentation and capture a profile of a single round of evaluation. There is a performance penalty for having the instrumented profiler on, but we just want to know what all the threads are doing. The horizontal lines are the different threads, and the colorful bars are where node callbacks are running. Don’t worry about the colors, they don’t have a special meaning for the purpose of this presentation. What sticks out here is that there are times where utilization is dropping. You can clearly see hole patterns in the profile. (Click) Let me highlight those. (Click) Turns out that there is some underutilization for about 20% of the timeline. Okay, that doesn’t sound toooooo bad…
But underutilization on a parallel workload is **bad**, as our old friend **Amdahl** discovered.

(Click) **So using Amdahl's law**, we find out that a 20% underutilization is actually really bad.

And that our results are pretty close to what Amdahl's law would suggest.

But where does this underutilization **come from**?
But underutilization on a parallel workload is **bad**, as our old friend **Amdahl** discovered. (Click) So using Amdahl's law, we find out that a 20% underutilization is actually really bad. And that our results are pretty close to what Amdahl's law would suggest. But where does this underutilization **come from**?
- Let’s look at another data flow example
- In this case, the deformer node C has a second input, which sources it’s value D
- And D depends on the deformation result of A
- (Click) When we apply the strip mining, this type of input dependency will create cross dependencies between strips
- These cross dependencies in turn cause synchronization between the strips
Let’s look at another data flow example. In this case, the deformer node C has a second input, which sources its value D. And D depends on the deformation result of A. (Click) When we apply the strip mining, this type of input dependency will create cross dependencies between strips. These cross dependencies in turn cause synchronization between the strips.
- I spent an **afternoon** looking at a character and quickly found **2 or 3 of these** dependencies which I fixed (rig edit)
- The **hardest part** here was finding **where** these synchronization points occur and **why**
- (Click) A profile captured **after my fixes** shows that some of these holes have gone away, but there are **still some left**
- (Click) I was able to increase utilization **by about 5%**
- I spent an **afternoon** looking at a character and quickly found **2 or 3 of these** dependencies which I fixed (rig edit)
- The **hardest part** here was finding **where** these synchronization points occur and **why**
- (Click) A profile captured **after my fixes** shows that some of these holes have gone away, but there are **still some left**
- (Click) I was able to increase utilization **by about 5%**
- I spent an **afternoon** looking at a character and quickly found **2 or 3 of these** dependencies which I fixed (rig edit)
- The **hardest part** here was finding **where** these synchronization points occur and **why**
- (Click) A profile captured after my fixes shows that some of these holes have gone away, but there are **still some left**
- (Click) I was able to increase utilization by **about 5%**
- So we **started with a 3.3x** speedup right out of the gate
- And invested a few hours to **increase utilization to 85%**
- (Click) which **puts us here**
- (Click) in line with **Amdahl's law** at 85% utilization
- (Click) and the new speedup ends up being **4.3x**
- So we started with a 3.3x speedup right out of the gate
- And invested a few hours to increase utilization to 85%
- (Click) which puts us here
- (Click) in line with Amdahl's law at 85% utilization
- (Click) and the new speedup ends up being 4.3x
- So we started with a **3.3x** speedup right out of the gate
- And invested a few hours to increase utilization to **85%**
- (Click) which puts us here
- (Click) in line with **Amdahl’s law** at 85% utilization
- (Click) and the new speedup ends up being **4.3x**
- So we **started with a 3.3x** speedup right out of the gate
- And invested a few hours to **increase utilization to 85%**
- (Click) which puts us here
- (Click) in line with **Amdahl's law** at 85% utilization
- (Click) and the new speedup ends up being **4.3x**
- In **summary**, we are **pretty happy** with how the strip mining approach has worked out so far:
  - (Click) Our animators and riggers are happy with the speed-up, because it didn’t require any **changes to existing assets or workflows**, and everyone already has the appropriate hardware anyways.
  - (Click) We barely had to touch any **plugin code** to allow for this to work. There were some instances where existing API had been used erroneously, but just worked in the old system, and was now causing issues in the new system.
  - (Click) We know this algorithm **scales with the number of points deformed**, modulo attenuations to the scalability curve from to the synchronization points we had just looked at.
  - (Click) We now have this **new front that we can push on**, to get additional speedups, namely fixing synchronization points in the network.
  - We currently do not have great **tools** to solve this **new type of problem**, so we will have to invest in infrastructure for finding these issues.
In summary, we are pretty happy with how the strip mining approach has worked out so far:

- (Click) Our animators and riggers are happy with the speed-up, because it didn’t require any changes to existing assets or workflows, and everyone already has the appropriate hardware anyways.
- (Click) We barely had to touch any plugin code to allow for this to work. There were some instances where existing API had been used erroneously, but just worked in the old system, and was now causing issues in the new system.
- (Click) We know this algorithm scales with the number of points deformed, modulo attenuations to the scalability curve from to the synchronization points we had just looked at.
- (Click) We now have this new front that we can push on, to get additional speedups, namely fixing synchronization points in the network.
- We currently do not have great tools to solve this new type of problem, so we will have to invest in infrastructure for finding these issues.
- In summary, we are pretty happy with how the strip mining approach has worked out so far:
- (Click) Our animators and riggers are happy with the speed-up, because it didn’t require any changes to existing assets or workflows, and everyone already has the appropriate hardware anyways
- (Click) We barely had to touch any plugin code to allow for this to work. There were some instances where existing API had been used erroneously, but just worked in the old system, and was now causing issues in the new system.
- (Click) We know this algorithm scales with the number of points deformed, modulo attenuations to the scalability curve from to the synchronization points we had just looked at
- (Click) We now have this new front that we can push on, to get additional speedups, namely fixing synchronization points in the network.
- We currently do not have great tools to solve this new type of problem, so we will have to invest in infrastructure for finding these issues
- In summary, we are pretty happy with how the strip mining approach has worked out so far:
- (Click) Our animators and riggers are happy with the speed-up, because it didn’t require any changes to existing assets or workflows, and everyone already has the appropriate hardware anyways
- (Click) We barely had to touch any plugin code to allow for this to work. There were some instances where existing API had been used erroneously, but just worked in the old system, and was now causing issues in the new system.
- (Click) We know this algorithm scales with the number of points deformed, modulo attenuations to the scalability curve from to the synchronization points we had just looked at
- (Click) We now have this new front that we can push on, to get additional speedups, namely fixing synchronization points in the network.
- We currently do not have great tools to solve this new type of problem, so we will have to invest in infrastructure for finding these issues
In summary, we are pretty happy with how the strip mining approach has worked out so far:

- (Click) Our animators and riggers are happy with the speed-up, because it didn’t require any changes to existing assets or workflows, and everyone already has the appropriate hardware anyways.
- (Click) We barely had to touch any plugin code to allow for this to work. There were some instances where existing API had been used erroneously, but just worked in the old system, and was now causing issues in the new system.
- (Click) We know this algorithm scales with the number of points deformed, modulo attenuations to the scalability curve from to the synchronization points we had just looked at.
- (Click) We now have this new front that we can push on, to get additional speedups, namely fixing synchronization points in the network.
- We currently do not have great tools to solve this new type of problem, so we will have to invest in infrastructure for finding these issues.
Thanks

@mpflorianz
@multithreadvfx

multithreadingandvfx.com

“Multithreading for Visual Effects”
Watt, Coumans, ElKoura, Henderson, Kraemer, Lait, Reinders