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

Regis PORTALÉ — ALTIMESH — regis.portalez@altimesh.com — Florent DUGUET — ALTIMESH — florent.duguet@altimesh.com

MOTIVATION

Existing source code usually interfaces data management, error-checking, testing 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 task 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.

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 naive 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 snapshot, the execution time of a launch is about 25 microseconds (50% configuration and 50% launch). Within this time, about 2 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 transcendental (such as exponential), memory bound. We can see performance is driven by memory operations and not arithmetic complexity.

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 pears 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 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.

REFERENCES

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

Florent DUGUET — ALTIMESH — florent.duguet@altimesh.com

Regis PORTALÉ — ALTIMESH — regis.portalez@altimesh.com