Domain Specific Languages for Financial Payoffs

Matthew Leslie
Bank of America Merrill Lynch

Outline

- Introduction
    - What, How, and Why do we use DSLs in Finance?
- Implementation
    - Interpreting, Compiling
- Performance
    - Parallelism
    - How Fast Can It Be?
    - What Optimisations Are Important?

Financial Payoffs

- Pay a certain amount on a certain date.
- Pay if a stock price is above a certain level on a certain date.
- Pay if the average performance of a basket of stocks remains above a certain level over a certain period.

How do we describe them

- Imperative
  - If(Spot[Expiry]>Barrier,1,0)
  - HitDate=FindFirst(Spot[start..end]>Barrier)
- Declarative
  - When (at Expiry) (Spot-Strike or 0)

Imperative Descriptions

Basic Language Constructs

Get a Simulated Asset Price

Record A Payment

Add,Sub,Div,Mul,Exp,Log

Min,Max,Avg

Conditionals

Loops

Assignments

- Arrays
  - Of stock prices and dates
- Input Parameters
  - Expiry Date
  - Strike
  - Basket Constituents

Simple Payoff Examples

- Asian Call Option
  - Max(Avg(Spot(1:n) / Spot[0] - strike ,0)
- Cliquet
  - Sum(Max(Spot(1:n) / Spot(0:n-1) – strike ,0))
- Capped Floored Cliquet
  - Max(Sum(Max(Min(Returns(Spots(0:n)), loc_cap), loc_floor)), glob_floor)

How do we use the description?

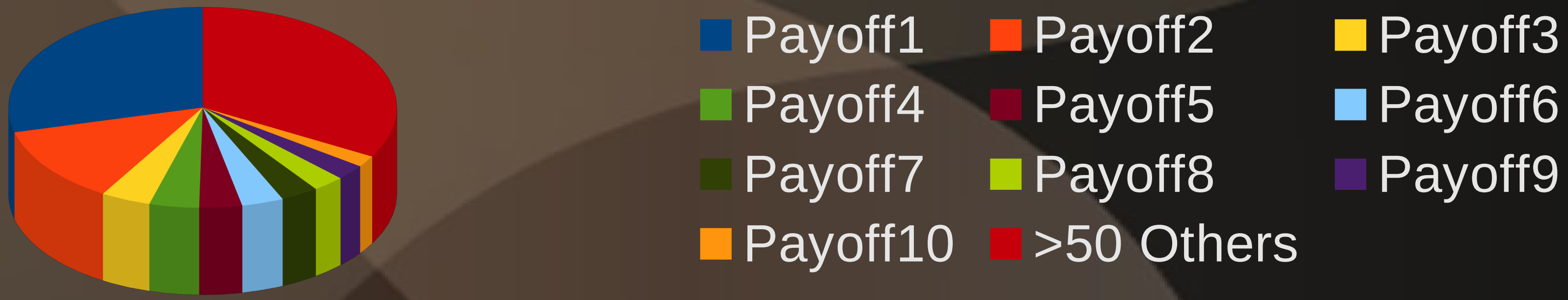
- Estimate the fair value of a contract
  - Monte Carlo Models
  - PDE Models
  - Tree Models

Monte Carlo

- For (instrument in portfolio)
  - For (scenario...)
    - For(path...)
      - Generate random numbers
      - Use model to generate asset paths
      - Calculate value of payoff

- Order of 10,000,000,000 times in valuing a portfolio
- Payoff calculation can dominate execution time.

Workload by Payoff



- Payoffs are Parametrised
- Small number of payoffs are very common
- Long tail of uncommon payoffs

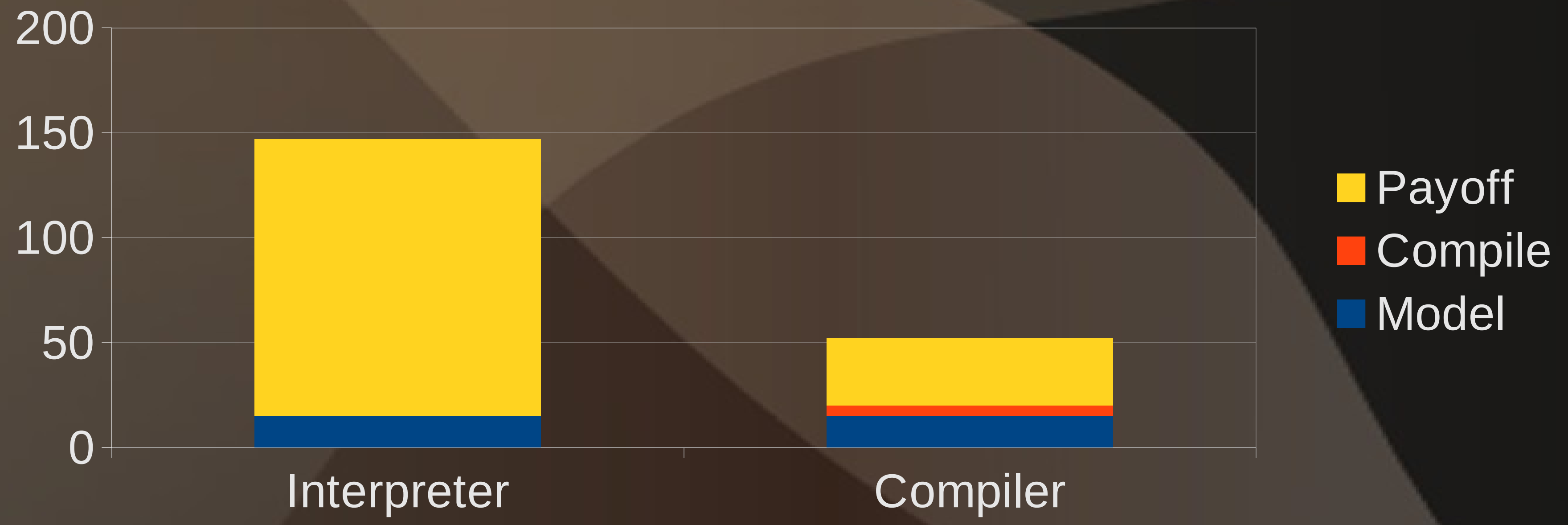
Outline

- Introduction
  - What, How, and Why do we use DSLs in Finance?
- Implementation
  - Interpreting, Compiling
- Optimisations
  - Parallelism
  - How Fast Can It Be?
  - What Optimisations Are Important?

How can we execute them

- Parse Payoff
  - Generate an Abstract Syntax Tree
- Interpret Payoff
  - Software controlled execution
  - Works best with fewer, slower instructions
- Compile Payoff
  - Hardware controlled execution

Interpreters and Compilers



Interpreters on GPU

- Interpreter on CPU, launch kernels on GPU
  - Memory Overhead Of Interpreter
  - Flush intermediate values to global memory from registers
  - Divergence Management
- Interpreter can run on GPU
  - Interpreter state needs to be on GPU
  - A memory overhead in storing the interpreter state

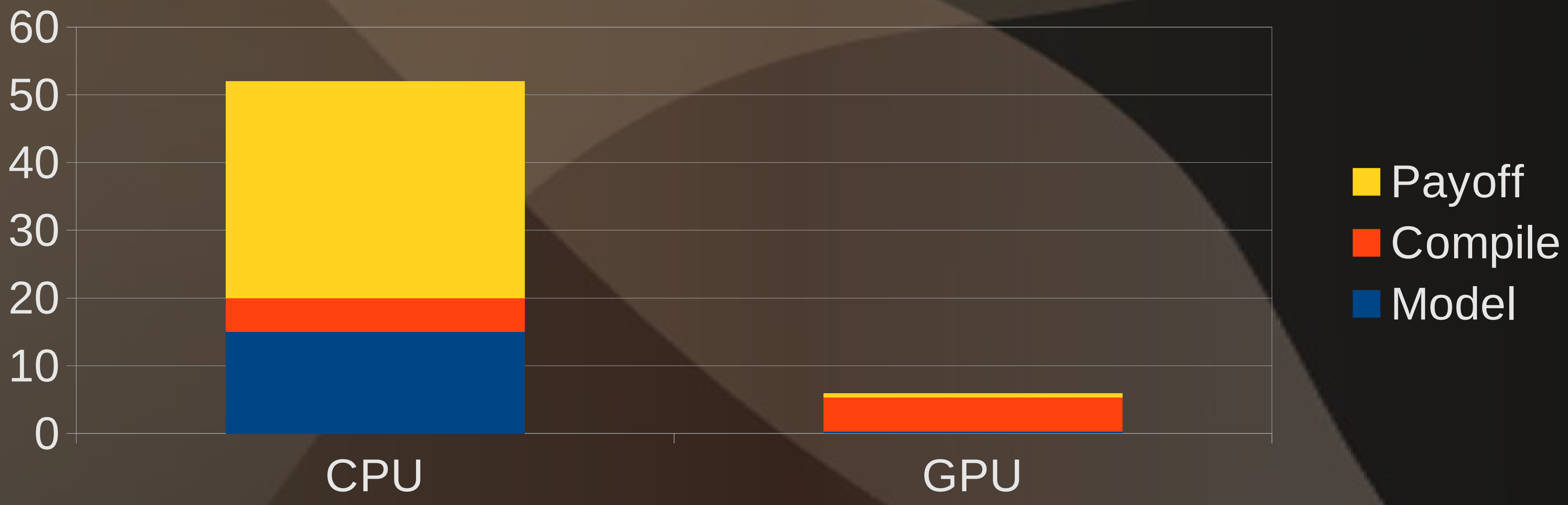
Compiling The Payff

- Somehow compile a function on the GPU which meets a specific API
  - Takes a description of how many paths there are
  - How they are laid out in memory
  - Where to write output (payment data)
- Have your MC framework call that function
  - Typically with a large number of paths
  - Possibly many scenarios

JIT compilers

- A JIT compiler gives you runtime information:
  - Numbers of Assets
  - Numbers of Timesteps
- This lets you do
  - In-lining of parameters
  - Loop unrolling
  - Memory Allocation
- This costs you
  - Compilation Time

Speed and Latency



JIT Compilers for GPU

- Amdahl's Law
  - Compilation is sequential
  - JIT Compilation not a bottleneck on CPU
  - JIT Compilation may limit GPU performance
- Caching Required
  - Compiled Payoffs Must Be Reusable
  - Compile-In Per-Instrument Constants ?

Compilation Methods

- Many different routes to compiled code
  - CUDA
  - NVVM
  - PTX
  - OpenCL
  - Others

Compiling the Payoff for GPU

- Cross-compilation to CUDA C
  - Compile with NVCC
  - Produce a shared library object
  - Dynamically Load and Execute
- Pros:
  - Familiar language
  - Good compiler
- Cons:
  - Larger Compiler, Language and Libraries
  - Slower Compilation

Compiling the Payoff for GPU

- Cross-compilation to OpenCL
  - Similar effort to cross compilation to CUDA
- Pros:
  - Relatively Readable Compiler Output
  - Relatively Platform Portable
  - No need to distribute compiler
- Cons:
  - Slow Compilation
  - Difficult to integrate with CUDA

Compiling the Payoff for GPU

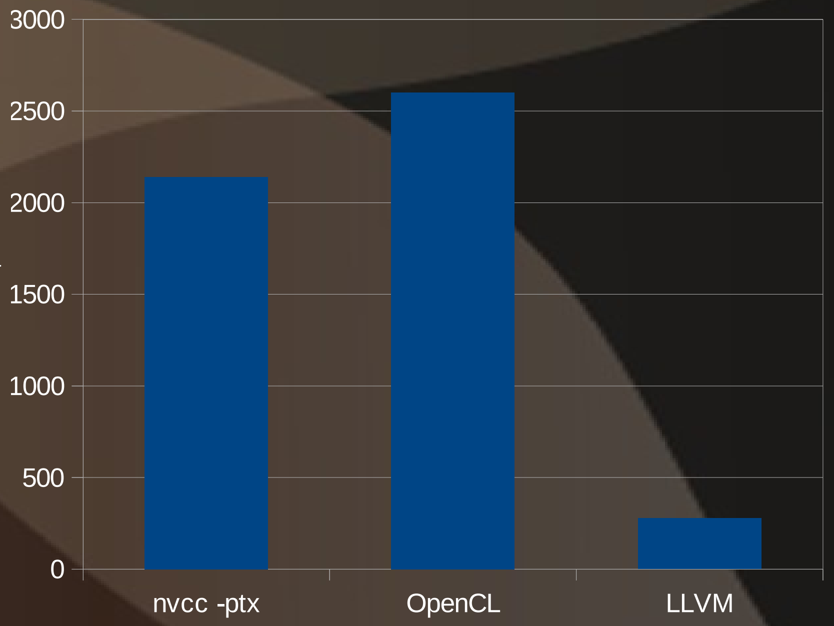
- Compile to PTX
  - PTX code can be translated by driver
- Pros:
  - Fast Compilation
  - No additional libraries or tools
- Cons:
  - Hard to debug
  - Built In Optimisation

Compiling the Payoff for GPU

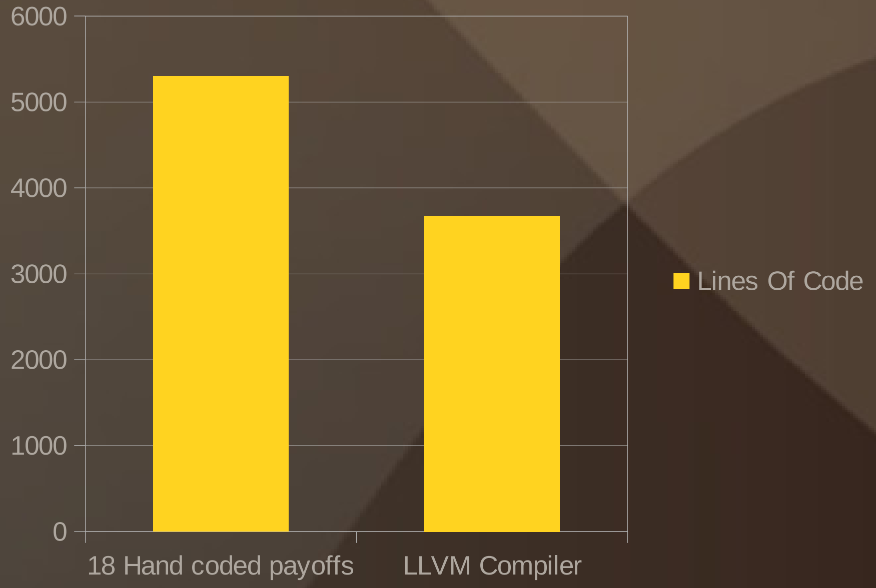
- Compile to NVVM
- Pros:
  - Good optimisation
  - Good debugging tools
  - Easy to adapt to CPU
  - Very little to distribute
- Cons:
  - Learning curve

NVVM

- NVVM was best option for us
  - Existing Experience with LLVM
  - Ease of Adaptation to CPU
  - Fast Compile Times



How hard is it?



- LLVM does a lot of work for you
- A compiler is still complex
- Less daunting than hand coding payoffs?

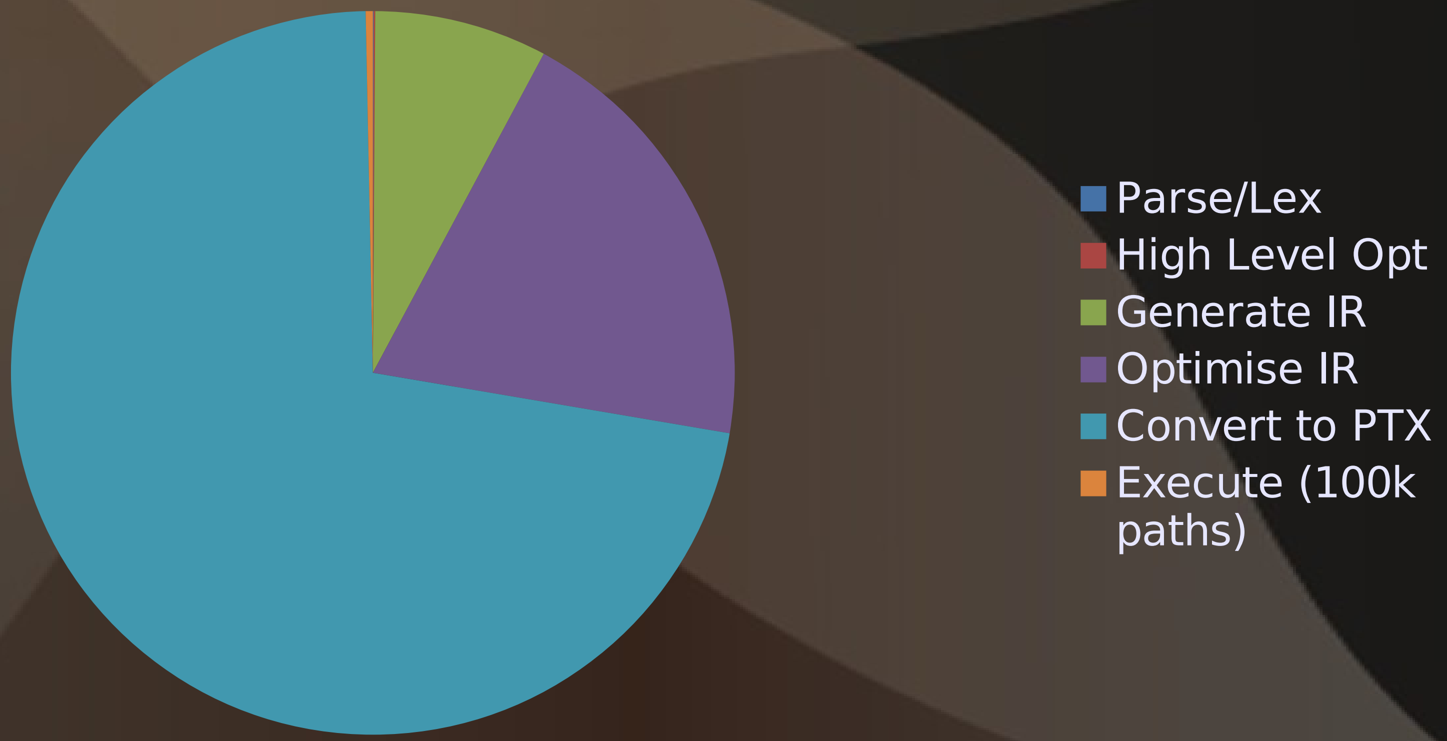
Compiling the Payoff for GPU

- SPIR Khronos group
  - An intermediate language for OpenCL
  - LLVM based
  - Not available yet
- HSAIL HSA foundation
  - Another standard for GPU intermediate languages
  - Also LLVM based

Compilation Strategy

- We know we are in an inner loop
  - In-line everything
  - Unroll all loops
- Use LLVM constants
  - LLVM will pre-calculate constant expressions
  - Move them outside of inner loop
- LLVM vector types
  - Good for CPU performance

Compilation Is Still A Bottleneck!



Outline

- Introduction
  - What, How, and Why do we use DSLs in Finance?
- Implementation
  - Interpreting, Compiling, and Parallelising
- Optimisations
  - What Optimisations Are Important?
  - How Fast Can It Be?
  - Parallelism

Optimisations

- Optimisation For Free!
  - LLVM includes configurable optimization passes
  - NVVM include optimisations
  - PTX is further optimised during compilation
- Your compiler can emit quite bad code...
  - But not everything can be optimised for you

Performance of GPU Payoffs

- Easily Become Memory Bound
  - CPU : Single Path fits in cache
  - GPU : Thousands of Paths, cache ineffective
- Memory Usage Optimisation Effective
  - Global Writes/Reads Cannot Be Eliminated Safely
  - Avoiding reading or writing intermediate data

Example

Asian option on Worst-of-basket

WorstPerf=Avg(Min(Performance(x) for x in basket) for t in times)

Pay(Max(WorstPerf-Strike,0))

- Naive

  Performance(x) for x in basket

  1. Generates a list of length basket size
  2. Writes to a temporary
  3. Read temporary list and find Min
  4. Minimum written to second list of length times

- Global Memory Read/Writes

Higher Order Functions

- Represent List Operations as Maps and Folds
- Fuse them!
- Fold(f(x,y),i,{a,b,c}) =

  f(f(f(i,a),b),c)
- Min(Peformance(x) for x in basket) =
  - Fold (Min(x,y) , inf, Map(Peformance(x),basket))
- Min(Peformance(x) for x in basket) =
  - Fold(Min(Performance(x),Performance(y)), inf, basket)

How Fast Can It Be?

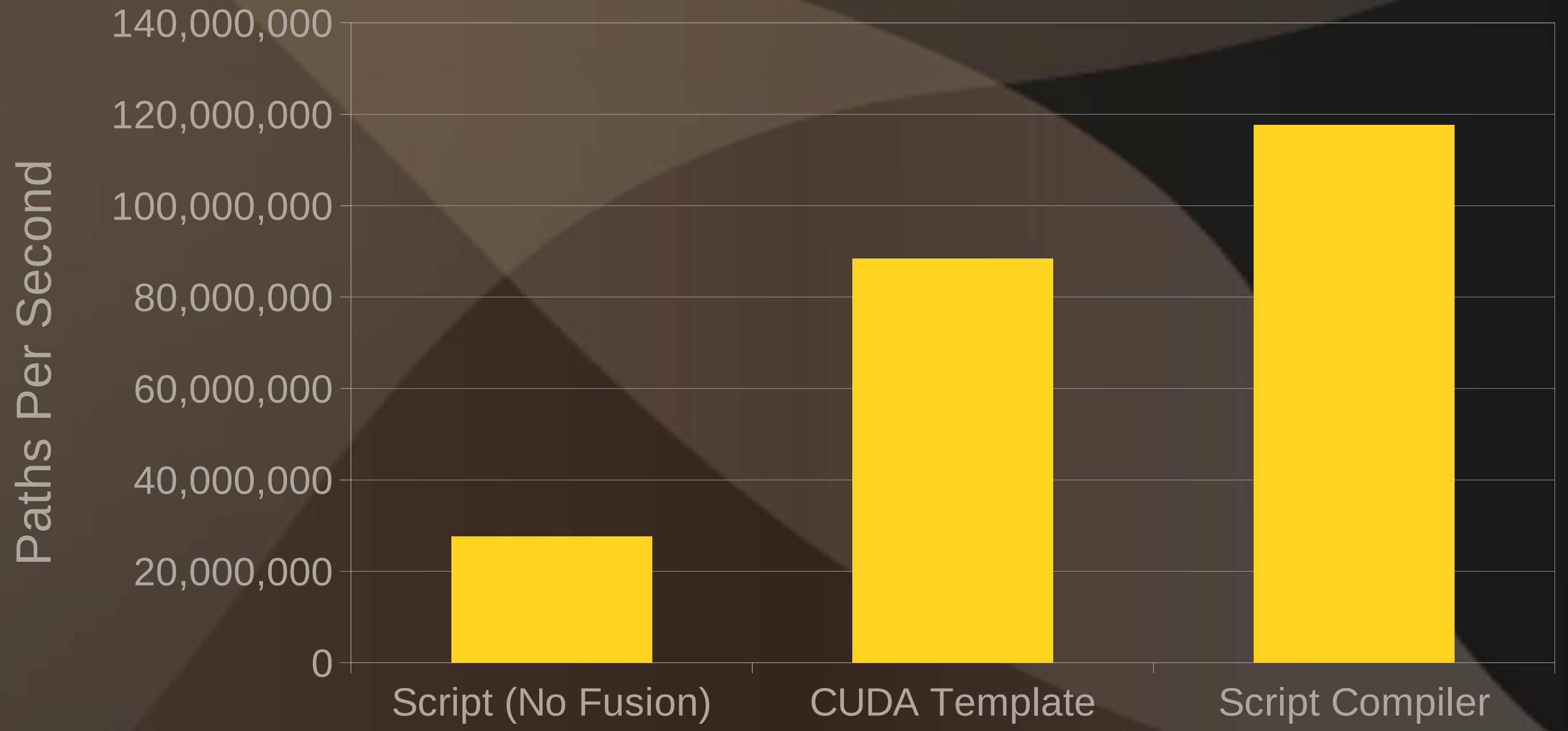
WorstPerf=Avg(Min(Performance(x) for x in basket) for t in times);

Pay(Max(WorstPerf-Strike,0))

```
__global__ void     worstOfAsian(
        const PathReader* pathReader,
        const size_t nTimeSteps,
        const size_t nPaths,
        const size_t nEquityUnderlyings,
        const float strike,
        const PaymentWriter*  paymentWriter)
    {

        const size_t iPath = threadidx.x + blockidx.x*blockdim.x;
        if (iPath>=nPaths) return;
        float average = 0;
        for (size_t iTime=0; iTime<nTimeSteps; ++iTime)
        {
            float worstPerf=1E36;
#pragma unroll 4
            for (size_t iAsset=0;iAsset<nEquityUnderlyings;++iAsset)
            {
                float myInitialSpot = pathReader->read(iAsset,iPath,0);
                float mySpot =  pathReader->read(iAsset,iPath,iTime);
                float myPerf=mySpot/myInitialSpot;
                if (myPerf<worstPerf) worstPerf=myPerf;
            }
            average+=worstPerf;
        }
        average/=nTimeSteps;
        float payoff = max(average-strike,0.0f);
        paymentWriter->write(payoff);
    }
```

How Fast Can It Be?



Scripting Difficulties

- Choose Appropriate FP Precision
  - Single Precision Often, Not Always, Sufficient
  - Allow Users To Specify Precision?
  - Always Use Double Precision?
- Effective Use of Shared Memory
  - Shared Memory can cache intermediate results
  - Where this is useful and appropriate, it is a huge performance boost

Conclusion

- Scripting Languages Can Be Executed Efficently on GPU
- Interpreter Overhead is High
- JIT Script Compilation Can Be A Bottleneck
  - Caching Is Essential
  - Trade-off between speed and latency
- NVVM An Excellent Tool For Compiling Payoffs
- Higher Order Functions And Fusion Give Good Performance