Leverage GPU Acceleration for Your Program on Apache Spark

GTC 2017

Kazuaki Ishizaki +, Madhusudanan Kandasamy *, Gita Koblents -

+ IBM Research – Tokyo
* IBM India
- IBM Canada
Spark is Becoming Popular for Parallel Computing

- Write a Scala/Java/Python program using parallel functions with distributed in-memory data structures on a cluster
  - Can call APIs in domain specific libraries (e.g. machine learning)

```scala
val dataset = ...((x1, y1), (x2, y2), ...) // input points
val model = KMeans.fit(dataset)           // train k-means model
...
val vecs = model.clusterCenters.map(vec => (vec(0)*2, vec(1)*2)) // x2 to all centers
```

Leverage GPU Acceleration for your Program on Apache Spark
Spark is Becoming a Friend of GPUs

Leverage GPU Acceleration for your Program on Apache Spark

GPU Acceleration on Apache Spark™

Authors:
Machuudanan Kandasamy and Josiah Samuel

Bringing Apache Spark™ Closer to SIMD and GPU

GPU Acceleration on Databricks

Speeding Up Deep Learning on Apache Spark

by Joseph Bradley, Tim Hunter and Yandong Mao

Posted in ENGINEERING BLOG | October 27, 2016

Databricks is adding support for Apache Spark clusters with Graphics Processing Units (GPUs), ready to accelerate Deep Learning workloads (read press release). With Spark deployments tuned for GPUs, plus pre-installed libraries and examples, Databricks offers a simple way to leverage GPUs to power image processing, text analysis, and other Machine Learning tasks. Users will benefit from 10x speedups in Deep Learning, automated configuration of GPU machines, and smooth integration with Spark clusters. The feature is available by request, and will be generally available within weeks.
What You Will Learn from This Talk (1/2)

- How to easily accelerate your code using GPUs on a cluster
  - Hand-tuned GPU program in CUDA
    ```scala
    _global_ void yourGPUKernal(double *in, double *out, long size) {
        long i = threadIdx.x + blockIdx.x * blockDim.x;
        out[i] = in[i] * PI;
    }
    ```
    ```scala
    val mapFunction = new CUDAFunction(..., "yourGPUKernel.ptx")
    val output = data.mapExtFunc(..., mapFunction)
    ```

- Spark program with automatic translation to GPU code
  ```scala
  val output = data.map(p => Point(p.x * 2, p.y * 2))
  ```
What You Will Learn from This Talk (2/2)

- How to easily accelerate your code using GPUs on a cluster
  - Hand-tuned GPU program in CUDA
  - Spark program

- Achieve good performance results using one P100 card over 160-CPU-thread parallel execution on POWER8
  - 3.6x for CUDA-based mini-batch logistic regression
  - 1.7x for Spark vector multiplication

- Address ease of programming for non-experts, not address the state-of-the-art performance by Ninja programmers
Comparison of Two Approaches

- Non-expert programmers can use GPU without writing GPU code

<table>
<thead>
<tr>
<th>Use case</th>
<th>GPU program</th>
<th>Spark program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case</td>
<td>Prepare highly-optimized algorithms for GPU in domain specific library (e.g. MLlib)</td>
<td>Write more generic code in an application</td>
</tr>
<tr>
<td>GPU code</td>
<td>Hand-tuned by programmer</td>
<td>Automatically generated</td>
</tr>
<tr>
<td>How to write GPU code</td>
<td>CUDA</td>
<td>Spark code (Scala/Java)</td>
</tr>
<tr>
<td>Spark Enhancement</td>
<td>Plug-in</td>
<td>Changing Spark and Java compiler</td>
</tr>
<tr>
<td>GPU memory management, data copy between CPU and GPU, data conversion between Spark and GPU</td>
<td>Automatically performed</td>
<td>Automatically performed</td>
</tr>
</tbody>
</table>
Outline

▪ Goal

▪ Motivation

▪ How to Execute Your GPU Program on Spark

▪ How to Execute Your Spark Program on GPU

▪ Performance Evaluation

▪ Conclusion
Why We Want to Use Spark for Parallel Programming

- High productivity
  - Ease of writing a parallel programming on a cluster
  - At Scale
    - Write once, run any cluster
    - Rich set of domain specific libraries

- Computation-intensive applications in non-HPC area
  - Data analytics (e.g. The Weather Company)
  - Log analysis (e.g. Cable TV company)
  - Natural language processing (e.g. Real-time Sentiment Analysis)
Programmability of CUDA vs. Spark on a node

- CUDA requires programmers to explicitly write operations for
  - managing device memories
  - copying data between CPU and GPU
  - expressing parallelism

- Spark enables programmers to just focus on
  - expressing parallelism

```java
val datasetA = ...
val datasetB = datasetA.map(e => e * 2.0)
```

```c
void fooCUDA(N, float *A, float *B, int N) {
    int sizeN = N * sizeof(float);
    cudaMalloc(&d_A, sizeN); cudaMalloc(&d_B, sizeN);
    cudaMemcpy(d_A, A, sizeN, HostToDevice);
    GPU<<<N, 1>>>(d_A, d_B, N);
    cudaMemcpy(B, d_B, sizeN, DeviceToHost);
    cudaFree(d_B); cudaFree(d_A);
}
```

```c
__global__ void GPU(float* d_a, float* d_b, int n) {
    int i = threadIdx.x;
    if (n <= i) return;
    d_b[i] = d_a[i] * 2.0;
}
```
Outline

- Goal
- Motivation
- How to Execute Your GPU Program on Spark
- How to Execute Your Spark Program on GPU
- Performance Evaluation
- Conclusion
Hand-tuned your GPU Program in a Nutshell

▪ This is available at https://github.com/IBMSparkGPU/GPUEnabler
  – Blog entry: http://spark.tc/gpu-acceleration-on-apache-spark-2/

▪ It is implemented as Spark package
  – Can be drop-in into your version of Apache Spark

▪ The Spark package accepts PTX (an assembly language file that can be generated by a CUDA file) as GPU program
  – Convert data between Spark and GPU, manage GPU memory, and copy data between GPU and CPU

▪ The Spark package launches GPU program from map() or reduce() parallel function
How to Write and Execute Your GPU Program

1. Write a GPU program and create a PTX

   ```java
   __global__ void multiplyBy2(int *inx, int *iny, int *outx, int *outy, long size) {
   long i = threadIdx.x + blockIdx.x * blockDim.x;
   if (size <= i) return;
   outx[i] = inx[i] * 2; outy[i] = iny[i] * 2;
   }
   
   $\text{nvc} \text{c example.cu -ptx}$
   
2. Write a Spark program

   ```scala
   case class Point(x: Int, y: Int)
   Object SparkExample {
     val mapFunction = new CUDAFunction(
       "multiplyBy2", Array("this.x", "this.y"), Array("this.x", "this.y"), "example.ptx"
     )
     val output = sc.parallelize(1 to 65536, 24).map(e => Point(e, -e))
       .cache
       .mapExtFunc(p => Point(p.x*2, p.y*2), mapFunction).show }
   
3. Compile and submit them

   $$\text{mvn package}$$
   $$\text{bin/spark-submit --class SparkExample SparkExample.jar}$$
   $$\text{--packages com.ibm:gpu-enabler_2.11:1.0.0}$$
   
   Leverage GPU Acceleration for your Program on Apache Spark
How Your GPU Program is Executed

- Optimize data layout for GPU
  - Columnar oriented layout

- Copy data between CPU and GPU

- Exploit parallelism
  - among GPU kernels
  - among CUDA cores
Outline

- Goal
- Motivation
- How to Execute Your GPU Program on Spark
- How to Execute Your Spark Program on GPU
- Performance Evaluation
- Conclusion
Spark Program in a Nutshell

▪ This is on-going project
  – Blog entry: http://spark.tc/simd-and-gpu/

▪ We are enhancing Spark by modifying Spark source code
  – Also apply changes to Java Just-in-time compiler

▪ The enhanced Spark accepts an expression in map() for now

▪ The enhanced Spark handles low-level operations for GPU
  – Generate GPU code from Spark program
  – Convert data between Spark and GPU, manage GPU memory, and copy data between GPU and CPU
How Scala Code is Executed

- Already optimized data layout for GPU
  - Modified Spark to use columnar oriented layout

- Generate GPU code from Scala code

- Copy data between CPU and GPU

- Exploit parallelism
  - among kernels
  - among CUDA cores

```scala
... 
.map(p => Point(p.x*2, p.y*2))
...

__global__ void multiplyBy2(…) {
  …
  outx[i] = inx[i] * 2;
  outy[i] = iny[i] * 2;
}
...`
Translation of Spark Program

- Generate GPU code from **an expression**
- Allocate/deallocate GPU memory and **copy data between GPU and CPU**

Spark program

```scala
dataset2 = dataset1.map(p => Point(p.x*2, p.y*2))
...
```

Generated CPU code

```java
Column colinx = dataset1.getColumn(0); // Point.x in dataset1
Column coliny = dataset1.getColumn(1); // Point.y in dataset1
Column coloutx = dataset2.getColumn(0); // point.x in dataset2
Column colouty = dataset2.getColumn(1); // point.y in dataset2
int nRows = column0.numRows; int nBytes = nRows * 4;
cudaMalloc(&d_colinx, nbytes); cudaMemcpy(d_colinx, &colinx.data, nbytes, H2D);
cudaMalloc(&d_coliny, nbytes); cudaMemcpy(d_coliny, &coliny.data, nbytes, H2D);
<<...>> GPU(d_colinx, d_coliny, nRows) // launch GPU
```
Outline

- Goal
- Motivation
- How to Execute Your GPU Program on Spark
- How to Execute Your Spark Program on GPU
- Performance Evaluations
- Conclusion
Performance Improvements of GPU Program over Parallel CPU

- Achieve 3.6x for CUDA-based mini-batch Logistic Regression using one P100 card over POWER8 160 SMT cores

IBM Power System S822LC for High Performance Computing “Minsky”, at 4 GHz with 512GB memory, one P100 card, Fedora 7.3, CUDA 8.0, IBM Java pxl640sr4fp2-20170322_01(SR4 FP2), 128GB heap, Apache Spark 2.0.1, master="local[160]", GPU Enabler as 2017/5/1, N=112000, features=8500, iterations=15, mini-batch size=10, parallelism(GPU)=8, parallelism(CPU)=320
Performance Improvements of Spark Program over Parallel CPU

- Achieve 1.7x for Spark vector multiplication using one P100 card over POWER8 160 SMT cores

![Relative execution time over GPU version](image)

IBM Power System S822LC for High Performance Computing “Minsky”, at 4 GHz with 512GB memory, one P100 card, Fedora 7.3, CUDA 8.0, IBM Java pxl6480sr4fp2-20170322_01(SR4 FP2), 128GB heap, Based on Apache Spark master (id:657cb9), master="local[160]", N=480, vector length=1600, parallelism(GPU)=8, parallelism(CPU)=320
Takeaway

- How to easily accelerate your code using GPUs on a cluster
  - Hand-tuned CUDA kernel
  - Spark program

- How Spark runtime executes a program on GPUs
  - No programmer’s work for low-level operations related to GPU
    - Data conversion, GPU memory management, data copy, kernel invocation, and program translation

- Achieve good performance results using one P100 card
  - 3.6x and 1.7x over 160-CPU-thread parallel execution on POWER8
    - Address ease of programming for many non-experts, not for the state-of-the-art performance by small numbers of Ninja programmers