Paper SASSD706-2017

Accelerate your SAS Programs with GPUs

Henry Bequet, Huina Chen, SAS Institute Inc.

ABSTRACT

Graphical Processing Units (GPUs) are quickly becoming a commodity on many desktops and servers. For less than $1 per core, a server can go from a few cores to thousands of cores. The major goal of the research project that we describe in this paper is to investigate how we can easily leverage the computing power of GPUs in SAS programs. To complete this project, we relied heavily on the Infrastructure for Risk Management solution (IRM), but the techniques that we describe in this article are applicable to plain old SAS MVA code.

Some programming, either in Base SAS or C, will be required to replicate the results that we describe in this paper. Readers with an advanced knowledge of Base SAS (9.4), the SAS Macro Language, and PROC FCMP will understand how to leverage GPU computing to achieve unsurpassed performance improvements in their solutions. Readers with an advanced knowledge of C will understand how to integrate their GPU-aware code to SAS programs.

IRM

The purpose of IRM is to provide SAS programmers with an easy to use development platform for the fastest analytics. To meet this goal, IRM leverages microservices (Fowler) and Many Task Computing (MTC) (Raicu 2008). In fact, IRM is a development platform for microservices written in SAS MVA code.

So how do you define a microservice in SAS MVA code? It turn out that it is fairly easy; simply follow those two rules:

1. Package your code in a .sas file where the input and output files are defined in metadata.
2. The inputs and outputs must be immutable.

That's it! All inputs and outputs must be files, typically a SAS dataset. An important corollary is that SAS code must not use any global state, typically global macro variables. All inputs and outputs must be defined; no inputs or outputs may be hidden; there are no exceptions.

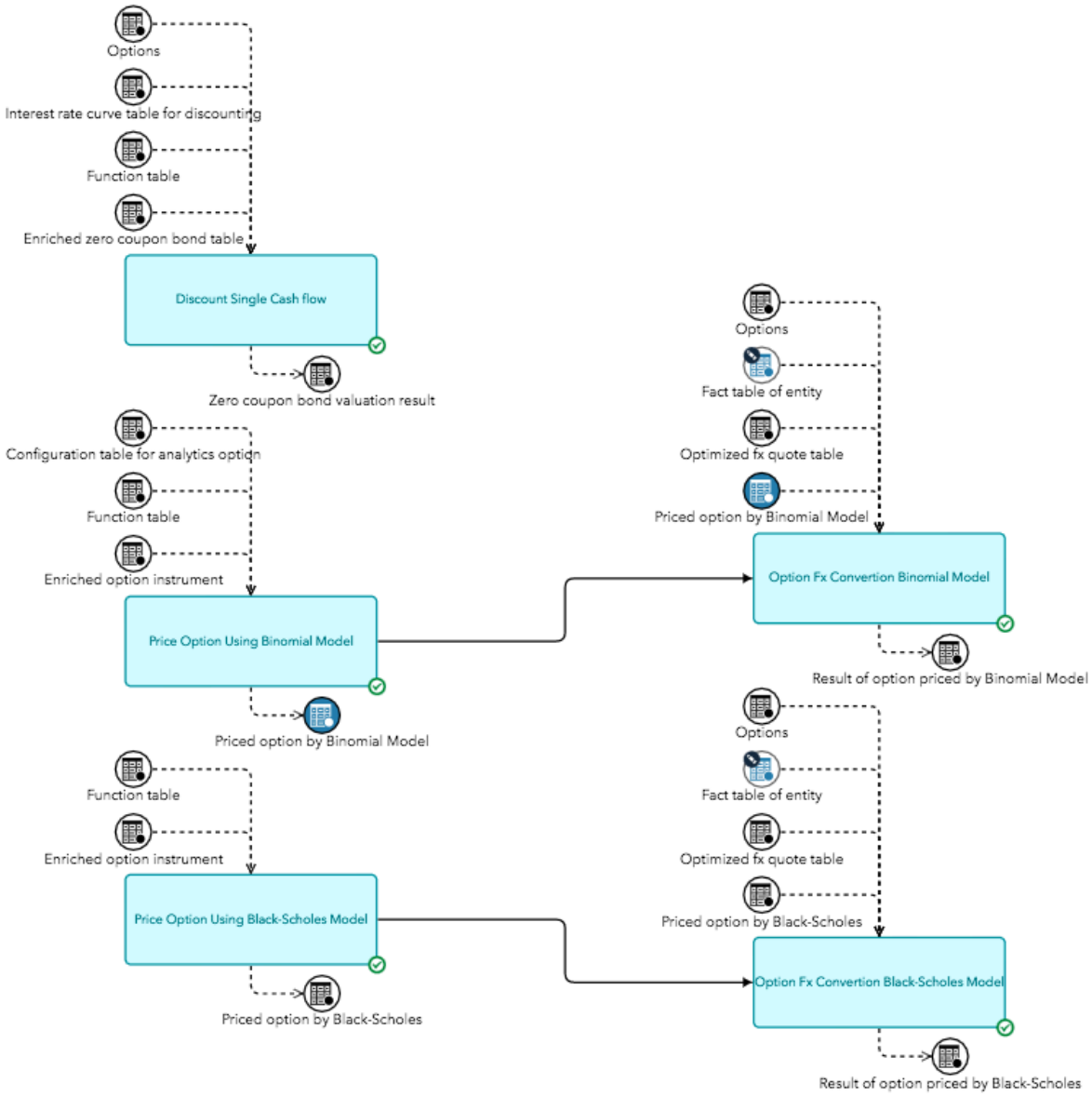

Figure 1. A Simple Job Flow

Armed with this information, IRM can schedule SAS code on thousands of cores without the SAS programmer worrying at all about synchronization. How is that possible? It is actually very simple.

The set of microservices and their inputs and outputs are called a job flow. When the focus is on job flows, SAS microservices are typically called tasks. Figure 1 shows an example of such a job flow. If you direct your attention to the tasks labeled ‘Price Option Using Binomial Model’ and the task labeled ‘Option Fx Conversion Binomial Model’, you’ll notice that they are joined by an arrow that indicates that one comes before the other. That order is implied because the data object highlighted in blue and labeled ‘Price option by Binomial Model’ is output of the first task and input to the second.

Regarding scheduling, it is worth pointing out that in the job flow of Figure 1 there are two (vertical) levels: the first one with 3 tasks and the second one with 2 tasks. Those levels actually constitute the tasks that are independent of each other and consequently that can run in multiple threads.

That implicit parallelism is key to the ease of use that IRM provides: SAS programmers can define very complicated parallel algorithms that can run on hundreds of machines and thousands of cores by simply defining the inputs and outputs of their tasks.

It is also possible to partition SAS data sets (e.g. with a by variable) to get even more parallelism. In that case IRM will schedule one instance of the task per partition and there are no communications between the task running on one partition and the task running on another partition. In other words, using partitions only works for embarrassingly parallel problems (Foster).

We will revisit in a few moments why this is important for GPU computing inside IRM, but first let's discuss GPUs and how we plan to use them.

GPU

Graphical Processing Units (GPUs) were initially designed to perform graphical operations like 3D scene rendering as fast as possible. Typically real time performance is required. In order for GPUs to perform graphical operations quickly, they execute the same algorithm on a lot of different data at the same time. In other words, GPUs have a high degree of parallelism: they run 1,000's of threads on 1,000's of cores. Super Computing engineers quickly realized that they could leverage the computational power of GPUs to perform non-graphical operations like solving a system of equations. This gave rise to the General Purpose GPU or GPGPUs. For the rest of this paper, when we refer to GPU, we really mean GPGPU.

If you look a little bit closer, you realize that GPUs excel at a particular class of parallel algorithms that exhibit at least two characteristics:

1. Require a lot of computations compared to the amount of data that needs to be processed. For instance, a good candidate for GPU computing is matrix multiplications and a bad candidate is matrix transposition.
2. Require identical processing on many versions of the data.

Without getting into the details of the architecture of GPUs that is outside of the scope of this paper, an easy way to visualize the computational model of a GPU is to think that all the cores (typically 1,000's) that participate in the computation run the exact same instructions but on different data. A good example of a parallel algorithm that follows the pattern that we just described is applying the same task to many partitions of a SAS dataset. We will discuss a problem that require such an algorithm in more details below.

GPU, NVIDIA, AND CUDA

For this research project, we decided to concentrate on one vendor: NVIDIA. The main reason for selecting NVIDIA is not the fact that NVIDIA builds the most GPUs, but rather the fact that NVIDIA builds the faster GPUs. The supported language of NVIDIA GPU is the Compute Unified Device Architecture or CUDA. For more information on CUDA, you can go to NVIDIA's developer site: https://developer.nvidia.com/cuda-zone.

As you start coding for CUDA, you'll quickly realize that this is no picnic and exposing CUDA to SAS developers in general and IRM users in particular definitely would not meet the IRM requirement of delivering an easy to use development platform. Enter the SAS Function compiler or FCMP.

FCMP AND CUDA

FCMP is a procedural language that is readily available to SAS programmers. In particular, you can call FCMP code from a data step. For more information about FCMP, please refer to the SAS 9.4 documentation.

As we mentioned a couple of paragraphs ago, CUDA is the language that you typically use to program NVIDIA GPU. FCMP code will not run "as is" on a GPU device. So how do we bridge that gap?

The implementation of PROC FCMP generates C code with additional checks like array boundaries. So one possibility to bridge the gap would simply be to modify the PROC FCMP implementation to generate

CUDA code instead of C code. That approach would certainly work and might find an implementation in a future SAS release. However, we decided to focus on GPU accelerations for IRM tasks, which is much simpler than the general case of cross-compiling any FCMP code to CUDA. So we adopted an external cross-compiler that will be invoked manually.

So why is the IRM case much simpler than the general FCMP to CUDA conversion? The main reason is the fact that in IRM, one must declare the inputs and outputs in an IRM task, so the argument list for the CUDA code is readily available and there are no global variables or side effects to deal with. To keep the cross-compiler as simple as possible we also adopted a few additional constraints:

1. Only consider doubles in argument list. In other words, don't support characters (and formats). This constraint is acceptable given the nature of GPU devices: they can manipulate characters, but floating point operation acceleration is really what we are after.
2. Convert SAS datasets to arrays of doubles. In other words, the problem has to fit in memory. If you consider that partitions of SAS dataset is what we're using as inputs and outputs, only the partition has to fit in memory and one can create as many partitions as necessary.

Those constraints didn't prevent us from our initial goal: research how much GPU devices can accelerate SAS code in general and IRM tasks in particular.

So let's see how this works in practice. Consider the following FCMP code fragment:

```
subroutine income_statement_kernel(offset,index_scen,n_rows_per_slice,
       n_scen_mort_cols,d_matrix_mort[30,4],
       n_scen_lapse_cols,d_matrix_lapse[30,4],
       n_pol_rows,n_pol_cols,d_matrix_pol[32940,20],
       n_incsta_cols,d_matrix_is[30,15]);
      outargs d_matrix_is;

      /* Create a temporary array to hold aggregated income statement items and initialize
         it to all 0 */
      array is_temp[30,4];
      do ndx_year=1 to 30;
         is_temp[ndx_year,1] = 0;
         is_temp[ndx_year,2] = 0;
         is_temp[ndx_year,3] = 0;
         is_temp[ndx_year,4] = 0;
      end;
…
```

To get the CUDA equivalent, you would run the cross-compiler from the command line:

```
$ java risk.irm.server.ccross.FcmpToC datalib.funcs.csv -cuda income_statement_kernel
FCMP -> C: Entering...
  Cross-Compiling the following functions from datalib.funcs.csv:
    income_statement_kernel
  -- Output is in .cu file
FCMP -> C: All done!
```

And now you have the CUDA code that can be compiled using the NVIDIA ncc compiler and produce a shared object on LINUX or a DLL on Windows:

```
__device__ void income_statement_kernel( double offset,double index_scen,double
n_rows_per_slice,double n_scen_mort_cols,double* _irm_d_matrix_mort,double
n_scen_lapse_cols,double* _irm_d_matrix_lapse,double n_pol_rows,double n_pol_cols,double*
_irm_d_matrix_pol,double n_incsta_cols,double* _irm_d_matrix_is){
double (* d_matrix_mort)[(int)4] = (double (*) [(int)4])_irm_d_matrix_mort;
double (* d_matrix_lapse)[(int)4] = (double (*) [(int)4])_irm_d_matrix_lapse;
double (* d_matrix_pol)[(int)20] = (double (*) [(int)20])_irm_d_matrix_pol;
double (* d_matrix_is)[(int)15] = (double (*) [(int)15])_irm_d_matrix_is;
// subroutine income_statement_kernel(offset,index_scen,n_rows_per_slice,
n_scen_mort_cols,d_matrix_mort[30,4],n_scen_lapse_cols,d_matrix_lapse[30,4],
n_pol_rows,n_pol_cols,d_matrix_pol[32940,20],n_incsta_cols,d_matrix_is[30,15]);
// outargs d_matrix_is;
// array is_temp[30,4];
double is_temp[30][4];
// do ndx_year=1 to 30;
```

```
int ndx_year;
for(ndx_year=1; ndx_year <= 30; ndx_year++) {

// is_temp[ndx_year,1] = 0;
is_temp[(int)(ndx_year - 1)][(int)(1 - 1)] = 0;
// is_temp[ndx_year,2] = 0;
is_temp[(int)(ndx_year - 1)][(int)(2 - 1)] = 0;
// is_temp[ndx_year,3] = 0;
is_temp[(int)(ndx_year - 1)][(int)(3 - 1)] = 0;
// is_temp[ndx_year,4] = 0;
is_temp[(int)(ndx_year - 1)][(int)(4 - 1)] = 0;
// end;
}
```

As you can see, beauty and readability of the code were not the essential goals.

We will revisit the shared objects and DLLs later. The `datalib.funcs.cs` argument might be surprising; since macro substitution can take place inside the FCMP code we read the internal DATALIB.FUNCS.SAS7BDAT that contains the FCMP statements after macro expansion. More specifically, the cross-compiler written in Java reads the CSV equivalent of the DATALIB.FUNCS.SAS7BDAT dataset.

Finally we should emphasize that FCMP and the cross-compiler is not per se required to accelerate your SAS programs with CUDA-compatible devices. We will revisit this statement a little later. But first let's have a look at PBR: the problem that prompted us to start this research.

PBR

Principle-Based Reserving (PBR), the new US life insurance regulation requiring stochastic calculation on both the asset and the liability sides of the balance sheet, demands intensive computation. It therefore poses big challenges to insurers' technology infrastructure. In this research, we use a stochastic term life insurance pricing model as an example to present an easy and scalable methodology to help insurers calculate thousands, possibly millions, of scenarios as fast as possible. The model calculates income statements for a portfolio of one thousand policies for one million scenarios, a total of one billion projections.

PBR TASK

The PBR task calls a SAS macro that can run the PROC FCMP code on the CPU and the cross-compiled CUDA code on the GPU:

```
%macro irm_lf_define_is_kernel_func(language=);
   %if &language = fcmp %then %do;
      %put running as FCMP...;
      proc fcmp outlib=datalib.funcs.is;
      subroutine income_statement_kernel(offset,index_scen,n_rows_per_slice,
...
      endsub;
      run;
   %end;
   %else %do;
      %put running as FCMP compiled C code...;
      proc proto package=datalib.funcs.cfcns;
         link '/home/sas/libpbrCUDAso.so' nounload;
         void irm_c_income_statement_kernel(double offset,double index_scen,
               double n_rows_per_slice, double n_scen_mort_cols,
               double* _irm_d_matrix_mort, double n_scen_lapse_cols,
               double* _irm_d_matrix_lapse, double n_pol_rows,double n_pol_cols,
               double* _irm_d_matrix_pol,double n_incsta_cols,double* _irm_d_matrix_is);
      run;

      proc fcmp inlib=datalib.funcs outlib=datalib.funcs.is;
         /* function to compute income statement kernel */
         subroutine income_statement_kernel(offset,index_scen,n_rows_per_slice,
               n_scen_mort_cols,_irm_d_slice_scen_mort[*,*],n_scen_lapse_cols,
```

```
                _irm_d_slice_scen_lapse[*,*],
                n_pol_rows,n_pol_cols,_irm_d_matrix_pol[*,*],n_incsta_cols,_irm_d_matrix_is[*,*]);
                outargs _irm_d_matrix_is;
                call irm_c_income_statement_kernel(offset,index_scen,n_rows_per_slice,
                       n_scen_mort_cols,_irm_d_slice_scen_mort,n_scen_lapse_cols,
                       _irm_d_slice_scen_lapse,
                       n_pol_rows,n_pol_cols,_irm_d_matrix_pol,n_incsta_cols,_irm_d_matrix_is);
          endsub;
     run;
   %end;
...
data _null_;
  set datalib.lapse_scenario (obs=&nbr_lapse_rows);

  if _n_ = 1 then do;
    call sas_allocate_table(2, &nbr_lapse_rows., &nbr_lapse_cols.);
  end;

  array s_l_row {*} _numeric_;
  call sas_add_row(2, s_l_row);
run;

data _null_;
  array istmp[&nbr_scen_rows,&nbr_incsta_cols] _temporary_;
  put 'Before sas_income_statement';
  call sas_income_statement(&gpu_card_num,&start_scen_num,&nbr_scen_rows,
                       &nbr_rows_in_scen_slice, &nbr_scen_cols,
                       &nbr_mortality_rows, &nbr_mortality_cols, &nbr_lapse_rows,
                       &nbr_lapse_cols,
                       &nbr_pol, &nbr_pol_cols, &nbr_incsta_cols, istmp);
  put 'After sas_income_statement';

  call sas_free_table(1);
  call sas_free_table(2);
  call sas_free_table(0);
run;
```

The fact that the same code can run the FCMP code and the CUDA code is convenient: one can debug on the CPU with small data and FCMP and then once everything works fine, turn on the GPU by simply changing the value of the `language` variable.

The call to the PROC PROTO (`link '/home/sas/ libpbrCUDAso.so' nounload;`) is simply to make the cross-compiled code available to the FCMP code: we rely on FCMP to call C code.

Finally, the PBR task code contains several statements to load the data into arrays and then call the FCMP subroutine. This works inside an IRM task but would also work in plain old SAS MVA macro code if you wanted to try it without IRM.

For this research project, we were content with the manual steps of invoking the cross-compiler and then calling the FCMP code versus the CUDA code based on the value of a variable. We should point out that all the information is available to completely automate that process.

There is one more step left before we can run the CUDA code inside IRM: the PBR task must be part of a flow. So let's do that.

PBR FLOW

Figure 2 below shows the PBR flow. The details of the computations are outside of the scope of this paper, but a few things are worth pointing out. The circle labeled `result_income_statement_aggr` is in fact a SAS dataset that contains the income statement projections. The last blue box labeled `recombine_income_statement` is a task that takes two datasets as input and merge them into the final result. The real deal as far as GPUs are concerned is in the `calculate_income_statement_gpu` sub-flow (you can have a flow within a flow). That sub-flow is shown in Figure 3.

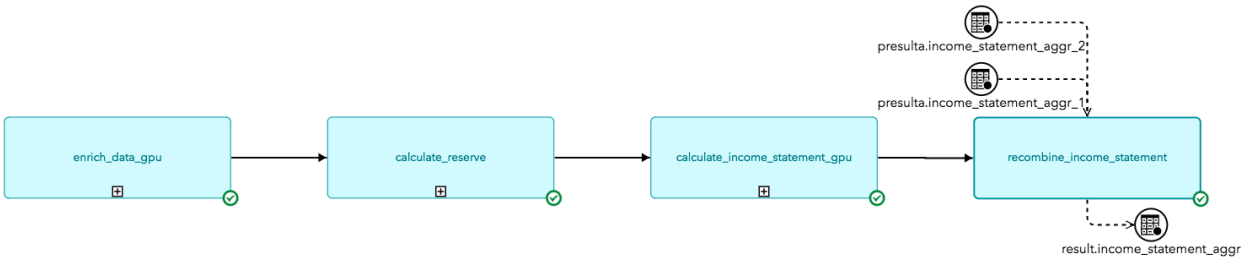

Figure 2. The PBR Flow

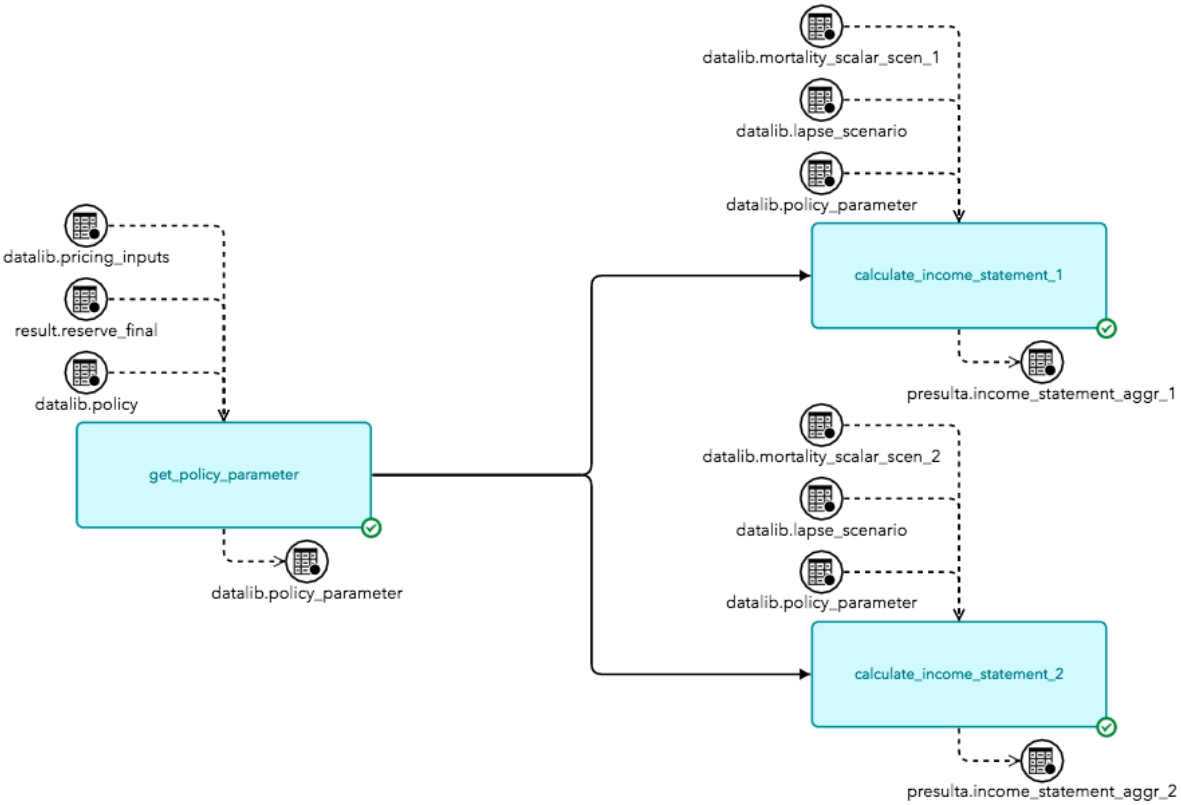

Figure 3. The GPU Flow

If you turn your attention to the `calculate_income_statement_[12]` tasks, you'll notice that they output the SAS datasets that we aggregated in Figure 2. Each of those tasks runs the PBR task on the GPU device. As you can see on the partial log below, we first load the prototype of the CUDA functions and then we run the FCMP code that loads the data, runs the computations on the GPU device, and then finally transfers the results back to SAS:

```
NOTE: The current task: task ID='_1067' ('calculate_income_statement_1') is running on host
fsnlax05 under process ID=146521 1476
%INCLUDE '/local/install/fa.life/source/sas/nodes/life/lf_income_statement_gpu1.sas';
NOTE: '/home/sas/libpbrCUDAso.so' loaded from specified path.
NOTE: Prototypes saved to WORK.PROTO_DS.CFCNS.
NOTE: PROCEDURE PROTO used (Total process time):
       real time           0.00 seconds
       cpu time            0.00 seconds

...

Before sas_income_statement
After sas_income_statement
```

```
NOTE: '/home/sas/libpbrCUDAso.so' loaded from specified path.
NOTE: DATA statement used (Total process time):
       real time           19.13 seconds
       cpu time            18.84 seconds
```

The 19 seconds or so take into account all operations (load, execute, save); only the execution portion is done on the GPU device.

PERFORMANCE RESULTS

Table 1 shows the performance data that we collected for different runs with and without GPU.

Technology		Calculation Time in Seconds
CPU	FCMP	763
GPU	FCMP -> CUDA	73
	CUDA	11

Table 1. CPU and GPU Performance.

As you can see on the second row, using the cross-compiled FCMP code gave us more than one order of magnitude of calculation times improvement. The last row represents the performance of a CUDA routine written from scratch: it performs the same calculations, but the CUDA code takes advantage of the GPU architecture. In this case of pure CUDA code, FCMP doesn't enter the picture at all. As the cross-compilation from FCMP to CUDA improves, we expect the 73 seconds to move closer to the 11 seconds.

CONCLUSION

In this article we have shown that by relying on the development methodology supported by IRM, namely microservices and Many Task Computing, it was possible to leverage the power of GPUs and get an order of magnitude of improvements in SAS code.

The research presented required several manual steps, but those can potentially be automated in future work.

Finally, the approach presented here of leveraging a GPU as an accelerator is not specific to GPUs. It could be used with a grid like the SAS Cloud Analytical Services (CAS) or other accelerator technologies like Intel's Xeon Phi.

REFERENCES

Fowler, Martin. "Microservices." Available at http://www.martinfowler.com/articles/microservices.html.

Raicu, Ioan and Foster, Ian and Yong, Zhao. 2008. "Many-task computing for grids and supercomputers." 2008 Workshop on Many-Task Computing on Grids and Supercomputers. Available at http://ieeexplore.ieee.org/document/4777912/.

Foster, Ian (1995). *"Designing and Building Parallel Programs"*. Addison–Wesley (ISBN 9780201575941)

The FCMP Procedure available at https://support.sas.com/documentation/onlinedoc/base/91/fcmp.pdf.