# CS 205: Homework 2

Eric Dunipace

March 22, 2017

# 2. Analysis of Parallel Algorithms [10%]

(a) $\boxed{\textbf{Define iso-efficiency function for an ideally scalable parallel system.}}$

Iso-efficiency for an ideally scalable parallel system measures how well we can maintain the same efficiency as we add processors to a given problem. As Manju says in lecture 5, iso-efficiency is "the rate at which the problem size must increase with respect to the number of processing elements to keep the e ciency fixed." Put in another way, it is how easy it is to add processors to a given problem. Small iso-efficiency values indicate a problem is highly parallelizable; large iso-efficiency values indicate a problem is not highly parallelizable.

To derive this function, we start with efficiency, which equals

$$E = \frac{S}{p},$$

where $p$ is the number of processors and $S = \frac{T_1}{T_p}$ is the speedup achieved with $p$ processors. From here,

$$S = \frac{Wp}{W + T_o(W,p)},$$

where $W$ is the share of the serial work done on each parallel processor and $T_o(W,p)$ is the overhead of start-up and communication for the $p$ processors. Thus the work of one serial processor is simply $Wp$. Then, plugging back into efficiency,

$$E = \frac{\frac{Wp}{W+T_o(W,p)}}{p} = \frac{W}{W + T_o(W,p)} = \frac{1}{1 + \frac{T_o(W,p)}{W}}.$$

Solving for $W$ gives

$$\begin{aligned}
E &= \frac{1}{1 + \frac{T_o(W,p)}{W}} \\
E\left(1 + \frac{T_o(W,p)}{W}\right) &= 1 \\
E + E\frac{T_o(W,p)}{W} &= 1 \\
E\frac{T_o(W,p)}{W} &= 1 - E \\
\frac{ET_o(W,p)}{1-E} &= W,
\end{aligned}$$

then taking $\frac{E}{1-E}$ as a constant $K$ of the desired efficiency

$$W = KT_o(W,p)$$

(b) **Scaled speed-up is defined as the speedup obtained when the problem size is increased linearly with the number of processing elements; that is, if $W$ is chosen as a base problem size for a single processing element, then**

$$\textbf{scaled speedup} = \frac{pW}{T_p(pW,p)}.$$

**For the problem of adding $n$ numbers on $p$ processing elements, assume that it takes 20 time units to communicate a number between two processing elements, and that it takes one unit of time to add two numbers. Plot the standard speedup curve for the base problem size $p = 1, n = 256$ and compare it with the scaled speedup curve with $p = 2^2, 2^4, 2^5, 2^8$.**

Assuming that $T_p(pW, p)$ is the standard parallel time to run the algorithm, then the following code calculates the scaled speed up versus the standard speedup calculations.

```
import numpy as np
import matplotlib.pyplot as plt
import math
import matplotlib.patches as mpatches

#processors
exponents  = [2,4,5,8]
p = [2**i for i in exponents]

#problem size
n = 2**8 #256

#costs
comm = 20
W_t = 1 # work per time to sum 2 numbers

def serial_time(w_t, n):
   T_1 = n*w_t
   return(T_1)

def parallel_time(p, C, w_t, n):
   T_p = serial_time(w_t,n/p) + C
   for i in range(p):
       T_p += C + p/(2**i)*w_t
   return(T_p)

def standard_speed(p, C, w_t, n):
   T_p = parallel_time(p, C, w_t, n)
   T_1 = serial_time(w_t, n)

   return(T_1/T_p)

def scaled_speedNoComm(p, w_t, n):
   T_p = serial_time(w_t, n / p)
   T_s = 0

   total = T_s + T_p * p
   return(p - (p - 1) * (T_s / total))

def scaled_speed(p, C, w_t, n):
   T_p = serial_time(w_t, n/p) + C
   T_s = 0
   for i in range(p):
       T_s += C + p/(2**i)*w_t

   total = T_s + T_p * p
   return(p - (p - 1) * T_s/total)
   #return((T_s / total) + p * (1 - T_s / total))



scale = [scaled_speed(i,comm, W_t, n) for i in p]
scaleNoCom = [scaled_speedNoComm(i, W_t, n) for i in p]
stand = [standard_speed(i, comm, W_t, n) for i in p]
axes = plt.gca()
plt.figure(1)
plt.plot(p, stand, '-b', label='Standard Speed-up')
plt.plot(p, scale, '-r', label='Scaled Speed-up (With Communication)')
plt.plot(p, scaleNoCom, '-k', label='Scaled Speed-up (No Communication)')
axes.set_ylim([0,np.ceil(max(scale + stand + scaleNoCom))+1])
axes.set_xlim([0,max(p)])
plt.xlabel('Number of Processors')
plt.ylabel('Speed-up')
plt.title('Plot of Speed-up Versus Time')
plt.legend(loc=2)
plt.savefig("figures/speedup.png")

plt.figure(3)
axes = plt.gca()
plt.plot(p, stand, '-b', label='Standard Speed-up')
axes.set_ylim([0,np.ceil(max(stand ))])
axes.set_xlim([0,max(p)])
# plt.yscale('log')
plt.xlabel('Number of Processors')
plt.ylabel('Speed-up')
plt.title('Plot of Stand Speed-up Versus Time')
plt.savefig("figures/speedup2.png")
```

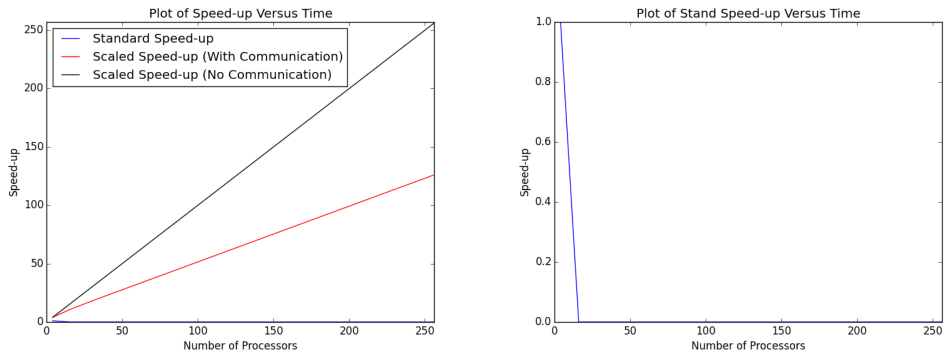

Figure 1: Comparison of standard speed-up versus scaled speed-up, where scaled speed-up increases the base problem size of $n = 256$ elements by a factor of the processors, $p$. Close up of standard speed up presented to actually see the line

## 3. Parallel Matrix Algorithms in CUDA and OpenAcc [40%]

In this problem we will revisit the tiling transformation from HW1 and investigate parallel algorithm design and implementation for Matrix Multiplication in the SIMT programming model for Manycore GPU architecture.

We have studied two ways to construct parallel programs for GPU architectures. CUDA programs use explicit parallel programming to express parallelism and to specify locality. OpenAcc, in contrast, provides a directive based mechanism to implement parallel programs through an automatic compilation system that generates parallel programs from annotated sequential programs. This offers a programming approach to rapidly implement different versions of parallel programs which simplifies the task of exploring the solution space for an efficient parallel program.

In this programming task you will implement and optimize the tiled Matrix Multiplication in the two programming languages as follows:

1. **CUDA**
   (a) **First benchmark a naive version (nested three loop matrix multiplication CUDA parallel program) against the reference implementation in CUBLAS. Generate a performance plot of throughput (GFlop/s) for problem sizes $n = 2^6, 2^{10}, 2^{16}$ for single precision floating point real numbers and compare against CUBLAS.**
   (b) **Implement a tiled parallel matrix multiplication program using the same relationship as before: make the block size $b$ as large as possible so that $3b^2 \leq M$, where $M$ is the size of the "fast" memory on a GPU node.**
   (c) **Benchmark for problem sizes $n = 2^6, 2^{10}, 2^{16}$.**
2. **OpenAcc**
   (a) **Implement a tiled parallel matrix multiplication using gang (thread blocks), vector (threads) and tile features for loop scheduling and optimization.**
   (b) **Tune the performance by varying gang, vector and tile parameters. Carefully consider the vector size in relation to warp size.**

(c) **Benchmark for problem sizes $n = 2^6, 2^{10}, 2^{16}$.**

(d) **What is the search space of the above brute force approach for very large problem size $n$? Is there a better way to search the space of parallel programs in (b) than brute force? Briefly discuss in the write-up.**

Submission: A write-up covering the following aspects uploaded as a single PDF file.

- A short one page description of the diffrent design principles that was used to develop successive versions of the parallel program.
- The optimizations used and the results obtained with reference to performane plots.
- Code listing and test cases (runs from single vs parallel to validate correctness of outputs).
- Performance plots of throughput (GFlop/s). The plot should indicate the peak throughput achievable on the GPU node and should compare the two versions of the implementations.