Supporting Data Parallelism in Matcloud: Final Report

Yongpeng Zhang, Xing Wu

1 Overview

Matcloud is an on-line service to run Matlab-like script on client's web browser. Internally it is accelerated by CUDA-enabled GPUs to provide better performance. For calculations on large matrices, the performance margin is large enough to compensate for the network delays that do not exist on locally installed Matlab. Previous work has been focusing on one-to-one mapping between a script command and a CUDA kernel or library calls. However, there are cases when it is highly inefficient to do so. One example is *for* loops that work on fine-grained matrix element at each iteration. Mapping single iteration to a CUDA kernel is possible but it is against the massive data-parallel spirit of modern GPUs. Therefore it is necessary to aggregate iterations into one CUDA kernel. Moreover, the latest NVIDIA GPU (code name "fermi") supports concurrent kernel execution. It improves the utilization rate of GPU resource when single kernel is not in a large enough scale to fully exercise the all cores in GPU. This feature brings up an opportunity to accelerate operations on smaller matrices, which are common in Matcloud.

In this project, we use compiling techniques to optimize our Matcloud framework. Namely, our goal is to (1) build a just-in-compiler (JIT) framework to accelerate *parfor* loop and (2) propose *parsection* block (similar to parallel sections in OpenMP) in Matcloud to exploit the concurrent kernel execution feature in the latest generation GPUs. The detailed implementation is described in Section 2. In Section 3, we present some experimental results. The task description and each person's contribution are shown in Section 4 for grading purpose. A few remarks about our future work can be found in Section 5.

2 Implementation

2.1 Parfor

We have implemented the support of the parallel *for* loop. User can use the *parfor* keyword to indicate that a loop should be parallelized. However, user need to ensure that there is no

dependence between loop iterations. We parallelize the *parfor* loop with GPU by assigning iterations to different CUDA threads. The following pseudo code outlines the generation and execution of the *parfor* loop.

```
/* evaluate the memory requirements */
foreach line of parfor loop body do
    if not an assignment
        continue;   /* non-assignment commands are simply ignored */
    analyze and update the maximum index;
endfor
/* code generation */
generate the kernel code and function signature;
generate iteration mapping, boundary check, etc;
foreach line of parfor loop body do
    generate code for the destination sub-tree;
    generate code for the source sub-trees;
endfor
generate the host code, block/grid sizes, etc;

/* just-in-time compilation and execution */
compile as a dynamic library with nvcc
allocate memory for destination matrixes
load dynamic library at run-time
```

To parallelize a *parfor* loop, one challenge is the automatic detection of matrix dimension. In Matlab scripts, matrices are not declared before being used. Therefore, we need to analyze the input script and allocate a proper amount of memory for the matrices in the definition set before launching the generated kernel code. To address this problem, we pre-walk through the *parfor* block to determine the maximal possible dimension sizes for the target matrices. Under the assumption that the matrix index value is linearly correlated with the value of the basic induction variable, we execute the matrix index expression with the minimum and maximum induction variable values to calculate the memory requirement for each statement. We update the memory requirement whenever a new maximum is found until every statement inside the *parfor* loop is evaluated.

We then generated the CUDA host and kernel functions for the *parfor* loop. With the detected maximum memory requirements, we allocate memory for each matrix in the definition set, encapsulate the pointers and the loop specification (step, lower and upper bounds) in a structure, and pass the structure to the generated host function. Currently, we fix the block size to 256 and use a one dimensional block topology. We then dynamically generate the grid size specification according to the total number of iterations in the original

parfor loop. In the generated kernel code, we consider the basic induction variable, the lower loop bound, and the step size, and map a thread to an iteration with the following equation,

```
iteration_id = (threadIdx.x + blockIdx.x * blockDim.x) * step
     + lower_bound.
```

We further check the boundary condition in case there are more CUDA threads than the number of iterations in the *parfor* loop by generating the following statement,

```
if(iteration_id > upper_bound) return;
```

In this way, we establish the mapping between CUDA threads and *parfor* iterations. We then walk through the original *parfor* loop again to generate the CUDA code for each statement in the loop body. Finally, we compile the generated CUDA code as a dynamic library with nvcc and launch it by loading the dynamic library at run-time.

2.2 Parsec

The realization of using multiple streams relies on the support from underling libraries that we use. Fortunately, CUBLAS added an API for user to specify a stream for any subsequent CUBLAS library calls:

```
cublasStatus cublasSetKernelStream (cudaStream_t stream)
```

To use *parsec*, user needs to create a parsec block as shown in the script code below. S5 and S6 can be run independently, therefore they can be put into a parsec block. During run-time, Matcloud can generate two GPU streams and assign each command a stream. The independence of commands inside a *parsec* block is currently guaranteed by the user.

```
S1   a = rand(5,4);
S2   b = rand(4,3);
S3   c = rand(4,3);
S4   parsec
S5      d = a * b;
S6      e = a * c;
S7   endsec
```

3 Experimental Results

We ran the following simple script on our Matcloud framework with GTX 280. We also ran a similar script written in for loop on a modern dual-core laptop installed with Octave.

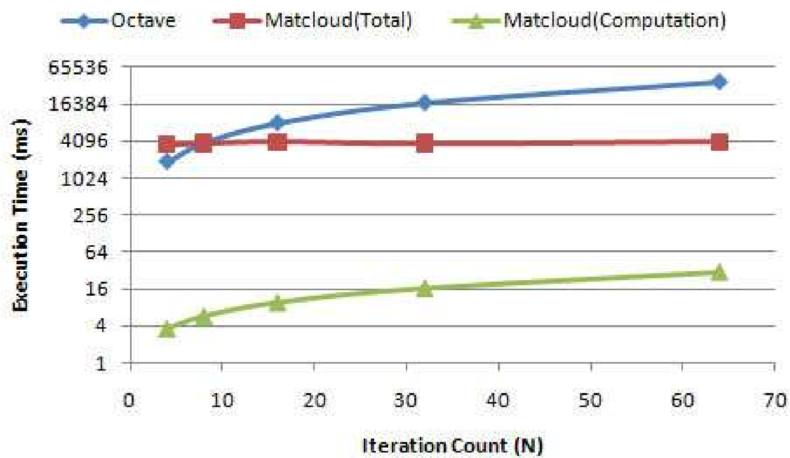

Figure 1: Matcloud vs. Octave

We varied the loop size (N) from 40,000 to 640,000 and compared their execution time, shown in Figure 3. As we can see, the pure execution time is almost negligible comparing to the compiling time. The total execution time in Matcloud is better than CPU version at N greater than 12,000.

```
parfor i=0:1:N
  a(i) = sin(i * 0.1);
  b(i) = cos(i * 0.1);
  c(i) = tan(i * 0.1);
endfor
```

Since GTX 280 is not a fermi architecture, it does not support concurrent kernel execution. We will skip the performance test for *parsec*.

4 Individual Contributions

Task Description	Effort [1]	Xing Wu	Yongpeng Zhang
Parfor loop parsing	2	100%	
Parsection loop parsing	2		100%
Launch nvcc at run-time	1		100%
Design code generation algorithm for parfor	3	50%	50%
Predict variable dimensions	3	50%	50%
Generating CUDA kernel code	3	100%	
Generating CUDA host-side code	3		100%
Manage CUDA streams	2	100%	
Implement matrix operators (+,/)	3	100%	
Implement matrix operators (-,*)	3		100%

[1] 1: easy; 2: medium; 3: complicated

5 Future Work

There are still a lot of room to improve our implementations.

- As shown in the experiment, compiling the parfor loop takes more time than the execution time. This is a common issue in all JIT framework. The usual approach is to cache compiled object file and reuse it whenever the same block is encountered again. Only by removing unnecessary compilng can the compiler time be amortized and this approach is justified.
- Support the parallel reduction. To use parallel reduction, user only need to specify a variable and an operator for the reduction. Compared to the *parfor* loop, parallel reduction requires the generation of the reduction tree for the CUDA kernel.
- Migrate our framework to fermi GPUs to demonstrate the benefits of using *parsection* block.

6 Appendix

In the Appendix, we show the mapping of one matlab script written in parfor format (Figure 6) to the cuda code (Figure 6).

```
1 parfor i=0:1:50000
2    a(i) = sin(i ∗ 0.1);
3    b(i) = cos(i ∗ 0.1);
4    c(i) = tan(i ∗ 0.1);
5 endfor
```

Figure 2: Original Parfor Matlab Script

```
1  #include <stdio.h>
2  #include <math.h>
3  #include <cuda.h>
4  typedef struct {
5     int numArgs;
6     void **arglist;
7     cudaStream_t stream;
8  } Args;
9  typedef float MyKernelDataType;
10 __global__ void mykernel(MyKernelDataType * a,MyKernelDataType * b,
       MyKernelDataType * c)
11 {
12    int i = (threadIdx.x + blockIdx.x * blockDim.x) * 1 + 0;
13    if (i > 50000)
14       return;
15    a[i]=sinf(i*0.1);
16    b[i]=cosf(i*0.1);
17    c[i]=tanf(i*0.1);
18 }
19 extern "C" void myparfor(Args *args)
20 {
21    MyKernelDataType * a = (MyKernelDataType *)args->arglist[0];
22    MyKernelDataType * b = (MyKernelDataType *)args->arglist[1];
23    MyKernelDataType * c = (MyKernelDataType *)args->arglist[2];
24    dim3 threads(256, 1, 1);
25    dim3 grids(196, 1, 1);
26    mykernel<<<threads, grids, 0, args->stream>>>(a,b,c);
27 }
```

Figure 3: Generated CUDA Code