Heterogeneous CPU+GPU Computing: Models, Tools, and Applications

Ana Lucia Varbanescu
Stijn Heldens

University of Amsterdam

February 4, 2017

Introduction

Heterogeneous systems combining CPUs and GPUs are becoming mainstream. The performance potential of these processors working together is significant. Yet most applications choose to use such systems as homogeneous, exploiting either the GPU or the CPU exclusively. The main goal of this tutorial is to demonstrate that implementing and deploying heterogeneous applications is simpler than expected, is supported by easy-to-use tools, and leads to significant performance gain for many applications.

For these exercises, we assume some experience with Unix command-line tools (`cd`, `mv`, `ssh`, etc.) and basic proficiency in C/C++ programming. You will be learning about working with the following tools:

- Low-level tools
  - CUDA+OpenMP
- High-level
  - StarPU[1]
  - Heterogeneous Programming Libray (HPL)[2]
  - OmpSs[3]
- Domain-specific (Graph Processing)
  - HyGraph[4]

[1] `http://starpu.gforge.inria.fr/`
[2] `https://github.com/fraguela/hpl`
[3] `https://pm.bsc.es/ompss`
[4] `https://github.com/stijnh/HyGraph`

Getting Started

For all exercises we will be using the DAS-5[5]: the national supercomputer of the Netherlands used for computer science research on parallel/distributed systems. You will obtain credentials for your DAS-5 account from the session coordinators. For this hands-on session, we will be working remotely using ssh.

On Window, you can log into the DAS-5 using WinSCP[6] with these options:

```
Host name: fs0.das5.cs.vu.nl
Port number: 22
```

On Unix, you can log in using ssh:

```
$ ssh username@fs0.das5.cs.vu.nl
username's password:
*******
[username@fs0 ~]$
```

After logging in, execute the following command to copy the .bashrc file with correct settings for all platforms:

```
[username@fs0 ~] $ cp /var/scratch/sheldens/ppopp17/.bashrc ~
```

Now load these settings using source:

```
[username@fs0 ~] $ source ~/.bashrc
```

After source, all tools that we will be using should be ready. You can check this by, for example, checking if OmpSs has initialized correctly. Your output should match the following:

```
[username@fs0 ~] $ mcc --version
mcxx 2.0.0 ((distributed version) 713c99c 2016-06-16 09:45:58 +0200)
```

Finally, copy the exercise directories to your home directory.

```
[username@fs0 ~] $ cp -r /var/scratch/sheldens/ppopp17/exercises ~
```

[5] http://www.cs.vu.nl/das5/
[6] https://winscp.net/

Part 1: Matrix-Matrix Multiplication

First, we will be experimenting with performance tuning. Programming using the different platforms is the goal of the second part of the session. Because matrix-matrix multiplication (DGEMM) is a fundamental kernel of many scientific applications, we will experiment with how this application can be expressed using different heterogeneous programming tools.

Disclaimer: the goal of this session is not to implement a highly optimized matrix multiplication routine (consider BLAS for these operations). Instead, matrix multiplication is only used to showcase how to program different tools.

Our kernel computes the product of two input matrices, `A` ($n \times p$) and `B` ($p \times m$), and writes the output in matrix `C` ($n \times m$). Figure 1 shows pseudo-code for how to multiply matrices `A` and `B` using the classic textbook method.

```
for i in 0...n:
  for j in 0...m:
    for k in 0...p:
      C[i][j] += A[i][k] * B[k][j]
```

Figure 1: Basic matrix-matrix multiplication

.

To parallelize this kernel, we split the work into tasks by dividing matrix `C` on a row-by-row basis. Each task computes a number of rows of matrix C. Figure 2 shows pseudo-code for such a parallel matrix multiplication.

```
parallel for task_id in 0...tasks:
  i_start = find_first_row_of_task(task_id)
  i_end = find_last_row_of_task(task_id)
  for i in i_start...i_end:
    for j in 0...m:
      for k in 0...p:
        C[i][j] += A[i][k] * B[k][j]
```

Figure 2: Parallel matrix-matrix multiplication

.

Framework

Navigate to the `matmul` directory:

```
[username@fs0 ~] $ cd ~/ppopp17/exercises/matmul
```

To ease your work, we have built a framework which provides a uniform command-line interface for all tools. The framework is responsible for generating random matrices, calling the platform-specific code to perform the matrix multiplication, and validating its output. You can see the internals of the framework by checking the code in `framework.c`, but this is not mandatory.

Sequential Baseline

To compare performance, we provide the implementation of a sequential version of matrix multiplication. Check out the code in `sequential.c` using `nano` or `vim` and make sure you understand the code.

```
[username@fs0 matmul] $ nano sequential.c
```

Next, run the code using the provided run script for matrices of size 1024×1024:

```
[username@fs0 matmul] $ ./run.sh ./sequential 1024
```

Important: ALWAYS use the provided `run.sh` *script to run benchmarks. This script compiles the code and submits a job to SLURM to run your program on one of the compute nodes. NEVER run programs directly on the master node since it is not intended for compute jobs.*

Run the sequential program for different matrix sizes to observe the difference in performance (i.e., the different execution times reported for different values of n, m, p). Keep in mind that matrix multiplication is an application with $\mathcal{O}(n^3)$ complexity: increasing the input dimensions by 2x increases the number of operations by 8x. Do you observe this behavior in the performance results?

OpenMP + CUDA

The file `cuda_openmp.cu` contains code for matrix multiplication using static partitioning by using OpenMP and CUDA. Examine the code using `nano` or `vim`

```
[username@fs0 matmul] $ nano cuda_openmp.cu
```

Next, execute the code on the CPU using this command:

```
[username@fs0 matmul] $ ./run.sh ./cuda_openmp 1024 CPU
```

Or execute the code on the GPU:

```
[username@fs0 matmul] $ ./run.sh ./cuda_openmp 1024 GPU
```

Experiment with different matrix sizes (for example, 2048, 4096, 8192) and understand how performance scales with the input matrix.

To combine CPU and GPU, use the following command:

```
[username@fs0 matmul] $ ./run.sh ./cuda_openmp 1024 HYBRID -v
```

By default, the matrix is partitioned 50%-50%: CPU and GPU both get half of the rows of `C`. The `-v` flag enables additional timing information, which consists of data transfer overhead and compute time of CPU and GPU. To adjust the partitioning, use the flag `-a` to set the $\alpha \in [0, 1]$ used for the division. A fraction α of the work is placed on the CPU and $1 - \alpha$ on the GPU.

For example, the following command will assign 70% of the work to the CPU and the remaning 30% to the GPU.

```
[username@fs0 matmul] $ ./run.sh ./cuda_openmp 1024 HYBRID -a 0.7
```

For different input sizes (e.g., 1024, 2048, 4096, 8192) experiment with this parameter and find the best value of α. For one input size, plot the run-time for different value of α (e.g., $\alpha = 0.0, 0.1, \ldots, 0.9, 1.0$).

StarPU

The file `starpu.c` contains matrix-multiplication code for StarPU using dynamic partitioning. Examine the source code and understand its contents.

```
[username@fs0 matmul] $ nano starpu.c
```

Run StarPU for different matrix sizes on either the CPU, the GPU or the CPU+GPU using the following command:

```
[username@fs0 matmul] $ ./run.sh ./starpu <size> <CPU or GPU or HYBRID>
```

Compare the performance of StarPU against CUDA+OpenMP. You will notice that StarPU is not much faster since it has not been tuned correctly. There are three reasons for this.

First, the code launches one task for every g rows which, by default, is set to 16. Increasing this value will increase the granularity of tasks and reduce overhead of launching tasks. You can change the value of g using the `-g` flag. Experiment with different values and see which value gives the best performance.

```
[username@fs0 matmul] $ ./run.sh ./starpu <size> HYBRID -g <rows-per-task>
```

Second, to achieve good scheduling, StarPU needs to be able to estimate, in advance, the duration of a task. To define a performance model for this example, open the file `starpu.c` and un-comment the line starting with `.model=&perf_model` inside the codelet definition. This will enable a simple performance history-based model which uses previous measurements.

Third, by default, StarPU uses a simple eager scheduler. Read the documentation[7] on different scheduling policies for StarPU and experiment with different schedulers. You set the scheduler by defining the environment variable `STARPU_SCHED`. For example, you can enable the `random` scheduler using the following command:

```
[username@fs0 matmul] $ ./run.sh env STARPU_SCHED=random ./starpu <size> HYBRID
```

See how performance improves by fine-tuning `-g` and the scheduling policy.

[7] `http://starpu.gforge.inria.fr/doc/html/OnlinePerformanceTools.html`

OmpSs

The file `ompss.c` contains matrix multiplication written in OmpSs. Examine the source code.

```
[username@fs0 matmul] $ nano ompss.c
```

Again, run the code for different sizes and targeting only CPU, only GPU, or CPU+GPU.

```
[username@fs0 matmul] $ ./run.sh ./ompss <size> <CPU or GPU or HYBRID>
```

To fine-tune the performance of OmpSs, you can modify the granularity parameter `-g`. Read the documentation[8] and experiment with different options. For example, you can enable enable overlapping GPU computation and GPU data transfers using the `NX_GPUOVERLAP` variable.

```
[username@fs0 matmul] $ ./run.sh env NX_GPUOVERLAP=yes ./ompss <size> HYBRID
```

Which options have a beneficial impact on the performance? Which options degrade performance?

Heterogeneous Programming Library (HPL)

The file `hpl.cpp` describes matrix multiplication in HPL using static partitioning. Examine the source code.

```
[username@fs0 matmul] $ nano hpl.cpp
```

You might notice that, contrary to the previous programs, HPL does not call the shared functions `matrix_multiply` and `cuda_matrix_multiply`. This is the case since HPL defines its own embedded programming language which compiles into OpenCL. The function `kernel_matrix_multiply` describes this kernel.

Run the code for different matrix sizes:

[8] `https://pm.bsc.es/ompss-docs/user-guide/run-programs.html`

```
[username@fs0 matmul] $ ./run.sh ./hpl <size> <CPU or GPU or HYBRID>
```

HPL uses static partitioning. Just like the CUDA+OpenMP program, you can set the value of α (i.e., fraction placed on CPU) using the flag `-a`. For example:

```
[username@fs0 matmul] $ ./run.sh ./hpl HYBRID -a 0.7
```

HPL allows users to define an `ExecutionPlan`. This object encapsulates the information needed to partition the work, which comes down to the percentage of work that each device will perform. The execution plan can also automatically balance the load among the devices by searching for the best work distribution when it is executed for the first time.

Currently, HPL offers three load-balance strategies[9]: `NOSEARCH` (no load-balancing), `ADAPT` (adapt weights after one run based on timing results), and `EXHAUS` (explore all possible divisions of work).

Modify the file `hpl.cpp` and change the execution plan from `NOSEARCH` to either `ADAPT` or `EXHAUS`. How does this change affect the performance?

Bonus: Non-Square Matrices

Up to this point, we have only explored the performance of matrix multiplication for square matrices since `A`, `B`, and `C` all have dimensions $n \times n$. Each program also has support for matrix multiplication of non-square matrices. Use the following command to perform matrix multiplication where the input matrices have dimensions $n \times p$ and $p \times m$, the output matrix has dimensions $n \times m$. Fix two of the parameters while varying the third parameter. Plot the results for one of the platforms.

```
[username@fs0 matmul] $ ./run.sh ./<binary> <n> <m> <p>
```

Bonus: Tile-based Matrix Multiplication

Our matrix multiplication based on a triple nested loop is not very efficient since it has poor data locality. Locality can be improved by *tiling*: dividing the matrices `A`, `B`, and `C` into $t \times t$ tiles. Each task now consists of computing one tile of `C`. Think about how you would implement such a tiling strategy for each of the platforms.

[9] `https://github.com/fraguela/hpl`

Figure 3: The three stages of the image pipeline.

Part 2: Image Processing Pipeline

For the second part of this hands-on session, it is time to get programming. We will consider an image pipeline as shown in Figure 3. The pipeline works on grayscale images where one pixel is one byte. Images are stored in row-major order. The goal is to implement this pipeline in StarPU, OmpSs, and HPL. The pipeline consists of three stages:

1. **Horizontal flip**: The image is flipped across the vertical axis, (i.e, from left to right). Since images are stored in row-major order, this corresponds to reversing the pixels in each row of the image.

2. **Finding minimum/maximum**: The minimum and maximum pixel values are detected. This corresponds to a reduction operation.

3. **Contrast adjustment**: Using the minimum and maximum values from the previous stage, each pixel value is transformed from the range $[min, max]$ to $[0, 255]$ using the following formula

$$\text{lbnd} = 0.9 \times \text{min} + 0.1 \times \text{max}$$

$$\text{ubnd} = 0.1 \times \text{min} + 0.9 \times \text{max}$$

$$\text{new\_pixel} = \begin{cases} 0 & \text{if old\_pixel} \leq \text{lbnd} \\ 255 & \text{if old\_pixel} \geq \text{ubnd} \\ \text{int}(255 \times \frac{\text{old\_pixel} - \text{lbnd}}{\text{ubnd} - \text{lbnd}}) & \text{Otherwise} \end{cases}$$

Framework

Navigate to the `pipeline` directory.

```
[username@fs0 ~] $ cd ~/ppopp17/exercises/pipeline
```

Provided is a framework which generates N images of size $n \times n$. The file `sequential.c` contains sequential code demonstrating how to use the framework. For example, the following command will process 100 random images having dimensions of 1024×1024 (i.e., each image is 1MB).

```
[username@fs0 pipeline] $ ./run.sh ./sequential 1024 100 --do-flip
      --do-contrast
```

The flags `--do-flip` and `--do-contrast` enable the first and second+third stage of the pipeline, respectively. You can omit one of them for debugging purposes.

CUDA code has also been provided in `cuda.cu`.

```
[username@fs0 pipeline] $ ./run.sh ./cuda 1024 100 --do-flip --do-contrast
```

Read through the sequential/CUDA code to get a feeling of how the framework is structured. You are free to use the library functions in `image.h` and `cuda_kernels.h` for your own program. There are two ways the pipeline can be parallelized:

- Process all images in parallel, but process each individual pipeline sequentially. This works best when processing many small images.
- Process all images sequentially, but perform each individual phase of the pipeline in parallel. This works best for few big images.

Your goal is to implement the image pipeline using one of three tools: StarPU, OmpSs, or HPL. StarPU and OmpSs use the first strategy, while HPL uses the second strategy. For each platform, skeleton code is provided that needs to be filled in to work correctly.

Pick the platform that appeals most to you and skip to the section of the platform of your choice.

StarPU

Open the file `starpu.c` and understand its contents. We refer to the StarPU documentation[10] if there are issues.

```
[username@fs0 pipeline] $ nano starpu.c
```

The code is not yet ready. Finish the code by following these steps.

Step 1: The function defined for the contrast adjustment (`task_contrast`) is empty. Fill it in by inserting a call to `image_contrast` with the correct arguments. Look at the functions `task_flip` and `task_find_min_max` for inspiration.

Step 2: Each StarPU requires the definition of a codelet structure which describes the input/output of the kernel and how it is implemented. The codelet for the contrast adjument function is missing. Create it by defining the `cl_contrast` codelet below the other codelet definitions.

Step 3: Call the codelet at the proper location and using the proper arguments using `starpu_task_insert`. When finished, test your code by running the complete pipeline.

```
[username@fs0 pipeline] $ ./run.sh ./starpu 1024 10 --do-flip --do-contrast
```

Step 4: To enable GPU acceleration, the codelet definitions need to be changed. Find the codelet definition for horizontal flip (`cl_flip`). Change the field `.where` from `STARPU_CPU` to `STARPU_CPU|STARPU_CUDA`. Also add an extra field `.cuda_func=gpu_task_field`. Afterwards, test your code, but only run the first stage of the pipeline:

```
[username@fs0 pipeline] $ ./run.sh ./starpu 1024 10 --do-flip
```

Step 5: The function for contrast adjustment on the GPU (`gpu_task_contrast`) is empty. Fill it in by insert a call to `cuda_launch_contrast` with the correct arguments. Apply the same changes as in step 4, but now for the codelet definitions `cl_find_min_max` and `cl_contrast`. Test your code and check if performance improves.

```
[username@fs0 pipeline] $ ./run.sh ./starpu 1024 10 --do-flip --do-contrast
```

[10]

Step 6: Experiment by varying the image size and the number of images. How do these parameters affect performance?

Bonus: Add a performance model to each codelet definition. Check the StarPU code for matrix multiplication to get a feeling of how insert performance models.

OmpSs

Open the file `ompss.c` and understand its contents. It is recommended to read through the OmpSs user guide[11] if you run into issues.

```
[username@fs0 pipeline] $ nano ompss.c
```

The code is unfinished. Five steps are required to finish the code.

Step 1: There are functions defined to perform the second (find min./max.) and the third stage (contrast adjustment) of the pipeline: `task_find_min_max` and `task_contrast`. However, these functions are not called anywhere. Insert function calls at the proper locations using the proper arguments. Test your code afterwards using the following command.

```
[username@fs0 pipeline] $ ./run.sh ./ompss 1024 10 --do-flip --do-contrast
```

Step 2: Your code is now working, but is not running parallel yet. To turn a regular function call into an OmpSs task, one needs to insert a `#pragma omp task` and define the data dependencies of each task using `in(...)` and `out(...)`. Insert these pragmas above every function prefixed with `task_`. Use the OmpSs code for matrix multiplication as inspiration. Test your code and check if performance improves.

```
[username@fs0 pipeline] $ ./run.sh ./ompss 1024 10 --do-flip --do-contrast
```

Step 3: Finally, to enable GPU acceleration, one needs to define GPU task function which overload the CPU task functions. Find the function named `gpu_task_flip` and define the pragmas `#pragma omp target device(cuda) implements(task_flip)` and `#pragma omp task in(...) out(...)`. Test your code, but only run the first stage of the pipeline:

[11] `https://pm.bsc.es/ompss-docs/user-guide/`

```
[username@fs0 pipeline] $ ./run.sh ./ompss 1024 10 --do-flip
```

Step 4: Also implement a GPU version of `task_find_min_max` and `task_contrast`. Test your code by running the complete pipeline:

```
[username@fs0 pipeline] $ ./run.sh ./ompss 1024 10 --do-flip --do-contrast
```

Step 5: Experiment by varying the image size and the number of images. How do these parameters affect performance.

Bonus: Similar to matrix multiplication, you can experiment with different run-time flags of OmpSs to fine-tine performance. How do the different flags affect performance?

Heterogeneous Programming Library (HPL)

Open the file `hpl.cpp` and understand the code structure. We recommend reading the HPL guide[12] in case you run into issues.

```
[username@fs0 pipeline] $ nano hpl.cpp
```

The code is nearly complete, but some steps are required to finish it.

Step 1: The contrast kernel has not yet been written. Write the code for this kernel by filling in `kernel_contrast`. Inspect `image_contrast` from `image.c` and matrix multiplication code for inspiration. Note that the kernel should be written in the HPL kernel language, NOT regular C++. The HPL kernel language is explained in the HPL manual. You can test your code using the following command:

```
[username@fs0 pipeline] $ ./run.sh ./hpl 1024 10 --do-flip --do-contrast
```

Step 2: The execution plan of each kernel is set to `NOSEARCH`, it thus does not perform any load-balancing. Experiment with different load-balancing strategies and observe which one gives the best performance.

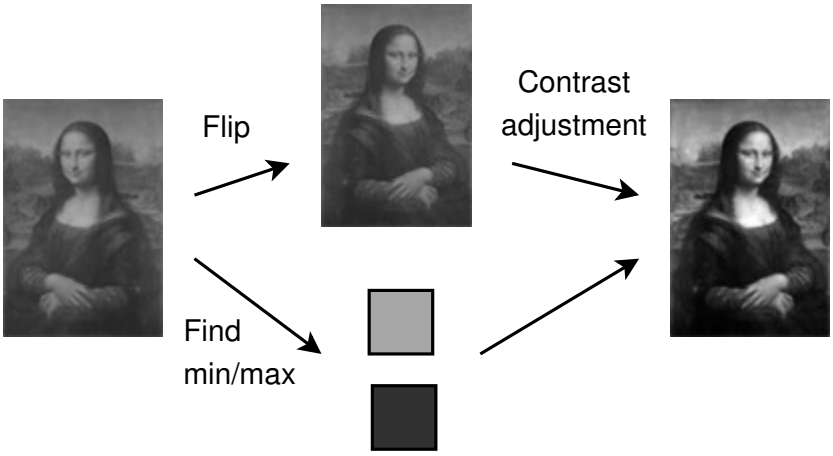

Figure 4: Alternative implementation of pipeline.

Bonus: Pipeline ⇒ Workflow

The pipeline presented in Figure 3 consists of three sequential stages. However, flipping the image does not affect the minimum/maximum pixel value. This means that the first stage and second stage can actually be executed in parallel. Think about what the data-dependencies are in this scenario. Think about how you would implement this change for your program in StarPU/OmpSs/HPL. Implement it if there is time.

Bonus: Image Smoothing

The parameter `--do-smooth` enables a mysterious fourth stage of the pipeline: **Smoothing**. This stage replaces the value of each pixel by the mean in a 3×3 neighborhood. For each pixel, the average of its own value and its 8 neighbors is calculated.

Implement or think about how you much work it would be to add this additional phase to the pipeline. How is smoothing different from contrast adjustment and flipping? Can smoothing be done in-place? You can use the functions `image_smooth` from `image.h` and `cuda_launch_smooth` from `cuda_kernels.h` if you are going to attempt implementing it.

[12] `http://hpl.des.udc.es/page1_assets/HPL_programming_manual.pdf`

Part 3: Graph Processing

For the final part of this hands-on session, you will experiment with graph processing on heterogeneous platforms. We will look into HyGraph. Navigate to the `graph_processing` directory.

```
[username@fs0 ~] $ cd ~/ppopp17/exercises/graph_processing
```

Examine the code using `nano` or `vim`. The main program can be found in `hygraph.cu`, while the HyGraph library is located in `HyGraph`.

```
[username@fs0 graph_processing] $ nano hygraph.cu
```

Execute the code. The program will generate a random graph having 2^k vertices and 10×2^k edges. Experiment with different values of k, for example $k = 15 \ldots 25$. What is the minimum value that CPU+GPU provides a performance benefit?

```
[username@fs0 graph_processing] $ ./run.sh ./hygraph k
```

The program solves the *single-source shortest paths* (SSSP) problem, implemented using the Bellman–Ford algorithm. The edges of the graph have weights which indicate distance. The SSSP algorithm computes the lengths of the shortest paths from one single source vertex to every other vertex in graph. Analyze the class `MyProgram` to understand how SSSP has been implemented.

The single-source *widest* path (SSWP) problem is a variation of SSSP. For SSWP, the weights on the edges indicate the *capacity* of the link (for example, the bandwidth between two routers). The goal of SSWP is not to find the shortest path between two points, but to find the path having *maximum capacity* for each link along the path (for example, the route between two servers having maximum bandwidth). Modify `MyProgram` such that it solves the SSWP problem instead of the SSSP problem.