GPU Accelerated Prognostics

George E. Gorospe Jr.[1], Matthew J. Daigle[2], Shankar Sankararaman[3], Chetan S. Kulkarni[4] and Eley Ng[5]

[1,3,4] *SGT Inc., NASA Ames Research Center, Moffett Field,CA, 94035, USA*
george.e.gorospe@nasa.gov
shankar.sankararaman@nasa.gov
chetan.s.kulkarni@nasa.gov

[2] *NIO, San Jose, CA, 95134, USA*
matthew.daigle@nio.io

[5] *Universities Space Research Association, 615 National Ave, Mountain View, CA 94043*
Eleyngmedia@gmail.com

Abstract

Prognostic methods enable operators and maintainers to predict the future performance for critical systems. However, these methods can be computationally expensive and may need to be performed each time new information about the system becomes available. In light of these computational requirements, we have investigated the application of graphics processing units (GPUs) as a computational platform for real-time prognostics. Recent advances in GPU technology have reduced cost and increased the computational capability of these highly parallel processing units, making them more attractive for the deployment of prognostic software. We present a survey of model-based prognostic algorithms with considerations for leveraging the parallel architecture of the GPU and a case study of GPU-accelerated battery prognostics with computational performance results.

1. Introduction

Prognostic methods are valuable to system operators and maintenance personnel for the prediction of future performance, remaining useful life (RUL), and the probability that the system can successfully complete the intended work cycle. However, the application of these methods requires the input of information about the system, data processing, and the repeated execution of computationally expensive prognostics algorithms for the return of prognostic results. To consider the uncertainty in the predictions, e.g., in future system usage, many algorithms typically consider many potential system execution samples in order to compute the statistics of the RUL distribution. Such approaches do not scale well in typical centralized computing architectures.

To satisfy these requirements, we have investigated the application of graphics processing units (GPUs) as a computational platform for general and real-time prognostics. The computations of the prediction algorithms are divided into a set of several parallel computations, after which results are aggregated. This paper presents our preliminary work in this area, including considerations and constraints for the application of GPUs for prognostic algorithms, and an example of GPU-accelerated prognostics for battery end of discharge (EOD) prediction.

Initial work involving GPUs for general purpose computing required the phrasing of computational problems in the form of graphics language calculations (Owens et al., 2007). As GPU technology advanced, Nvidia released the Compute Unified Device Architecture (CUDA) which provided developers with an application programming interface to the computational units of the GPU. With access to the full computational power of the GPU, software developers were able to instantiate their code on hundreds of cores running thousands of threads for massively parallel programming (Luebke, 2008). Recently, GPUs have seen extensive use in the training of neural networks, virtual reality and augmented reality, and advanced image and data processing methods.

Although no written publication of the application of GPUs towards model-based prognostics could be found, there are many cases in which the utilization of GPUs has benefitted computationally expensive model-based predictive algorithms. Researchers from the National Center for Atmospheric Research achieved an order of magnitude increase in performance by using GPUs for a performance critical module of their predictive state-of-the-art non-hydrostatic NWP weather research and forecast model (Michalakes & Vachharajani, 2008). Similarly, researchers from the Centre for Wireless Network Design utilized GPUs in the calculation of very precise radio coverage predictions based on a very computationally intensive Finite-Difference Time-Domain (FDTD) mode (Valcarce, De La Roche, & Zhang, 2008). The use of GPUs for computing uncertainty in computational mechanics was studied by researchers from the University of Minnesota, who presented 5 case studies featuring GPU implementations common uncertainty quantification techniques, in each case GPU accelerated computation showed considerable advantages (Wojtkiewicz et al., 2011).

As a case study, this paper presents initial work in the implementation of model-based battery prognostics algorithms on a GPU. Section II describes the computational performance characteristics of GPUs and considerations for the implementation of algorithms on GPUs. Section III presents potential advantages in the use of GPUs for real-time prognostic algorithm processing. Section IV is a case study and initial results from an implementation of model based battery prognostics algorithm on a GPU. Conclusions and future work are described in the final section.

2. CONSIDERATIONS FOR GPU USE

The massively parallel nature of the GPU and the accessible and well-documented CUDA API enables developers to quickly and easily decide which portions of their code should be processed by the CPU and which should be processed by the GPU. However, not all code or algorithms receive a performance boost when implemented on a GPU. Factors such as memory access, direct human interaction, or a serial nature can reduce the potential for GPU-based acceleration.

The single-instruction multiple-thread execution model employed by the GPU allows individual threads to operate simultaneously even on different data, which leads to improvements in performance but requires that the instruction for each thread or task on the processor be identical. Additionally, the parallel processing of tasks on the GPU means that threads should not be reliant on the results of other threads. Finally, individual threads receive a unique index or thread ID and block ID. This can be used to assign each thread a different part of the supplied data or to write results to an array at the completion of the thread.

Additionally, the GPU processing flow should be followed to supply the threads with data to process and available memory to write the result. The GPU processing flow relies on the allocation of memory on the GPU prior to execution of code on the GPU. This means that GPU memory must be allocated for both the data to be copied to the GPU and the resulting data before processing can begin. After allocation, data can be copied from the CPU memory to the GPU memory. Once

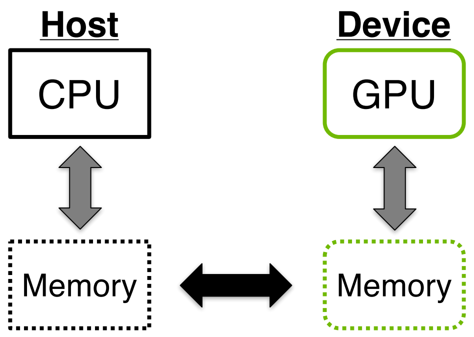


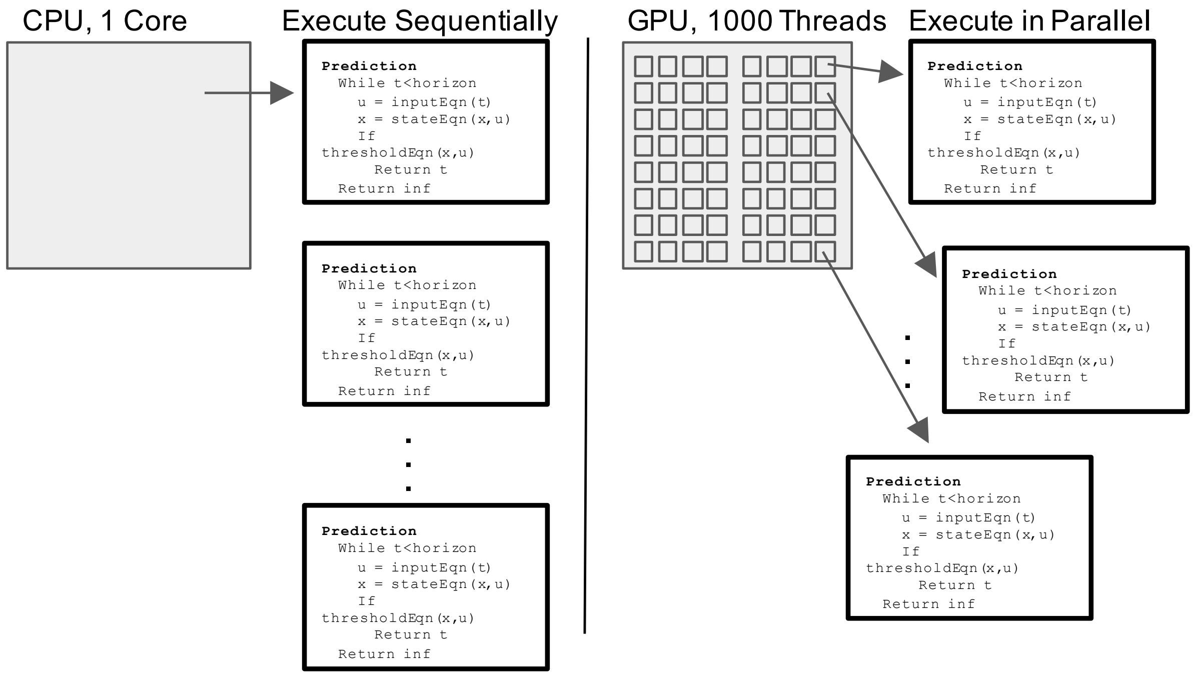
Figure 1. Processing flow for data transfer between memory on CPU and GPU

execution of the GPU code is complete, the data in the results array can be copied to the CPU memory from the GPU memory. GPU memory can be freed once this is done. Following this processing flow requires knowledge of the data to be copied and the result. Additionally, the overhead incurred by the copying or repeated copying of data between the CPU memory and the GPU memory can significantly reduce the benefit of parallelization.

3. PROGNOSTIC ALGORITHM ACCELERATION WITH THE USE OF GPUS

In this paper, we focus on the application of the model-based prognostics paradigm (Orchard & Vachtsevanos, 2009; Daigle & Goebel, 2013; Saha & Goebel, 2009), where prognosis is performed using a combination of a state estimation algorithm (often a Bayesian filter) and a prediction algorithm, both of which rely on a model of the monitored system. Prognostics approaches using machine learning can already be readily applied to GPUs using available software such as TensorFlow. Generally, within the model-based prognostics architecture, we must first perform state estimation, then a prediction up to a time horizon or until a threshold of interest is reached. For each of these steps, we use the system model. As new data is received, the process begins again.

For the state estimation part, the unscented Kalman filter (UKF) (Julier & Uhlmann, 2004) and the particle filter (PF) (Arulampalam, Maskell, Gordon, & Clapp, 2002; Doucet, Godsill, & Andrieu, 2000) are commonly applied within the context of prognostics. Both algorithms use a set of samples in order to estimate the system state. Thus, the computations that are performed on each sample individually can be readily parallelized. However, state estimation occurs every time new data is obtained, and the computations performed on each sample is minimal. Thus, state estimation algorithms are generally not good candidates for GPU implementation, since the overhead of copying memory to and from the CPU and GPU overcomes any benefit achieved by parallelization. In distributed prognostics approaches us-



```
Prediction
  While t<horizon
    u = inputEqn(t)
    x = stateEqn(x,u)
    If
thresholdEqn(x,u)
      Return t
  Return inf
```

```
Prediction
  While t<horizon
    u = inputEqn(t)
    x = stateEqn(x,u)
    If
thresholdEqn(x,u)
      Return t
  Return inf
```

```
Prediction
  While t<horizon
    u = inputEqn(t)
    x = stateEqn(x,u)
    If
thresholdEqn(x,u)
      Return t
  Return inf
```

```
Prediction
  While t<horizon
    u = inputEqn(t)
    x = stateEqn(x,u)
    If
thresholdEqn(x,u)
      Return t
  Return inf
```

```
Prediction
  While t<horizon
    u = inputEqn(t)
    x = stateEqn(x,u)
    If
thresholdEqn(x,u)
      Return t
  Return inf
```

```
Prediction
  While t<horizon
    u = inputEqn(t)
    x = stateEqn(x,u)
    If
thresholdEqn(x,u)
      Return t
  Return inf
```

Figure 2. Sequentially executed prediction steps under the standard model-based prognostics architecture

ing multiple filters (Daigle, Bregon, & Roychoudhury, 2012, 2014), each filter could be potentially parallelized into its own GPU thread, with results being aggregated on the CPU.

The prediction of the future state up to a certain time horizon within the our model-based prognostics architecture relies the commonly used Monte Carlo uncertainty propagation method to compute independent realizations from randomly drawn samples (Sankararaman, Daigle, & Goebel, 2014). For a single realization of the system state at the time of prediction and the future input trajectory, a single realization of the future system trajectory is predicted along with the corresponding RUL. Each of these realizations is produced by executing a simulation of the system with the given inputs using the system model. Each simulation can take a significant amount of time, depending on the time horizon and the number of the samples in the simulation. Increasing the number of samples in the simulation produces a more accurate probability density function but also increases the computational overhead. So, the prediction algorithm is a perfect candidate for parallelization on a GPU. Each simulation can be performed in parallel, rather than sequentially as on a CPU.

4. Case Study: GPU Accelerated Battery Prognostics

For our initial investigation into GPU programming for prognostics, we wanted to choose a prognostics algorithm which could benefit from parallelization. Our most recent electrochemistry model-based battery end of discharge algorithm (Daigle & Kulkarni, 2013) was chosen based on its complexity and computational overhead. This algorithm, which also employs the state estimation, then prediction steps discussed above, simulates the discharge of a single 18650 lithium-ion cell. The cell starts at a nominal voltage of 4.2 volts at 100% state of charge, and through a constant discharge, we are able to track a decrease in voltage over time.

A CPU-based implementation of the battery EOD algorithm, as used within the General Software Architecture for Prognostics was profiled to determine the runtime of the full algorithm including an Unscented Kalman filter algorithm for state estimation and a Monte Carlo algorithm for the prediction step. Results from the profiling at 1000 samples with a time horizon of 5,000 seconds, show that 99.93% of the runtime for the algorithm sources from the prediction step of the algorithm; hence, we focus on implementing the prediction step of the algorithm on the GPU.

To demonstrate the performance improvement of the Monte Carlo implementation on the GPU, two Monte Carlo estimation of Pi programs were created, a CPU-based version and a GPU accelerated version. Each program featured 268,435,456 samples and each program was compared using a laboratory workstation operating Ubuntu Linux with an Intel i5-5600 (Quad Core 3.2Ghz) COU and Nvidia Quadro K1200 GPU. The CPU version ran approximately 88 times slower than its respective GPU implementation with the same root-mean-square error of 0.11

Here, we focus only on the implementation of prediction step of the EOD algorithm on the GPU following the approach outlined in the previous section. To produce an estimation of the time until EOD, a Monte Carlo simulation of the battery state is performed. With a time step of 1 second the battery state is updated and checked against the low voltage threshold until a time horizon of 5,000 seconds or the end of discharge is reached. The implementation of this algorithm on the GPU required the creation of a kernel with instruction for the threads and state estimation and threshold checking functions to be compiled for the GPU in support of the kernel. Additionally, main code operating on the CPU was used to allocate memory, measure kernel compilation time, synchronize thread processing, and perform the final calculation of the EOD using the results of each thread. This code was developed in such a way that the number of samples in the simulation could be manually increased or decreased before the simulation.

Pseudo code of the device code (kernel) and the host code (main CPU program) can be found in the following subsections.

4.1. Host Code

The host code operates on the CPU and is the main body of the application. Within the host code, parameters for the simulation are defined, memory is allocated on the CPU memory and the GPU memory, and a GPU accelerated Mersenne Twister pseudo random number generator is quickly implemented to create random samples for the input of each Monte Carlo realization. Within the host code the GPU kernel is called specifying both the number of blocks to use and the number of threads per block. Additionally, pointers to the random number array and an empty results array are passed to the kernel during the call. While the GPU executes the kernel, each operation of the kernel operates on each thread of each block specified. During this time the CPU is free to continue executing the host code. In this case, we synchronize before continuing to ensure that all threads have completed before processing their results.

For this case study, the future battery loads are assumed to be constant, and these loads are sampled for the Monte Carlo prediction. This could easily be extended to also sample also the initial state at the time of prediction (i.e., from the distribution computed by the estimation algorithms).

Once all the threads have run, a memory copy operation makes the results available on the CPU memory. The host code aggregates the results and calculates the probability of end of discharge state.

```
# Host code, operates on CPU
int main(int argc, char* argv)

// Number of samples in the sim.
int threads = 100; int blocks = 100
int numSamples = blocks*threads;

// Allocation of memory for model input
// on the host and on the device
double *U_d;
cudaMalloc((void**)&U_d, numSamples *
sizeof(double));

// Allocation of memory for the EOL
// results array on the device
double *EOL_d;
cudaMalloc((void**)&EOL_d, numSamples *
sizeof(double));

// Allocate memory on CPU for results that
// will be copied from device
double *EOL = (double*)malloc(numSamples *
sizeof(double));

// Use CuRand to generate an array of
// random numbers on the device using
// GPU accelerated Mersenne Twister - PRNG
curandGenerator_t gen;
curandCreateGenerator(&gen,
        CURAND_RNG_PSEUDO_MRG32K3A);
curandSetPseudoRandomGeneratorSeed(gen,
        4294967296ULL^time(NULL));
curandGenerateUniformDouble(gen, U_d, numSamples);
curandDestroyGenerator(gen);

// Call the kernel function, specifying
// number of blocks and threads
kernel <<<blocks, threads>>> (U_d, EOL_d);

// Sychronize all threads, to make sure all
// results are in
cudaDeviceSynchronize();

// Copy results from GPU to CPU
cudaMemcpy(EOL, EOL_d, numSamples *
sizeof(double), cudaMemcpyDeviceToHost);

// Compute aggregate EOL results
// Go through all samples, compute average
// EOL and probability of EOL
double sumEOL = 0;
double numberOfEOL = 0;

// Go through all samples
for (int i=0; i<numSamples; i++) {
 // If EOL was reached, add to
 // the sum and count
        if (EOL[i] >=0) {
         //printf("EOL:
         sumEOL += EOL[i];
         numberOfEOL++;
        }}

// Compute average EOL (of those that were reached)
double averageEOL = sumEOL/numberOfEOL;
// Compute probability of EOL
double probabilityOfEOL = numberOfEOL/numSamples;

// Clean up
cudaFree(U_d); cudaFree(EOL_d); free(EOL);
return 0;
}
```

4.2. Device Code

The device code operates on the GPU in parallel. Once the GPU kernel is called by the main program, this code is executed in parallel by the threads within the blocks. The kernel function operates on a single sample, i.e., it runs a single realization of the future system evolution. In this case, it simulates up to a finite time horizon of 5,000 seconds. It saves whether this particular thread hit end of discharge, and if so, at what time.

```
__global__ void kernel()
{
// Set up index for each thread
// based on thread and block structure
int tid;
tid =blockDim.x*blockIdx.x + threadIdx.x;

// Set up parameters for the model
// Set up process noise if desired
// Set to initial state, 4.2 volts
// Get the model input
// Set default EOD value, 3 volts

// Simulate to a finite time horizon,
// with a sample time of 1
double tHorizon = 5000;
double dt = 1;

for (double t=0; t<=tHorizon; t+=dt) {
        // Simulate the model one step ahead
        stateEqn(param, t, x, u, Noise, dt);

        // Check if EOD has occurred
        bool atEOD = threshold(param, t, x, u);
  if (atEOD) {
                // Save EOD and exit kernel
    EOD_d[tid] = t;
    Return;}
}}
```

4.3. Baseline Results

To evaluate the performance benefits GPU implementation of prognostics algorithms, we used the Monte Carlo-based based battery end of discharge algorithm as described in the previous section. This algorithm was implemented using CUDA, with the CuRAND library and Mersenne Twister pseudo-random number generator, and executed on a Nvidia Jetson TX1 development kit operating Ubuntu Linux 14.04. These tests were compared with the same algorithm implemented in standard C++, on an Intel i7 powered, Apple Mac Book Pro without the assistance of a GPU. The following results are for multiple timed executions of the algorithm on each system with an increasing number of samples in the Monte Carlo simulation.

The GPU-based implementation provides a significant advantage, and scales much better than the purely CPU-based implementation. With a small number of samples, the overhead in copying memory between the CPU/GPU dominates, and the CPU-only approach is preferred. For larger numbers of samples (which is the case in practice), the GPU approach provides an order of magnitude improvement.

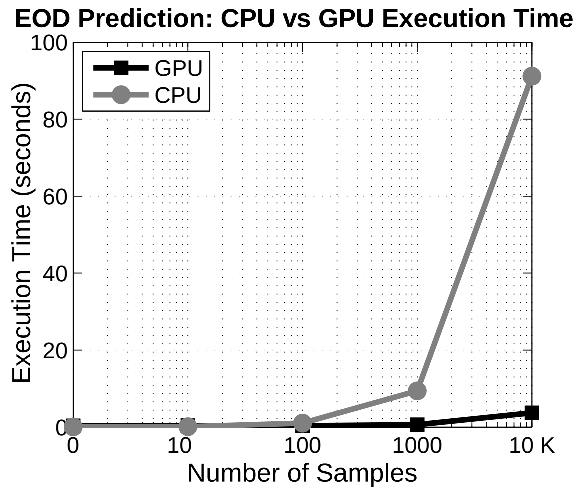


Figure 3. Initial results comparing CPU vs CPU processing time

N	CPU Execution Time	GPU Execution Time
1	**0.005**	0.313
10	**0.11**	0.365
100	1.001	**0.349**
1000	9.40	**0.619**
10K	91.194	**3.709**

Table 1. Baseline results of battery prognostics algorithm execution time

5. CONCLUSIONS

In this paper, we described how GPUs can be used to accelerate prognostics, specifically within the model-based prognostics approach. We showed that prediction algorithms can be easily parallelized and implemented on a GPU resulting in reduced execution time. The results demonstrate the improved performance of GPU-based approaches and open the door for increased accuracy through utilization of greater numbers of samples. Future work will involve exploring how GPUs can be used to improve prognostics-integrated decision-making algorithms and how they may enable a prognostics as a service paradigm.

REFERENCES

Arulampalam, M. S., Maskell, S., Gordon, N., & Clapp, T. (2002). A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking. *IEEE Transactions on Signal Processing*, *50*(2), 174–188.

Daigle, M., Bregon, A., & Roychoudhury, I. (2012, September). A distributed approach to system-level prognostics. In *Annual conference of the prognostics and health management society 2012* (p. 71-82).

Daigle, M., Bregon, A., & Roychoudhury, I. (2014, June). Distributed prognostics based on structural model decomposition. *IEEE Transactions on Reliability*, *63*(2), 495-510.

Daigle, M., & Goebel, K. (2013, May). Model-based prognostics with concurrent damage progression processes. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, *43*(4), 535-546.

Daigle, M., & Kulkarni, C. (2013, October). Electrochemistry-based battery modeling for prognostics. In *Annual conference of the prognostics and health management society 2013* (p. 249-261).

Doucet, A., Godsill, S., & Andrieu, C. (2000). On sequential Monte Carlo sampling methods for Bayesian filtering. *Statistics and Computing*, *10*, 197–208.

Julier, S. J., & Uhlmann, J. K. (2004, March). Unscented filtering and nonlinear estimation. *Proceedings of the IEEE*, *92*(3), 401–422.

Luebke, D. (2008). Cuda: Scalable parallel programming for high-performance scientific computing. In *Biomedical imaging: From nano to macro, 2008. isbi 2008. 5th ieee international symposium on* (pp. 836–838).

Michalakes, J., & Vachharajani, M. (2008). Gpu acceleration of numerical weather prediction. *Parallel Processing Letters*, *18*(04), 531–548.

Orchard, M., & Vachtsevanos, G. (2009, June). A particle filtering approach for on-line fault diagnosis and failure prognosis. *Transactions of the Institute of Measurement and Control*, *31*(3-4), 221-246.

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., & Purcell, T. J. (2007). A survey of general-purpose computation on graphics hardware. In *Computer graphics forum* (Vol. 26, pp. 80–113).

Saha, B., & Goebel, K. (2009, September). Modeling Li-ion battery capacity depletion in a particle filtering framework. In *Proceedings of the annual conference of the prognostics and health management society 2009.*

Sankararaman, S., Daigle, M., & Goebel, K. (2014, June). Uncertainty quantification in remaining useful life prediction using first-order reliability methods. *IEEE Transactions on Reliability*, *63*(2), 603-619.

Valcarce, A., De La Roche, G., & Zhang, J. (2008). A gpu approach to fdtd for radio coverage prediction. In *Communication systems, 2008. iccs 2008. 11th ieee singapore international conference on* (pp. 1585–1590).

Wojtkiewicz, S. F., et al. (2011). Use of gpu computing for uncertainty quantification in computational mechanics: A case study. *Scientific Programming*, *19*(4), 199–212.