ORACLE

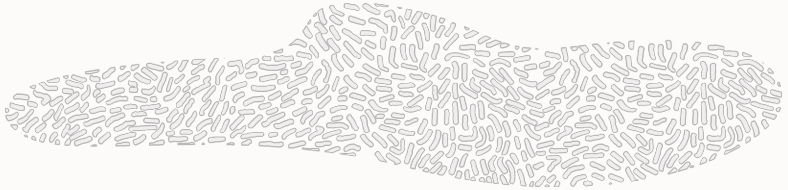
(Heterogeneous Accelerator Toolkit) HAT Update

How Babylon and Panama Enable Java GPGPU Collaboration Opportunities

Gary Frost and Paul Sandoz

8/5/2024

Agenda



- Heterogeneous Accelerator Toolkit (HAT)
  - An NDRange based Java GPGPU compute framework
    - Leveraging :
      - Panama FFM (Foreign Function & Memory API)
      - Babylon Code Reflection
      - Class File API
- Programming model
  - Modelling compute and kernel calls
  - How Babylon helps us maximize performance
  - How Panama FFM lets us offer object like 'views' of off heap data via interfaces
- Summary & Q/A

Heterogenous Accelerator Toolkit (HAT)

HAT is toolkit targeting Heterogeneous devices offering

- An NDRange style parallel programming model (idioms familiar to GPU developers)

  Other programming models could be supported

  Triton, OpenMP/OpenACC/TornadoVM style loop annotations

- A compute programming model

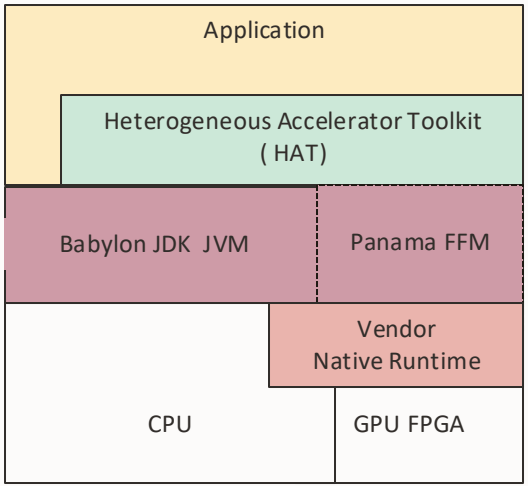
  For describing how kernels are to be dispatched

- Data models and patterns for representing off-heap data as Java interfaces

  Allowing data to be passed between Java code, devices and libraries

- A pluggable Backend abstraction

  Allowing vendors to best showcase their own capabilities



What is GPGPU Compute?

"Performing unnatural acts with shaders and vertex buffers for profit"

GPU (Graphical Processing Units) originally designed and optimized for pixel rendering

- Lots of small cores, high throughput, embarrassingly parallel
  Some communication possible with neighboring pixels (convolutions…)

- Conceptually a thread of execution (shader/kernel) dedicated to shading/coloring each pixel on a display
  or dedicated to positioning each vertex in 2D/3D space

GPGPU (General Purpose GPU) Compute

- The use of GPUs to solve more "general" (but still embarassingly parallel) compute problems

- How?
  Make input data look like vertex buffers, move data to the GPU and use kernels/shaders to modify data and return to the host

$S_{(ingle)}$IMD < $S_{(ingle)}$IMT < $S_{(imultaneous)}$MT

- https://yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html

Java GPGPU Pain Points

(…. pictures of scars available upon request)

Initial offerings provided JNI (Java Native Interface) bindings for existing OpenCL/CDUA frameworks

- Java implementation inherited/exposed the underlying programming model
- Kernel's were just OpenCL/CUDA (C99) style code in Java strings

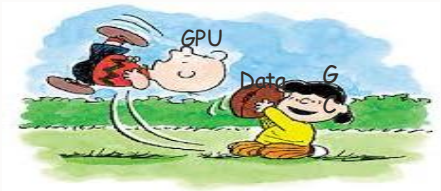
Later (Rootbeer/Aparapi/TornadoVM) allowed kernels to be expressed in Java

- Intent extracted by lifting from bytecode directly or using Graal to lift for us

Kernel's could only access uniform primitive and single dim primitive arrays as args

Each dispatch required copying heap allocated arrays primitive to/from GPU

- JNI allowed us to PIN buffers (to avoid GC moving them)
  - BUT We could only PIN for the duration of a single JNI call AND In reality PIN == STOP GC



Great for single kernels, but most applications need complex kernel graphs and minimal data copies

What do we mean by an NDRange Programming Model and what is a Kernel?

Here we have code to 'square' each value in an int array

... the **for()** loop provides a 1D (0..length-1) range of indices

```
for (int i = 0; i < arr.length; i++) {
    arr[i] = arr[i] * arr[i];
}
```

Lets move the loop body into a method (our **kernel**)

... which accepts index 'i' and the other ***uniform*** values as args

```
void square(int i, int[] arr) {
    arr[i] = arr[i] * arr[i];
}
```

We can of course still dispatch our kernel via a loop

```
for (int i = 0; i < arr.length; i++) {
    square(i, arr);
}
```

But we enable other options for mapping kernel identity

.. here an **IntStream** provides the index 'i' for each kernel

```
IntStream.range(0, arr.length).forEach(
    i -> square(i, arr)
)
```

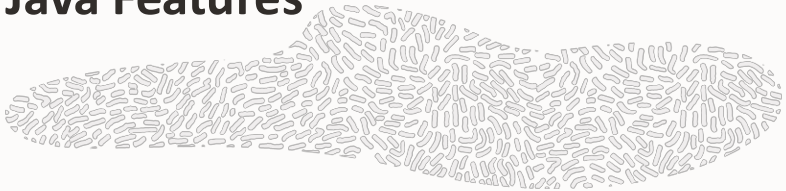
Most interesting from a GPU perspective...

.. we can also use a **parallel IntStream**

.. (**Sumatra**'s programming model)

```
IntStream.range(0, arr.length).parallel().forEach(
    i -> square(i, arr)
)
```


How HAT leverages current and Proposed Java Features



Panama FFM API

Offers a cleaner solution than JNI for linking to native code

Even automating generation of vendor bindings using jextract

With **`MemoryLayouts`** we can express structured off heap data (… align with existing APIs/Standards)

`MemorySegments` can be allocated (either by the JVM or by vendor runtime) and mapped to layouts

With vendor allocation, we can open zero copy routes to GPU

No pinning needed

Babylon/Code Reflection

Greatly simplifies extraction of kernel (and kernel dispatch) intent

We can access code models of Kernels (as well code dispatching the Kernels) without 'lossy' bytecode lifting

HAT Programming Model : Revisit Our Squares example

Here we just copy our previous 'kernel like' **`square(int i, …)`** method into a **`Square`** class.

Kernels can only access Java primitives and *'interface off-heap abstractions*' (more later…)

.. so we replace **`int[]`** with **`S32Arr`**

```
public class Square {
    @CodeReflection
    public static void square(int i, S32Arr s32Arr) {
        s32Arr.array(i, s32Arr.array(i) * s32Arr.array(i));
    }

    @CodeReflection
    public static void kernel(KernelContext kc, S32Arr s32Arr) {
        square(kc.x, s32Arr) ;
    }
  //...
```

We add a **`kernel(KernelContext kc, …)`**

.. HAT's notion of a kernel entrypoint

.. which delegates to **`square()`**

.. so **`kernel()`**'s **`kc.x`** field provides **`square()`**'s index

Kernels and any method 'reachable' from a kernel require Babylon's **`@CodeReflection`** annotation

HAT Programming Model

`KernelContext` allows us to abstracts away vendor specific mechanisms for querying kernel 'identity'

`kc.x` is our kernels 'identity'
in the 1D range **`0 .. kc.maxX`**

KernelContext	OpenCL	CUDA
kc.x	get_global_id(0)	blockIdx.x * blockDim.x + threadIdx.x
kc.maxX	get_global_size(0)	gridDim.x * blockDim.x
kc.group.x	get_group_id(0)	blockIdx.x
kc.group.maxX	get_group_size(0)	blockDim.x

Initially we support 1D range via **`KernelContext`**

We may expose 2D and 3D ranges via **`KernelContext2D`** and **`KernelContext3D`**

Offering additional 'identity' fields (**`kc2d.y`**, **`kc3d.z`**, **`kc2d.maxY`** etc)

HAT Programming Model

The **`KernelContext`** also provides a vendor neutral namespace for accessing GPU runtime APIs

Opportunities here for:

- atomics
- lanewise functions
- global to local memory copies
- prefix scans/sums
- id mapping (for reductions)

NDRange	OpenCL	CUDA
kc.local.barrier()	barrier(CLK_LOCAL_MEM_FENCE)	__syncthreads()
kc.group.alloc(T,size)	__local__ f32_t newArr[size]	__shared__ f32_t newArr[size]

`kc.math.invSqrt()`
`kc.lane.shuffleDown()`
`kc.atomic.inc(arr, idx)`		

We need vendor feedback to determine an appropriate ‘set’ of capabilities

HAT Programming Model : Back to the Squares example

We can of course use a single kernel method and move the logic inside the kernel

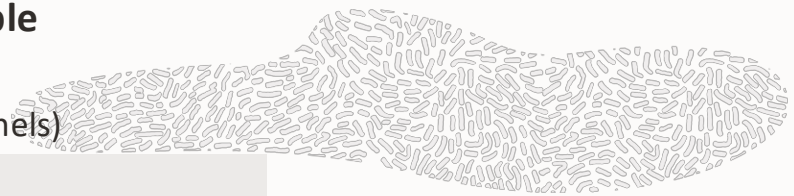
```
@CodeReflection
public static void square(int i, S32Arr s32Arr) {
    s32Arr.array(i, s32Arr.array(i) * s32Arr.array(i));
}

@CodeReflection
public static void kernel(KernelContext kc, S32Arr s32Arr) {
    square(kc.x, s32Arr);
}
```

```
@CodeReflection
public static void kernel(KernelContext kc, S32Arr s32Arr) {
    s32Arr.array(kc.x, s32Arr.array(kc.x) * s32Arr.array(kc.x));
}
```

However, maintaining this separation (during development) is useful for testing kernel logic

```
IntStream.range(0, s32Arr.length()).forEach(
    i -> square(i, s32Arr)
)
```


HAT Programming Model : Revisit Our Squares example



We need to add a compute method to dispatch a kernel (or kernels)

```
public class Square {
    @CodeReflection
    public static void kernel(KernelContext kc, S32Arr s32Arr) {
        s32Arr.array(kc.x, s32Arr.array(kc.x) * s32Arr.array(kc.x));
    }

    @CodeReflection
    public static void compute(ComputeContext cc, S32Arr s32Arr){
        cc.dispatchKernel(s32Arr.length(), kc -> kernel(kc, s32Arr));
    }
}
```

To execute a compute method we need an 'accelerator' bound to a vendor provided Backend

```
var acc = new Accelerator(MethodHandles.lookup(), backend -> backend instanceof OpenCLBackend);
```

Then we execute a compute entrypoint via the Accelerator

```
acc.compute(cc -> Square.compute(cc, s32Arr));
```


HAT Programming Model

Note we are really passing a Babylon 'Quotable Lambda' to the accelerator here

```
acc.compute(cc -> Square.compute(cc, s32Arr));
```

```
public class Square {
    @CodeReflection
    public static void kernel(KernelContext kc, S32Arr s32Arr) {
        s32Arr.array(kc.x, s32Arr.array(kc.x) * s32Arr.array(kc.x));
    }

    @CodeReflection
    public static void compute(ComputeContext cc, S32Arr s32Arr) {
        cc.dispatchKernel(s32Arr.length(), kc -> kernel(kc, s32Arr));
    }
}
```

We may allow everything to be wrapped in a quotable lambda

```
acc.compute(cc ->
    cc.dispatchKernel(s32Arr.length(),
        kc -> s32Arr.array(kc.x, s32Arr.array(kc.x) * s32Arr.array(kc.x))
    )
);
```

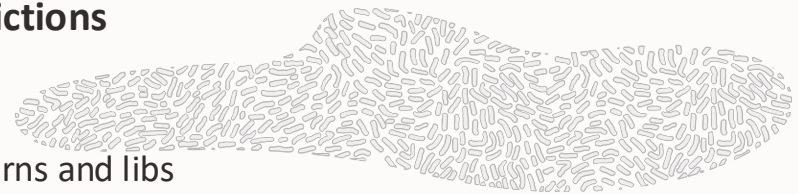
We may also allow avoiding the 'compute method' for trivial single kernel dispatches

```
acc.dispatchKernel(s32Arr.length(),
    kc -> s32Arr.array(kc.x, s32Arr.array(kc.x) * s32Arr.array(kc.x))
);
```

Accelerator + Backend has complete control once the Quotable Lambda is passed !!!!!!

- Consider the Java source of compute/kernel methods as an *expressions of intent*
  - Breakpoints set in compute methods and kernels will be hugely disappointing ;)

HAT Programming Model : Kernel and Compute restrictions



Kernels are restricted to a subset of Java constructs, patterns and libs

- No heap allocations, no exceptions, no object identity, no reflection, no arrays, no access to traditional Java objects

Kernels can :

- Access their args (**`KernelContext`**, + Java primitives and 'interface mapped Panama Segments` – more later)
- Perform basic arithmetic operations
- Call static public methods from the same class (subject to similar constraints as kernels)
- Call some Math.xxx() methods which can be mapped to vendor runtime
  - We might be forced to exert control via the **`KernelContext`** to avoid incompatibilities
    - kc.math.sqrt()
- Contain simple control flow
  - for(;;){}, while(){}, do{}while(), if(){}[else{}],switch(),()?:
  - But any form of branching effects performance

Compute methods have fewer restrictions

- … although some patterns may be adversely affect performance and/or where/how the compute method is executed

HAT Programming Model – Under The 'Bonnet'

```
public class Square {
    @CodeReflection
    public static void kernel(KernelContext kc, S32Arr s32Arr) {
        s32Arr.array(kc.x, s32Arr.array(kc.x) * s32Arr.array(kc.x));
    }

    @CodeReflection
    public static void compute(ComputeContext cc, S32Arr s32Arr) {
        cc.dispatchKernel(s32Arr.length(), kc -> kernel(kc, s32Arr));
    }
}
```

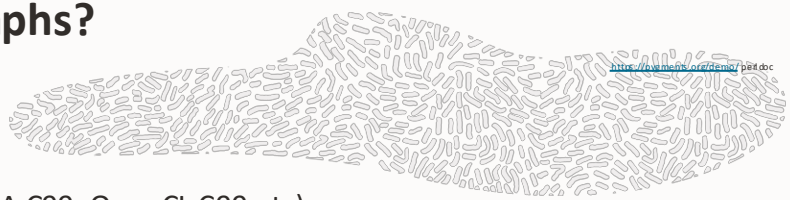
```
acc.compute(cc -> Square.compute(cc, s32Arr) );
```

When **`acc.compute(`*`Quotable`*`)`** is called we construct a **`ComputeContext`** from the code model of the quotable lambda

This **`ComputeContext`** contains a compute call-graph rooted at the lambda's code model and containing, transitively, all code models of reflected application methods that are invoked

For each kernel dispatched from this compute call-graph the **`ComputeContext`** maps a kernel call-graph to the appropriate **`dispatchKernel`** method in the graph

What does HAT/Babylon do with these call graphs?



HAT passes the kernel call-graph directly to the backend

- Which presumably generates vendor specific code (SPIRV, PTX, CUDA C99, OpenCL C99 etc)

To achieve optimal performance a backend needs information derived from:-

- The kernel code – Does a kernel mutate data passed to it? Or just read it?
- The compute code – Does any Java code (between kernel dispatches) access/mutate data ?

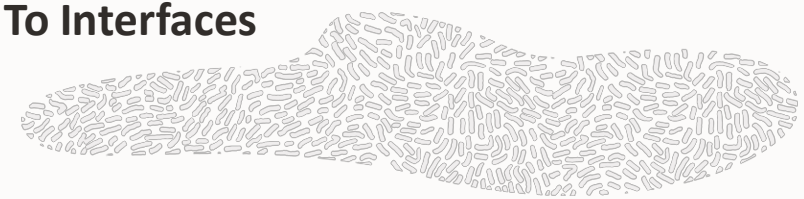
Exposing the compute call-graph, distinguishes HAT from Aparapi, RootBeer, Sumatra and TornadoVM

- TornadoVM users explicitly provide a 'task graph' ('host->gpu', 'dispatch A', 'gpu->host', 'dispatch B')
  - No loops, No conditionals

HAT presumes that the backend can deduce buffer access patterns

- Statically from the compute and kernel models
  - Or
- Dynamically by injecting code into the compute call graph before executing

Mapping Panama FFM API MemorySegments To Interfaces



GPU compute requires us to exchange data with GPU device(s)

We can't pass Java references (Objects or Records) to the GPU
- Even if we could locate the object/record on the heap and pin it (to stop GC from moving it)
- …the backend would have to be able to determine field order, and offsets… field size, padding, etc.

Arrays of Java references are even worse.
- We might find the enclosing array
  - … but it's just a list of references to Objects/Records which are on the heap

Panama FFM is key here

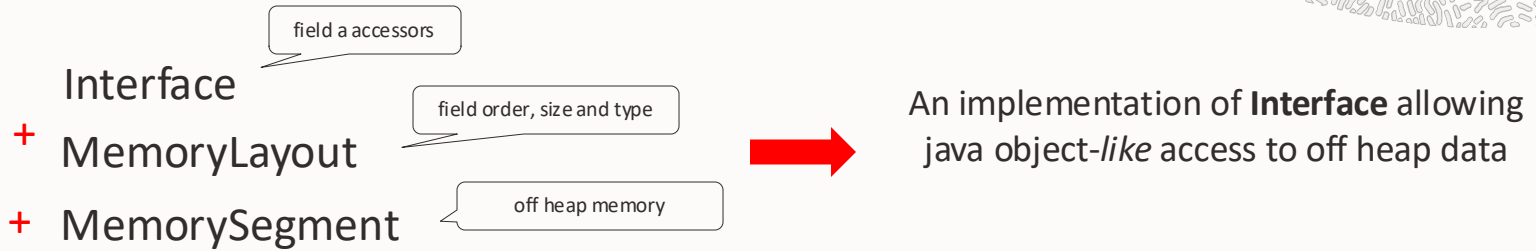
`MemoryLayout's` can describe complex in-memory data structures (think C99 style nested structs/unions)

`MemorySegment's` can be allocated off heap (contiguous memory) and mapped to **`MemoryLayouts`**

Although dealing with 'raw' **`MemorySegments`**, **`MemoryLayouts`**, **`VarHandles`** and **`MethodHandles`** is not trivial ☺

Mapping Panama FFM API MemorySegments To Interfaces

At JVMLS2023 we demonstrated a **`java.lang.reflect.Proxy`** based tool which took:



HAT now has a much more robust (and performant) implementation of this pattern (*)

Using the Classfile API to spin up a classes rather than using **`java.lang.reflect.Proxy`**

We also introduced a use fluent style builder API to simplify the creation of MemoryLayouts.

(*) Many thanks to Maurizio Cimadamore and Per-Ake Minborg for this implementation. It is awesome!

Mapping Panama FFM API MemorySegments To Interfaces

To create an off heap Point like 'entity' XY

We define an interface with accessors for x() and y()

```
public interface XY extends hat.buffer.Buffer {
    int x();
    void x(int x);
    int y();
    void y(int y);
}
```

Then a Schema which describes the order
(amongst other things as we will see later)

```
Schema<XY> schema = Schema.of(XY.class, s -> s
    .field("x")
    .field("y")
);
```

We can then allocate using an Accelerator(*)

```
XY xy = schema.allocate(accelerator);
```

Then use as if it were a Java class or record.

```
xy.x(1);
assert xy.x() == 1 ;
```

(*) The accelerator defers to it's backend, which defers to the vendor provided runtime.

Ideally vendor allocated memory can be in regions shared between the host and the GPU so we can avoid copies

Mapping Panama FFM API MemorySegments To Interfaces

A pattern emerged – Where we include the Schema and allocation function in the interface

```
public interface XY extends hat.buffer.Buffer {
    int x();
    void x(int x);
    int y();
    void y(int y);

   Schema<XY> schema = Schema.of(XY.class, s -> s
      .fields("x","y")
   );

   XY create(Accelerator accelerator) {
        return schema.allocate(accelerator);
   }
}
```

For CUDA and OpenCL Backends

The schema is also used for generating C99 style artifacts

```
typedef struct XY_s {
    int x;
    int y;
} XY_t;
```


Mapping Panama FFM API MemorySegments To Interfaces

Here is the **S32Arr** mapping we used in the earlier Square example

Note that the schema defines the order **and** describes the relationship between fields

Here we bind the **length** field (which does not have a 'setter') to an **array** field

At allocation, we provide a *length* value and the memory backing the **length** field is initialized appropriately

```
public interface S32Arr extends Buffer {
    int length();
    int array(long idx);
    void array(long idx, int i);

    Schema<S32Ar> schema = Schema.of(S32Arr.class, s -> s
        .arrayLen("length").of("array")
    );

    S32Arr allocate(Accelerator acc, int length) {
        return schema.allocate(acc, length);
    }
}
```

CUDA/OpenCL form

flexible array member

incomplete array type

```
typedef struct S32Arr_s {
    int length;
    int array[0]; //or[]or[1]?
} S32Arr_t;
```


Mapping Panama FFM API MemorySegments To Interfaces

We can also nest data .. For example here we have an array of 'structs'

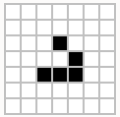
```
public interface XYWArr extends Buffer {
    public interface XYW extends Buffer.struct { // or Buffer.union
        int x();
        void x(int x);
        int y();
        void y(int y);
        float w();
        void w(float weight);
    }

    int length();
    XYW xyw(long idx);
    Schema<XYWArr> schema = Schema.of(XYWArr.class, s -> s
        .arrayLen("length").array("xyw",  arr -> arr
            .fields("x","y","weight")
        )
    );
    XYWArr create(Accelerator acc, int length){
        return schema.allocate(acc, length);
    }
 }
```

```
typedef struct XYW_s {
    int x;
    int y;
    float w;
} XYW_t;

typedef struct XYWArray_s {
    int length;
    XYW_t xyw[0];
} XYWArray_t;
```

```
var arr = XYArr.create(acc, 100);

var weightedPoint = arr.xyw(20);
weightedPoint.x(1)

assert arr.xyw(20).x() == 1;
```


John Conway's Game Of Life : Kernel

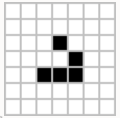
nw	n	ne
w		e
sw	s	se



```
for each generation
  for each cell in grid of cells
    if cell is 0 (dead) with 3 neighbours
      it comes alive
    else if cell is 1 (alive) with 2 neighbours
      it remains alive
    else
      it is dead
```

```
@CodeReflection
public static int neighbour(CellGrid grid,
   int from, int w, int x, int y) {
  return grid.cell(((long)y*w)+x+from)&1;
}
@CodeReflection
public static void life(KernelContext kc,
    Control ctrl, CellGrid grid) {
  int w = grid.width();
  int h = grid.height();
  int x = kc.x % w;
  int y = kc.x / w;
  int from = ctrl.from();
  byte cell = grid.array(kc.x + from);
  if(x>0 && x<(w-1) && y>0 && y<(h-1)) {
     int cnt=neighbour(grid,from,w,x-1,y-1)   //nw
             +neighbour(grid,from,w,x-1,y+0)  // n
             +neighbour(grid,from,w,x-1,y+1)  //ne
             +neighbour(grid,from,w,x+0,y-1)  // e
             +neighbour(grid,from,w,x+0,y+1)  // w
             +neighbour(grid,from,w,x+1,y+0)  //sw
             +neighbour(grid,from,w,x+1,y-1)  // s
             +neighbour(grid,from,w,x+1,y+1); //se
      cell=(cnt==3||(cnt==2 && cell==ALIVE))?ALIVE:DEAD;
   }
   grid.array(kc.x + ctrl.to(), cell);
}
```


John Conway's Game Of Life



https://pygments.org/demo/ perldoc

```
public interface CellGrid extends Buffer {
    //…
    Schema<CellGrid> schema=Schema.of(
         CellGrid.class, cg -> cg
          .arrayLen("width","height").mul(2)
          .array("cell")
    );
    //…
}
```

Rather than having separate **from** and **to** grid buffers
   … and swapping buffers on alternate generations

We have a small `Control` buffer with `from` and `to` offsets
   … which we DO swap on alternate generations

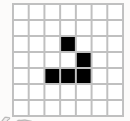
We double size our grid buffer (`2*img.width*img.height`)
   … width and height fields are bound to the cell array and we specify a 'multiplier' of 2 via the schema

Swapping separate buffers each generation would (likely) force 4 grid copies for each generation (2 in and 2 out)

This approach allows a backend (with access to kernel and compute code models) to leave the buffer on the GPU
   … across multiple generations
   … only copying out when Java wants the data

```
public interface Control extends Buffer {
    int from();
    void from(int from);
    int to();
    void to(int to);
}
```


John Conway's Game Of Life



A naïve (Aparapi/Rootbeer like) implementation

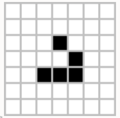
Without benefit of 'compute graph' analysis

cc.dispatchKernel()
- … Copy grid to device
- … Copy control to device
- … Execute kernel
- … Copy control back
- … Copy grid back

```
@CodeReflection
public static void compute(ComputeContext cc,
        Viewer viewer, Control ctrl, CellGrid grid) {
    while (viewer.isVisible()) {
        cc.dispatchKernel(
            grid.width()*grid.height(), // range
            kc -> life(kc, ctrl, grid)
        );

        int from = ctrl.from(); // swap
        ctrl.from(ctrl.to());   // ctrl.from <-> ctrl.to
        ctrl.to(from);          // …

        if (viewer.isReadyForUpdate()) {
            viewer.update(grid);
        }
    }
}
```

```
acc.compute(
    cc -> Life.compute(cc, viewer, ctrl, grid)
);
```


John Conway's Game Of Life : Using Babylon to Track Buffers



https://pygments.org/demo/ perldoc

One possible 'backend' approach :-

Transform the compute graph model

Wrapping iface setters and getters

And

Wrapping calls with buffer args

.. with calls to notify the backend

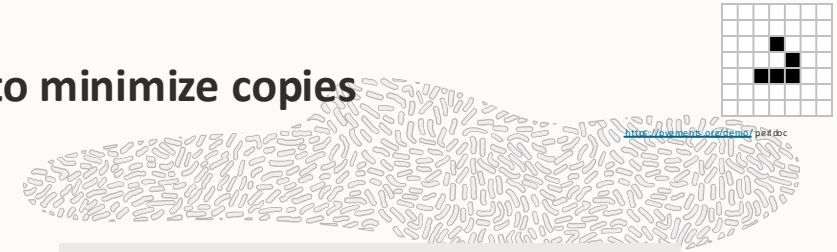
Then convert model to bytecode

... and just execute via the JVM

```
prevFOW.transformInvokes((bldr, invokeOW) -> {
    CopyContext bldrCntxt = bldr.context();
    Value cc = bldrCntxt.getValue(prevFOW.parameter(0));
    if (invokeOW.isIfaceMutator()) {          // Setter
       Value iface = bldrCntxt.getValue(invokeOW.operandNAsValue(0));
       bldr.op(CoreOp.invoke(MUT.pre, cc, iface));                    // cc->preMut(iface);
       bldr.op(invokeOW.op());                                        // original iface.v(newV);
       bldr.op(CoreOp.invoke(MUT.post, cc, iface));                   // cc->postMut(iface)
    } else if (invokeOW.isIfaceAccessor()) { // Getter
       Value iface = bldrCntxt.getValue(invokeOW.operandNAsValue(0));
       bldr.op(CoreOp.invoke(ACC.pre, cc, iface));                    // cc->preAcc(iface);
       bldr.op(invokeOW.op());                                        // original iface.v();
       bldr.op(CoreOp.invoke(ACCE.post, cc, iface));                  // cc->postAcc(iface);
    } else {
       invokeOW.ifaceOperands().stream().forEach(ifaceArg ->
          bldr.op(CoreOp.invoke(
            ESC.pre, cc, bldrCntxt.getValue(ifaceArg))));             // cc.preEsc(ifaceArg)

       bldr.op(invokeOW.op());

       invokeOW.ifaceOperands().stream().forEach(ifaceArg ->
          bldr.op(CoreOp.invoke(
             ESC.post, cc, bldrCntxt.getValue(ifaceArg))));           // cc.postEsc(ifaceArg)
    }
    return bldr;
});
```


8/5/2024

John Conway's Game Of Life : Using Babylon to minimize copies

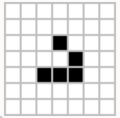


The Java source equivalent after executing our transformer

```
while (viewer.isVisible()) {
    cc.dispatchKernel(
        grid.width()*grid.height(),
        kc -> life(kc, ctrl, grid)
    );

    int from = ctrl.from();
    ctrl.from(ctrl.to());
    ctrl.to(from);

    if (viewer.isReadyForUpdate()) {
        viewer.update(grid);
    }
}
```

```
while (viewer.isVisible()) {
    cc.dispatchKernel(
        grid.width()*grid.height(),
        kc -> life(kc, ctrl, grid)
    );
    cc.preAccess(ctrl);
    int from = ctrl.from();
    cc.postAccess(ctrl);
    cc.preMutate(ctrl);
    cc.preAccess(ctrl);
    ctrl.from(ctrl.to());
    cc.postAccess(ctrl);
    cc.postMutate(ctrl);
    cc.preAccess(ctrl);
    ctrl.to(from);
    cc.postAccess(ctrl);
    if (viewer.isReadyForUpdate()) {
        cc.preEscape(grid);
        viewer.update(grid);
        cc.postEscape(grid);
    }
}
```


John Conway's Game Of Life : Notes from Compute method POV



https://pygments.org/demo/ perldoc

So with our tracing a backend
benefitting from 'compute graph' analysis

On first loop's cc.dispatchKernel()

- … Copy grid to device
- … Copy control to device
- … Execute kernel

On subsequent loop's cc.dispatchKernel()

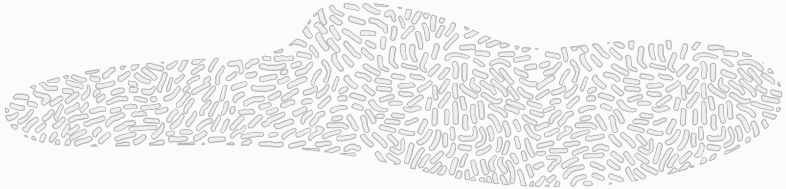
if (prev loop called viewer.update(grid)
- … Copy grid to device (it was updated!)
- … Copy control to device
- … Execute kernel

else
- … Copy control to device
- … Execute kernel

```
@CodeReflection
public static void compute(ComputeContext cc,
        Viewer viewer, Control ctrl, CellGrid grid) {
    while (viewer.isVisible()) {
        cc.dispatchKernel(
            grid.width()*grid.height(), // range
            kc -> life(kc, ctrl, grid)
        );

        int from = ctrl.from(); // swap
        ctrl.from(ctrl.to());   // ctrl.from <-> ctrl.to
        ctrl.to(from);          // …

        if (viewer.isReadyForUpdate()) {
            viewer.update(grid);
        }
    }
}
```


8/5/2024

Game Of Life : Demo



Healing Brush Demo

Based on Google's Renderscript demo.

Renderscript is/was GPU compute framework for Android (pre Vulkan)

https://github.com/yongjhih/HealingBrush/blob/master/src/rs/example/android/com/healingbrush

Summary

- We provided a brief update on the HAT Project
  - Discussed the programming model
  - Highlighted challenges and Opportunities
- We showed how HAT leverages :-
  - Babylon Code Reflection
  - Class File API
  - Panama FFM
- Described how these technologies help us realize GPGPU performance from Java

What's Next ?

- Feedback & Discussion
  Ongoing hardware vendors and existing Java GPU frameworks
  Come to the workshop this afternoon

- Collaboration
  https://github.com/openjdk/babylon
  Backend examples, demos, examples etc.

- Implementation
  Still some experimentation to do and API's to clean up