ArBB, Haskell and Accelerate

Summary of Internship at Intel

Joel Svensson
Supervisor: Ryan Newton

Array Building Blocks

- Data-parallel programmig model
  - High level collective operations
  - Threading and SIMDization – “for free”
  - Portability by JIT compilation
    - Makes use of the resources at hand
      - SSE/AVX
      - Threading
      - Acceleration devices* MIC, GPU
- C++ Interface
- Low level C API to the VM

* future work (?)

ArBB C++ example

C++

```
void arbb_dotprod(dense<f32> a, dense<f32> b, f32& result) {
   result = add_reduce(a * b);
}
```

ArBB C++ example

C++

```
void dot_product(dense<f32> a, dense<f32> b, f32& result) {
   result = add_reduce(a * b);
}
```

+ some “glue code”

C++

```
dense<f32> va, vb;
f32 vc;
bind(va, a, SIZE);
bind(vb, b, SIZE);
call(dot_product)(va, vb, vc);
```

There is another interface for getting data “into” arbb but its associated glue code is lengthier.

ArBB VM-API example

C

```
  arbb_function_t function;
  arbb_begin_function(context, &function, fn_type,"dot", 0, NULL);
  arbb_variable_t a, b, c;
  enum { is_input, is_output };
  arbb_get_parameter(function, &a, is_input, 0, NULL);
  arbb_get_parameter(function, &b, is_input, 1, NULL);
  arbb_get_parameter(function, &c, is_output, 0,NULL);
  arbb_variable_t tmp[1];
  arbb_create_local(function, tmp, dense_1d_f32, 0, NULL);
  arbb_variable_t in[] = { a, b };
  arbb_op(function, arbb_op_mul, tmp, in, 0, NULL);
  arbb_variable_t result[] = { c };
  arbb_op_dynamic(function, arbb_op_add_reduce, 1, result, 1, tmp, 0, NULL);
  arbb_end_function(function, NULL);
  arbb_compile(function, NULL);
```

Source: (slides with given title, no author specified)
Intel® Array Building Blocks
VM Interface Example

Haskell Bindings

- Using C2HS
  - By Manuel Chakravarty (UNSW).
  - Automates part of the bindings writing.
  - Still, writing bindings is very repetitive.
- Very direct mapping of the VM API into haskell
  - Each VM API C function gives a Haskell function in the IO Monad (means imperative programming)

An Example of C2HS

C

```
arbb_error_t arbb_create_global(arbb_context_t context,
                                arbb_global_variable_t* out_var,
                                arbb_type_t type,
                                const char *name,
                                arbb_binding_t binding,
                                void* debug_data_ptr,
                                arbb_error_details_t* details);
```

C2HS

```
{# fun unsafe arbb_create_global as createGlobal'
   { fromContext  `Context'     ,
     alloca-      `GlobalVariable' peekGlobalVariable* ,
     fromType     `Type'        ,
     withCString* `String'      ,
     fromBinding  `Binding'     ,
     id           `Ptr ()'      ,
     alloca-      `ErrorDetails' peekErrorDet*  } -> `Error' cToEnum #}
```

+ a small haskell wrapper

Haskell

```
createGlobal :: Context -> Type -> String -> Binding -> IO GlobalVariable
createGlobal ctx t name b =
  createGlobal' ctx t name b nullPtr >>=
  dbg "arbb_create_global" [("ctx",show $ fromContext ctx),
                            ("t" ,show $ fromType t),
                            ("name",name),
                            ("bind",show $ fromBinding b)]  ("GlobalVar",fromGlobalVariable) >>=

  throwIfErrorIO1
```

Repeat X times

```
arbb_get_dense_type
arbb_get_nested_type
arbb_create_constant
arbb_create_global
arbb_get_variable_from_global
arbb_get_function_type
arbb_get_function_type_parameter_alias
arbb_begin_function
arbb_abort_function
arbb_end_function
arbb_get_parameter
arbb_serialize_function
arbb_create_local
...
```

Same example using Haskell ArBB

Haskell

```
dotprod <- funDef_ "dotProd" [sty] [dty,dty] $ \[out] [in1,in2] -> do
   tmp <- createLocal_ dty "tmp"
   op_ ArbbOpMul [tmp] [in1,in2]
   opDynamic_ ArbbOpAddReduce [out] [tmp]
```

Full example PART 1

Haskell

```
main = arbbSession$ do
     sty <- getScalarType_ ArbbF32
     dty <- getDenseType_ sty 1

     dotprod <- funDef_ "dotProd" [sty] [dty,dty] $ \[out] [in1,in2] -> do
       tmp <- createLocal_ dty "tmp"
       op_ ArbbOpMul [tmp] [in1,in2]
       opDynamic_ ArbbOpAddReduce [out] [tmp]

     <.. continue on next slide ..>
```

Full example PART 2

Haskell

```
withArray_ (replicate (2^24) 1 :: [Float]) $ \ in1 ->
   withArray_ (replicate (2^24) 1 :: [Float]) $ \ in2 -> do
     inb1 <- createDenseBinding_ (castPtr in1) 1 [2^24] [4]
     inb2 <- createDenseBinding_ (castPtr in2) 1 [2^24] [4]

     gin1 <- createGlobal_ dty "gin1" inb1
     gin2 <- createGlobal_ dty "gin2" inb2

     vin1 <- variableFromGlobal_ gin1
     vin2 <- variableFromGlobal_ gin2

     outb <- getBindingNull_
     g    <- createGlobal_ sty "res" outb
     y    <- variableFromGlobal_ g

     execute_ dotprod [y] [vin1,vin2]
     result :: Float <- readScalar_ y

     liftIO$ putStrLn $ show result
```

Data.Array.Accelerate

- EDSL for DP programming
  - Embedded in Haskell
  - CUDA – GPU backend
  - Similar model to ArBB

Accelerate and ArBB features

Accelerate	**ArBB**
Map f	map f
ZipWith f	map f
Fold f	reduce_add reduce_mul ...
Scan f	scan_add scan_mul ...
FoldSeg	N/A
ScanSeg	N/A
Stencil	map
Permute/BackPermute	gather scatter unpack Pack shuffle unshuffle

Accelerate and ArBB features

Accelerate	**ArBB**
Map f	map f 
ZipWith f	map f 
Fold f	reduce_add reduce_mul ... 
Scan f	scan_add scan_mul ...
FoldSeg	N/A
ScanSeg	N/A
Stencil	map
Permute/BackPermute	gather scatter unpack Pack shuffle unshuffle

Accelerate example

Haskell

```
dotpAcc :: Vector Float -> Vector Float -> Acc (Scalar Float)
dotpAcc xs ys
  = let
      xs' = use xs
      ys' = use ys
    in
    fold (+) 0 (zipWith (*) xs' ys')
```

Accelerate example

Haskell

```
dotpAcc :: Vector Float -> Vector Float -> Acc (Scalar Float)
dotpAcc xs ys
  = let
      xs' = use xs
      ys' = use ys
    in
    fold (+) 0 (zipWith (*) xs' ys')
```

C++

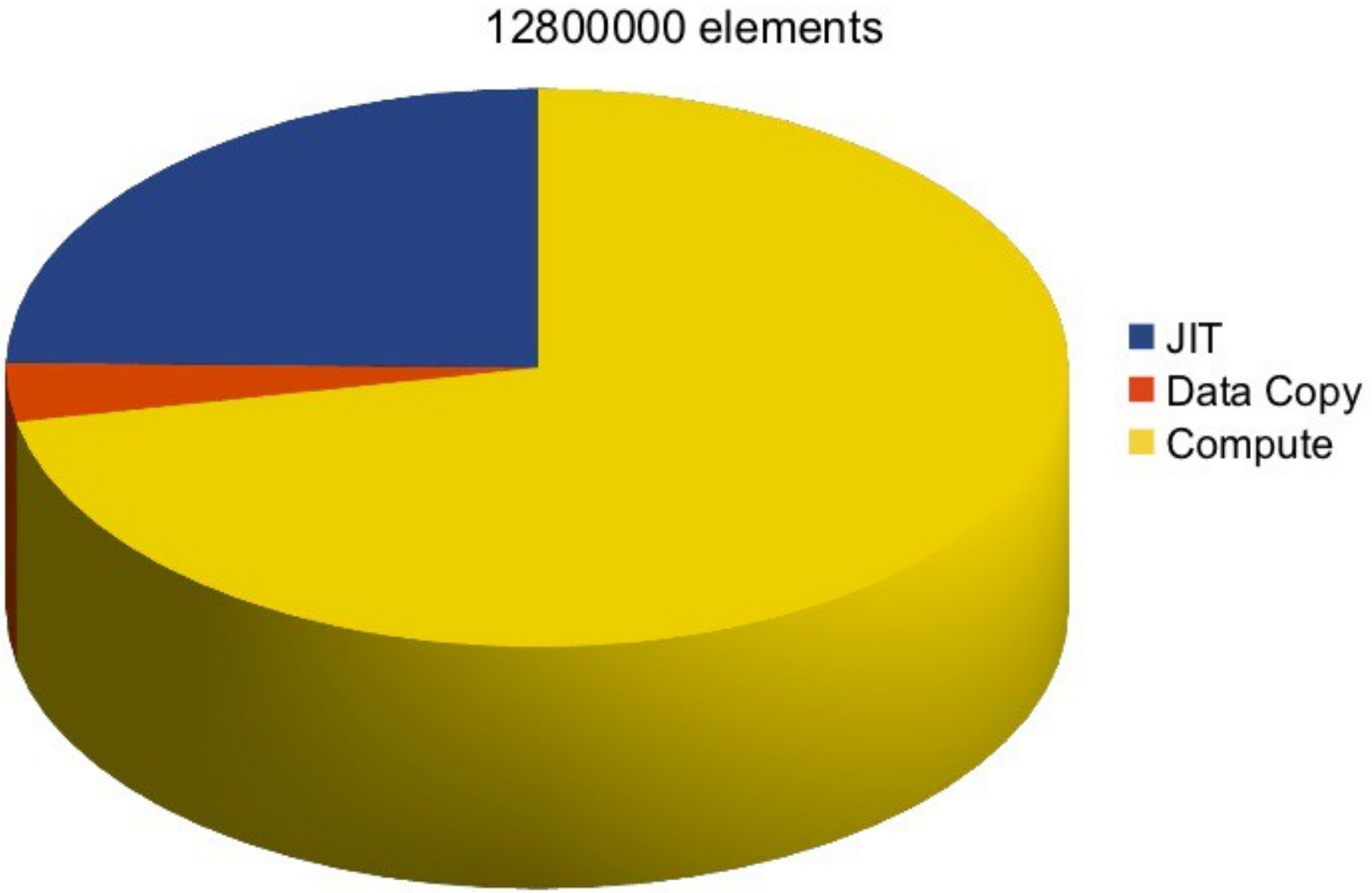
```
    void arbb_dotprod(dense<f32> a, dense<f32> b, f32& result) {
       result = add_reduce(a * b);
    }
```

Benchmarks

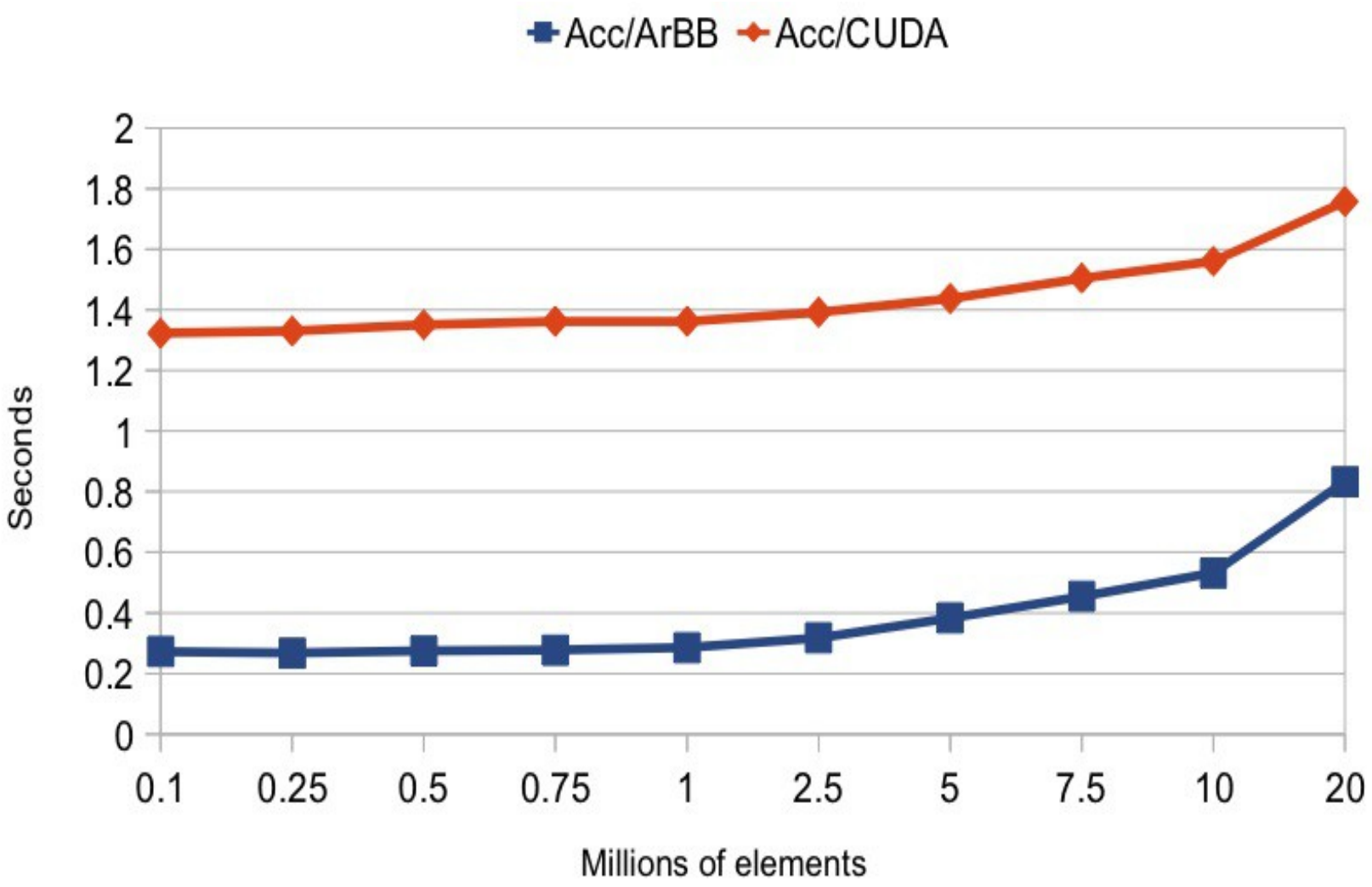
- The Accelerate ArBB backend is partial but enough for small benchmarks.
  - DotProduct
  - Sum
  - SAXPY
  - Black-Scholes
    - Charts on Black-Scholes on following slides.

Black-Scholes

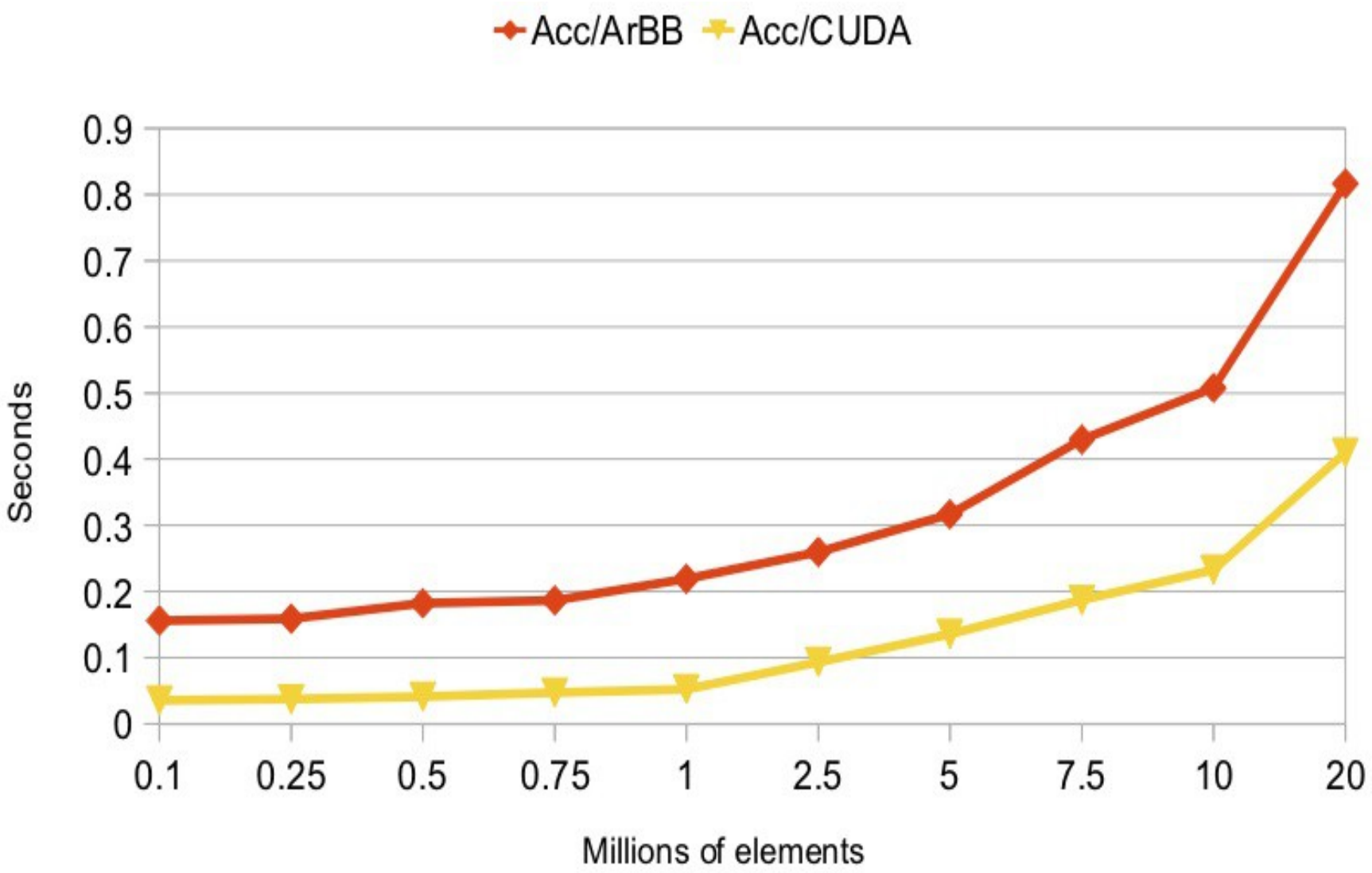
Transfer - JIT - Compute



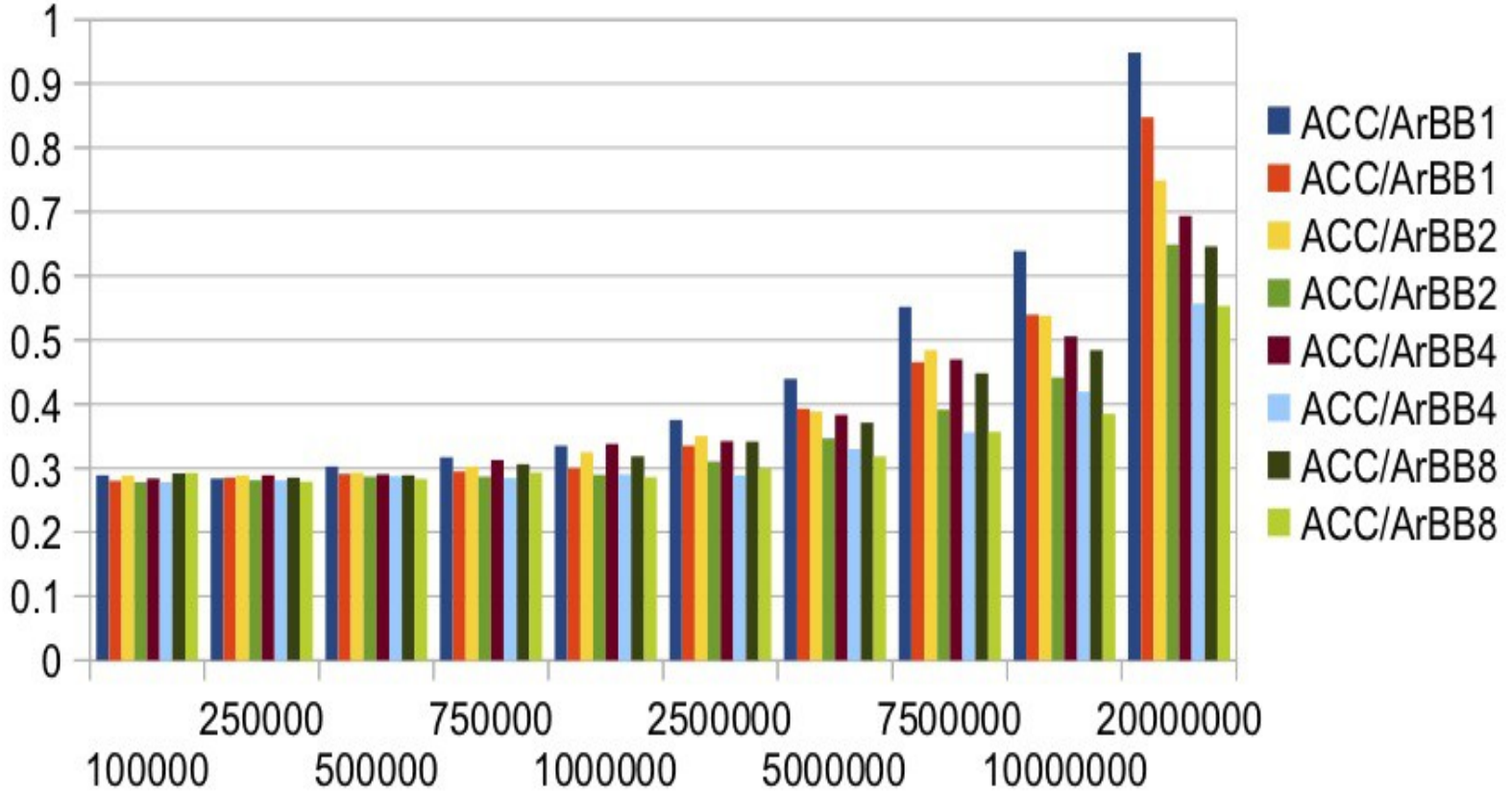
Black-Scholes



Black-Scholes



Black-Scholes



Results

- Haskell ArBB bindings are operational
  - Very low level interface (VM API)
  - I want to know more about ArBB garbage collector.
  - Will be released on “Hackage” for other Haskellers to use.
- Accelerate ArBB backend is partial
  - Preferred programming interface.
    - Beats the C++ interface when it comes to necessary “glue code”
    - Gets beaten on performance!
  - Able to run small benchmark.
  - TODO-list is long!
  - Accelerate team is open to including the ArBB backend in the future.
- We submitted a “position paper” to workshop FASPP 11
  (Future Architectural Support for Parallel Programming, @ ISCA)