Hybrid Vector Library—From Memory Bound to Compute Bound with NVVM

Régis PORTALEZ — ALTIMESH — regis.portalez@altimesh.com — Florent DUGUET — ALTIMESH— florent.duguet@altimesh.com

MOTIVATION

Existing source code usually interleaves data management, error-checking, text processing and actual compute. On general purpose processors, this mixture of code tasks is not necessarily an issue, and performance levels are often satisfactory as is.

However, when trying to use GPU, this hybrid computing turns into a coding challenge. Each individual computing tasks does not show sufficient workload, and porting the whole application requires a significant investment in the software asset.

We propose an alternate approach with runtime compilation based on function calls on a compute library. Hybrid Vector Library operates on vectors, in a manner similar to BLAS level 1 routines, with other functions such as square root or exponential, or MKL routines. In essence, all operations are performed on a vector of values. We illustrate the performance results of this approach on a typical financial benchmark.

Existing solutions such as ArrayFire [5] do not allow custom device function to be called in the middle of a level 1 routines sequence. We address that issue by processing these functions at compile time.

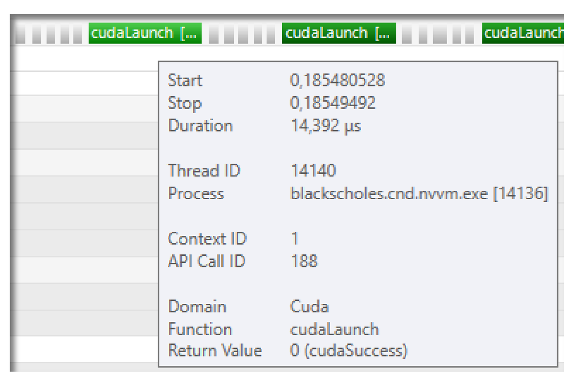
HYBRID VECTOR LIBRARY

Similar to MKL or BLAS Level-1 routines, Hybrid Vector Library exposes operations on vectors of values. These operations include basic arithmetic operations, along with mathematical function calls. It also exposes comparison tools and select operation to support basic value-dependent branching operations.

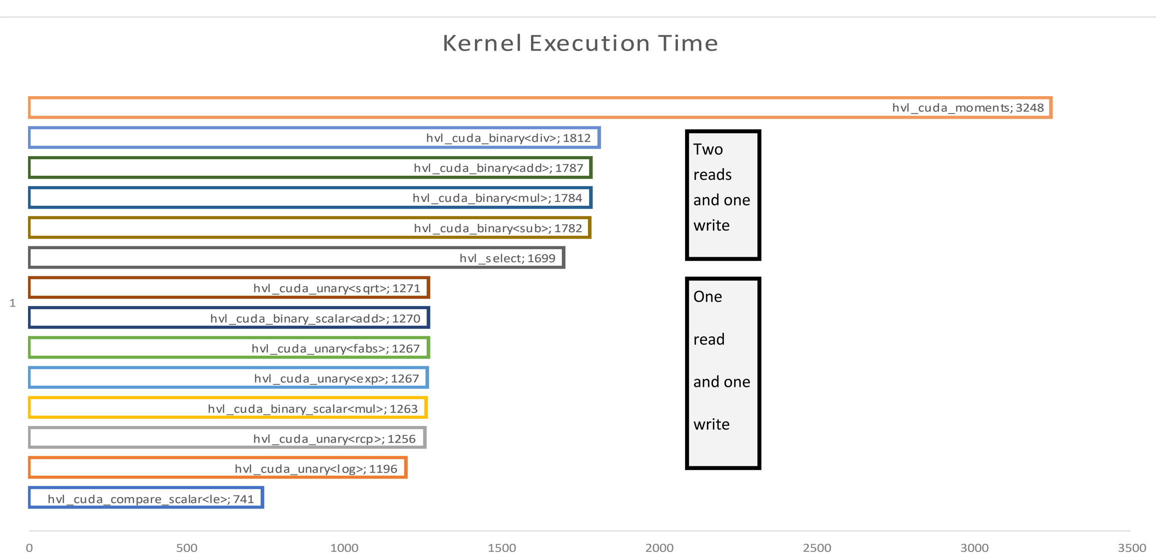
The API has several implementations that can be chosen at runtime to allow maximal flexibility. We illustrate here the use of two of these implementations.

PERFORMANCE OF NAIVE IMPLEMENTATION

The naïve implementation will perform a kernel call for each vector operation. Beyond the lack of compiler optimization that would for example reconstruct FMA operations, this implementation suffers an important performance penalty. Indeed, each kernel call needs to be scheduled and executed. As illustrated in the following profiling snapshots, the execution time of a launch is about 25 microseconds (10µs configuration and 15µs launch). Within this time, about 1 million vector entries can be processed (calculating exp or log of the values for instance)



Moreover, kernel executions are memory bound. Indeed, current GPUs can execute more than 50 FLOPS for each memory operation, making all simple math functions, including transcendentals such as exponential, memory bound. We can see performance is driven by memory operations and not arithmetic complexity.



When executing operations upon library API call, performance is memory-bound and kernel execution time solely depends on amount of memory read or written.

RUNTIME COMPILATION

Depending on the implementation of HVL, execution of the calculation is performed at different stages. For the basic implementation, execution is done upon the API call on a vector of data. When using the NVVM –backed version, intermediate results do not exist. Operations are done in four phases:

1 User-defined device functions are identified in the call graph. CUDA source is generated for each of them, as long as a cubin file. The pairs function/cubin is registered at application startup.

2 When calling API methods, the operations are not scheduled immediately on the device. The different calls are gathered in a graph, which is by construction directed and acyclic (DAG), and no operation is executed until results are queried.

3 At given milestones, the DAG is converted into NVVM source code: each node is an NVVM statement with a single output. The NVVM source code is compiled at runtime. The sequence of calls and the compilation result are cached for future usage.

4 The resulting device binary is then scheduled for execution and results can be queried. The DAG is encoded into a signature in order to cache the compilation results — CUDA binary module. As shown, the compilation time may be longer than the overall execution time.

C++ Application Code (1)

```cpp
// scalar code
double BlackScholesBodyScalar(
    double Sf, //Stock price
    double Xf, //Option strike
    double Tf, //Option years
    double Rf, //Riskless rate
    double Vf  //Volatility rate
)
{
    double S = Sf, X = Xf, T = Tf, R = Rf, V = Vf;

    double sqrtT = sqrt(T);
    double    d1 = (log(S / X) + (R + 0.5 * V * V) * T) / (V * sqrtT);
    double    d2 = d1 - V * sqrtT;
    double CNDD1 = CND(d1);
    double CNDD2 = CND(d2);

    double expRT = exp(- R * T);
    return (S * CNDD1 - X * expRT * CNDD2);
}

// vector code
// earlier : mycnd has been declared extern "C" for symbol to be retrieved
hvlvect BlackScholesBodyVect(const hvlvect& S,   const hvlvect& X, const hvlvect& T,
           const hvlvect& R, const hvlvect& V)
{
     hvlvect VsqrtT = V * sqrt(T);
     hvlvect     d1 = (log(S / X) + (R + 0.5 * V * V) * T) / (VsqrtT);
     hvlvect     d2 = d1 - VsqrtT;
     hvl_invoke(d1, mycnd);
     hvl_invoke(d2, mycnd);   (2)
     hvlvect expRT = exp(-R * T);
     return (S * d1 - X * expRT * d2);
}
```

(1) In this implementation all calls are within the same function but it can be spread on multiple source files or binary modules.

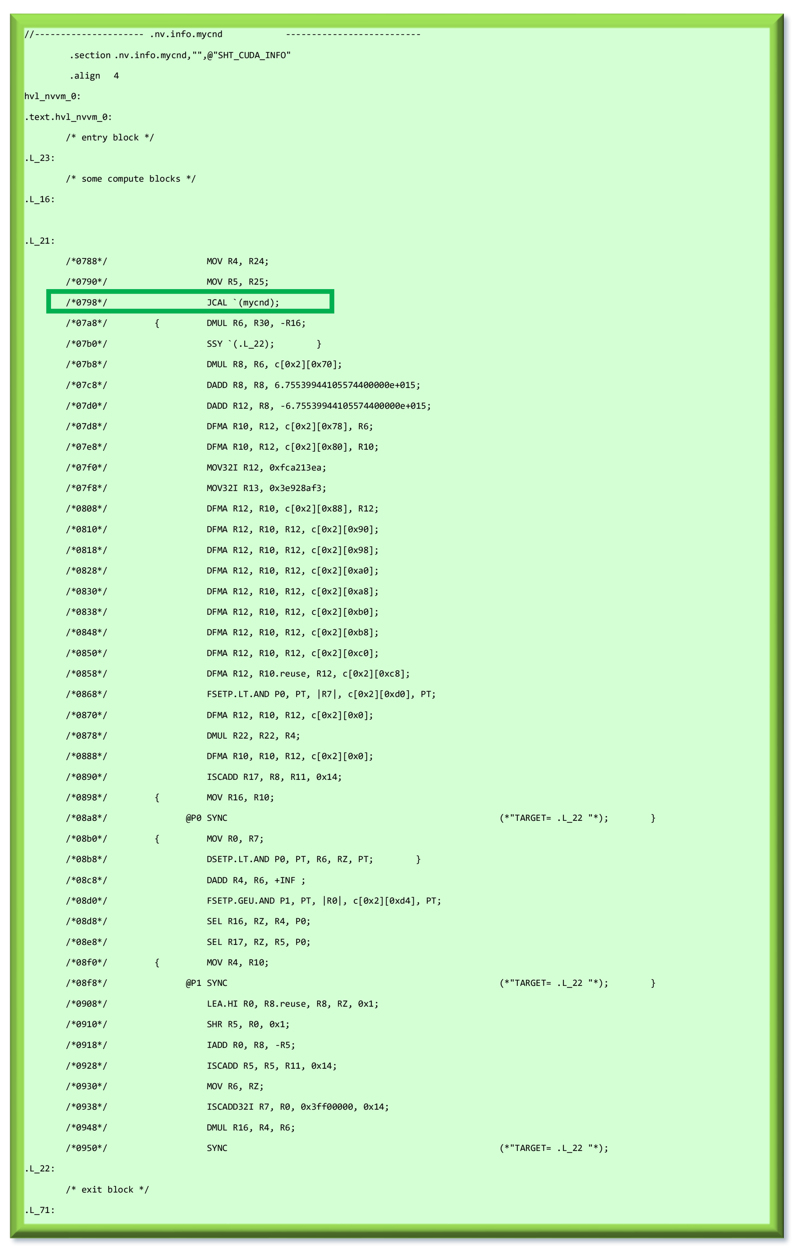
Vector operators are overloaded in C++ to make use of hvl library and include error-checking using exceptions. User defined device code is invoked through special API calls.

HVL API Calls

```
hvl_create
...
hvl_assign_hybridvector
hvl_apply_sqrt
hvl_assign_hybridvector
hvl_mul
hvl_assign_hybridvector
hvl_mul
hvl_mul_scalar
hvl_add
hvl_mul
hvl_assign_hybridvector
hvl_div
hvl_apply_log
hvl_add
hvl_div
hvl_assign_hybridvector
hvl_sub
HVL_invoke
HVL_invoke
hvl_assign_hybridvector
hvl_mul
hvl_mul_scalar
hvl_apply_exp
hvl_assign_hybridvector
hvl_mul
hvl_assign_hybridvector
hvl_mul
hvl_mul
hvl_sub
hvl_destroy
...
```

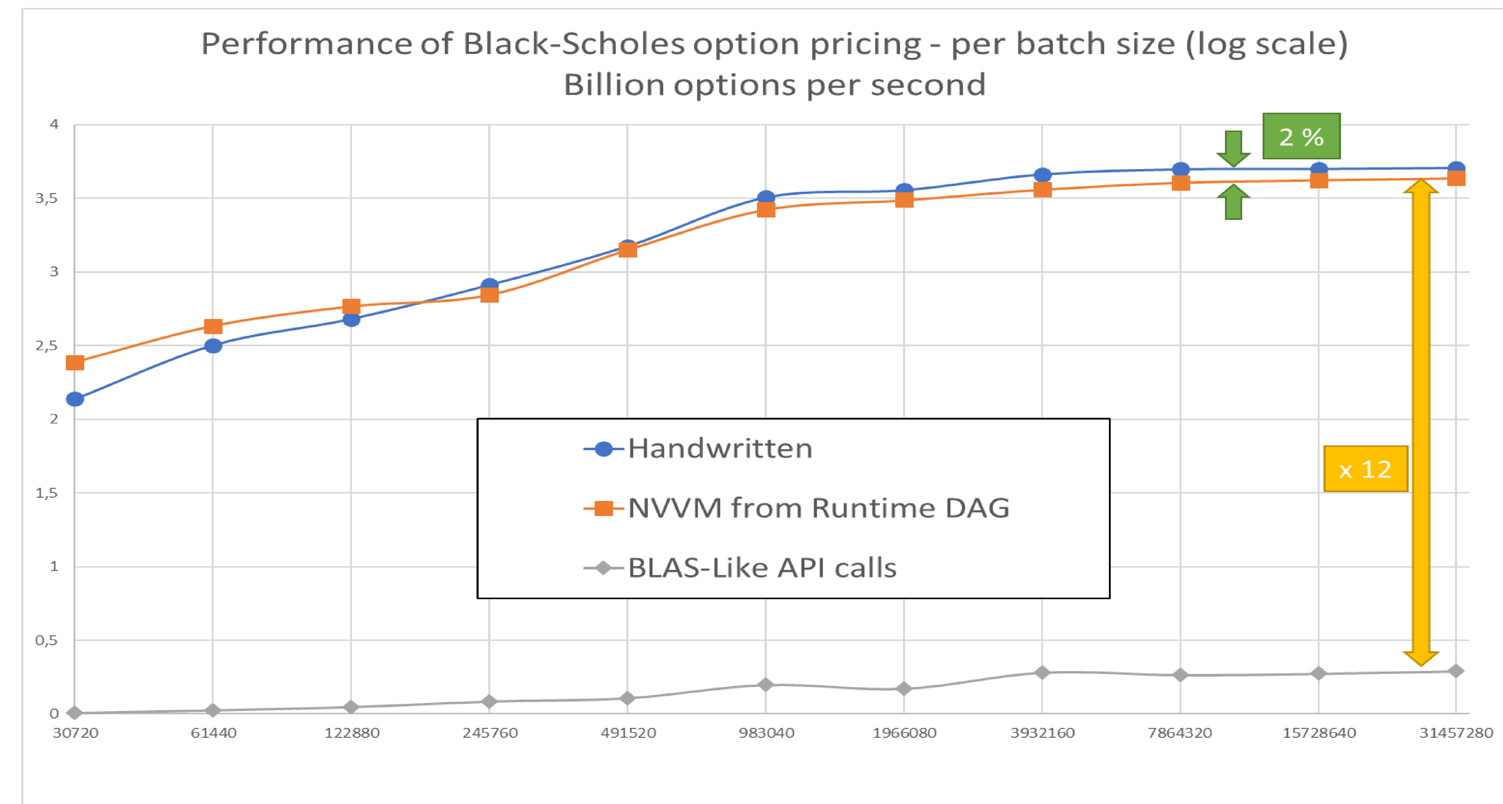
From HVL API Calls, a DAG is generated and upon request transformed into NVVM source code.

SASS Generated Code



A cubin file is generated at runtime and linked with precompiled modules of custom device functions.

NVVM Generated Code



Execution of same algorithm with same launch settings (120 blocks—256 threads on a Tesla K40c with CUDA 8.0 on a variety of options count)

BENEFITS OF USER-DEFINED FUNCTIONS

In the case of complex algorithm, for example when branching cannot be converted into functions like maximum, the set of methods exposed in the library are not necessarily sufficient for a single source implementation. It is sometimes necessary to either implement kernels by hand (in which case one per architecture), or retrieve data on the CPU losing significant performance benefit from the approach.

Enabling user-defined functions, it is possible for the user to write a function in a single version, and with a customized compilation tool-chain that function can be invoked by all underlying implementations (host or device). Such functions are declared using supplemental attributes for the toolchain to connect between implementations.

PERFORMANCE OF RUNTIME COMPILATION OF DAG

As we can see in this table, the runtime compilation requires significantly more CPU time than the execution for sizes in the 100k range.

Compilation of NVVM code takes about 50 milliseconds which is much higher than most execution times. A good caching strategy is needed.

As a future work, we consider performing register allocation and PTX generation directly for DAG instances where initial cost cannot be amortized by caching strategy.

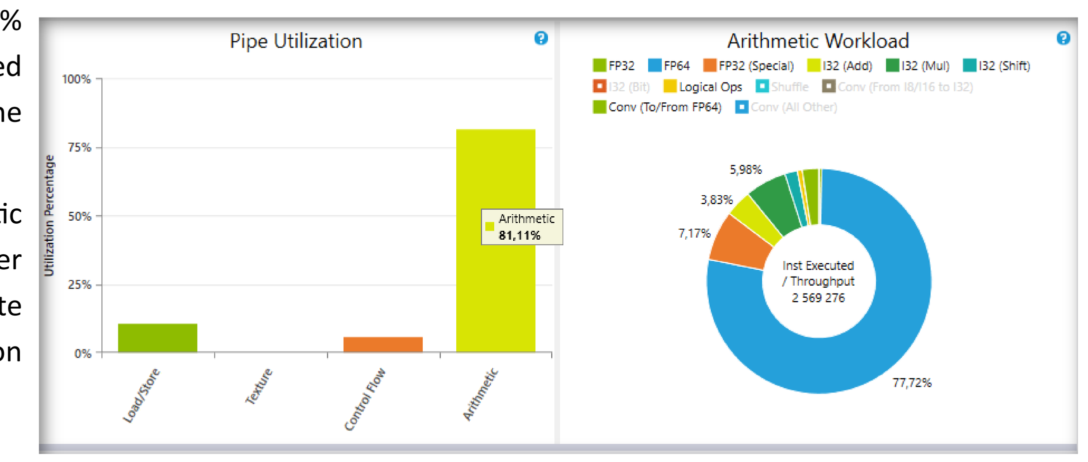
Task	Execution Time (micro seconds)
Converting DAG to NVVM	97.34
NVVM compilation to PTX	49,912.08
PTX compilation to CUBIN	1,674.64
CUBIN load	517.53

DISCUSSION

Whether due to kernel scheduling or systematic cache miss due to split kernel calls, execution of small tasks on a GPU lead to significant performance penalties. As a result, chosen approach is to perform a global porting of the application to GPU which is a tedious effort on long-lasting software assets.

We presented here an alternate solution that result in efficient execution of a queue of small GPU tasks, leveraging runtime compilation to avoid the cost of a kernel launch and cache miss on the device. With a good caching strategy, the overall performance is 98% of the performance obtained with a hand-tuned version of the same algorithm.

The utilization of the arithmetic pipe is above 80% on a Kepler K40 GPU, entering the "compute-bound" side of implementation class.



REFERENCES

[1] "Compiling Parallel Languages with the NVIDIA Compiler SDK", Mark Harris, supercomputing 2012

[2] "LambdaJIT: a dynamic compiler for heterogeneous optimizations of STL algorithms." Lutz, Thibaut, and Vinod Grover. *Proceedings of the 3rd ACM SIGPLAN workshop on Functional high-performance computing*. ACM, 2014

[3] "nvvm-IR documentation" : http://docs.nvidia.com/cuda/nvvm-ir-spec/index.html

[4] "Building GPU Compilers with libNVVM" Yuan Lin http://on-demand.gputechconf.com/gtc/2013/presentations/S3185-Building-GPU-Compilers-libNVVM.pdf

[5] "Array fire documentation" : http://arrayfire.org/docs/index.htm

Régis PORTALEZ — ALTIMESH — regis.portalez@altimesh.com
Florent DUGUET — ALTIMESH — florent.duguet@altimesh.com