A Support Vector Machine Implementation on NVIDIA Graphic Processing Units

Daniel Llamocca
Ingo Steinwart
Outline

- Objectives
- Basics of CUDA (Compute Unified Device Architecture) Programming Model
- Parallelization of the algorithms
- Issues and performance guidelines
- Results
- Conclusions
Objectives

- Expose and exploit parallelism of the following routines:
  - Support Vector Machine testing (classification)
  - Support Vector Machine training
  - Validation phase of SVM training
- GPU implementation of the parallel algorithms.
- Obtain significant speedup improvements.
**Basics of CUDA Programming Model**

- **CUDA**: Computing architecture that leverages the parallel computing engine on NVIDIA GPUs. It allows developers to use C language.
- Parallelism is described by a grid that consists of blocks, each block having a number of threads:
Basics of CUDA Programming Model

- Grid Size and Block Size limitation:
  NVIDIA GTX 295: 512 threads per block (1D, 2D, or 3D block). 65535 blocks per grid (1D or 2D grid)

- All threads in a block run in parallel. A limited number of blocks can be run in parallel. NVIDIA GTX 295: 30 SM (streaming multiprocessors) ➔ up to 30 blocks can execute concurrently, i.e. 30x512 = 15360 threads.
Basics of CUDA Programming Model

- Kernel: Code executed on the GPU:

  C Program Sequential Execution
  
  Serial Code

  Parallel Kernel 0

  Serial Code

  Parallel Kernel 1

  Host (CPU) ⇔ Device (GPU)

  Grid 0
  - Block(0,0)
  - Block(0,1)
  - Block(0,2)
  - Block(1,0)
  - Block(1,1)
  - Block(1,2)
  - Block(2,0)
  - Block(2,1)
  - Block(2,2)

  Grid 1
  - Block(0,0)
  - Block(0,1)
  - Block(1,0)
  - Block(1,1)
Basics of CUDA Programming Model

- **GPU Memory:** There are 4 basic kinds of memories:
  - **Global Memory:** Non-cached and high latency memory. It is available to all threads and is able to communicate with CPU memory.
  - **Shared Memory:** Faster than global memory. GTX295: 16384 bytes of shared-memory per thread block.
  - **Constant Memory, Texture Memory.**
  - **Streams:** Allow CPU to GPU communication to overlap with GPU kernel execution.
Parallelization of the algorithms

SVM testing

- The following function is to be computed:
  \[ f(x_i) = \sum_{k \in SV} y_k \alpha_k K(x_{t_k}, x_i), \forall i \in test\ dataset \]
  
  \( x_i \): test dataset sample, \( i = 1: N \)
  
  \( x_{t_k} \): train dataset sample, \( y_k \alpha_k \): train coefficients, \( k = 1: SV \)

- Parallelization: 2 Kernels:
- Compute individual summations: Each thread will compute SPT sums of products.
- Compute final function value: Each thread will add up the previous values to get the final function value.
Parallelization of the algorithms

SVM testing

- First Kernel: Each thread computes SPT sums of products. (SPT = 4, 8, 12, ...)

\[
\sum_{k=0}^{SPT-1} y_k \alpha_k K(x_t_k, x_i)
\]

- TBP : threads per block (256, 512)
- GY : test samples processed by a grid
- GX * GY ≤ 65535
- GX = \left\lceil \frac{SV}{TPB* SPT} \right\rceil, GY = \left\lfloor \frac{65535}{GX} \right\rfloor

- 1st kernel output: A vector of SV/SPT elements for every one of the GY samples. Vectors are stored in global memory
Parallelization of the algorithms
SVM testing

- Second Kernel: Each thread will add up the previous values to get the final function value for a test sample.

- $2^{nd}$ kernel output: A vector of GY function values

- Since $N$ test samples might be way larger than GY, the 2 kernels have to be run $NC$ times. $NC = \left\lfloor \frac{N}{GY} \right\rfloor$
Parallelization of the algorithms
SVM testing

- ‘frings.lanl.gov’ machine: It contains 2 NVIDIA GTX295 boards, each one with 2 GPUs.
- Therefore, we further parallelized the algorithm by running 4 CPU threads, each CPU thread run a GPU device code independently. The number of test samples is divided in 4 chunks, each chunk is processed by a GPU device.
Issues and Performance Guidelines

- GY, GX, NGY, NC involve floor and ceil operations. Take NC for instance: at the last run, the number of samples processed has to be smaller than GY, otherwise there will be wasted computation. This introduces more flow control that impacts the overall performance.

- Call to CPU thread that enables GPU operation: There is an overhead of 80 ms. Try to keep those calls as low as possible.

- Global GPU memory is not cached, thus this is the slowest memory. In our case, shared memory was deemed impractical and global memory was used.

- Communication between CPU memory and GPU global memory is the slowest process. Do it when really necessary, and transfer one big chunk instead of several small chunks.
## Results

- **C10-Oct14 Dataset: 11 decision functions**

<table>
<thead>
<tr>
<th>Support Vectors</th>
<th>Test size</th>
<th>815,000</th>
<th>50,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>4 CPU cores:</td>
<td>1 GPU:</td>
<td>36.8 s</td>
</tr>
<tr>
<td></td>
<td>274.6 s</td>
<td>2 GPUs:</td>
<td>20.3 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 GPUs:</td>
<td>15.3 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 GPUs:</td>
<td>13.2 s</td>
</tr>
<tr>
<td>2000</td>
<td>4 CPU cores:</td>
<td>1 GPU:</td>
<td>78.1 s</td>
</tr>
<tr>
<td></td>
<td>462.5 s</td>
<td>2 GPUs:</td>
<td>41.0 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 GPUs:</td>
<td>29.1 s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 GPUs:</td>
<td>23.6 s</td>
</tr>
</tbody>
</table>
Conclusions

- We achieve speed-up of about 20x in large datasets. In small datasets, the speed-up is about 6x. This suggests that the larger the datasets, the better the speed-up improvement.
- Consider that we are using double floating point arithmetic (64 bits). This requires twice the amount of registers in the GPU than in the case of single floating point arithmetic; if the registers do not fit in the register space, the compiler will place the ‘registers’ in the slow global memory.