# ECE60827: Assignment 0

January 10, 2024

## Objective

The objective of this homework is to help you check if you have the necessary background for the course *ECE60827: Programmable Accelerator Architectures*. The topics covered are primarily along some algorithmic thinking, computer architecture, and C++ concepts such as object-oriented programming, inheritance, and polymorphism. Please complete this homework exercise. You do not need to submit the homework. This will not count towards your grade.

## Advice

While the use of generative AI (such as LLMs) may give 'seemingly correct' solutions, be aware that it may neither compile as expected nor produce the desired functionality. Obviously, such peripheral learning may not be in the best interest of your professional goals. Hence, you are advised to refrain from using such tools to complete the following tasks.

## Task 1: C++ concepts

Predict the output of the following C++ program

```cpp
#include <iostream>
using namespace std;
class Base {
public:
        virtual void print() const = 0;
};
class DerivedOne : virtual public Base {
public:
        void print() const
        {
                cout << "1";
        }
};
class DerivedTwo : virtual public Base {
public:
        void print() const
        {
                cout << "2";
        }
};
class Multiple : public DerivedOne, DerivedTwo {
public:
        void print() const
        {
                DerivedTwo::print();
        }
};
int main()
{
        Multiple both;
```

```
        DerivedOne one;
        DerivedTwo two;
        Base* array[3];
        array[0] = &both;
        array[1] = &one;
        array[2] = &two;
        for (int i = 0; i < 3; i++)
                array[i]->print();
        return 0;
}

```

## Task 2: Algorithmic thinking

A sparse vector is a vector that has mostly zero values. Write a C++ program to compute the dot product between 2 sparse vectors **efficiently** using as few compute operations as possible. The idea is to leverage the sparsity of the vectors to optimize the dot product computation. Given that most of the elements in the vectors are zeros, we want to perform multiplications only for the non-zero elements.

**Example**
Input: v1 = {1,0,0,2,3}, v2 = {0,3,0,4,0}
Output: 8
Explanation: dotProduct(v1, v2) = 1*0 + 0*3 + 0*0 + 2*4 + 3*0 = 8

You can use the following code template

```

#include <iostream>

double dotProduct(int size, double* v1, double* v2) {
    int i = 0, j = 0;
    double result = 0.0;

   // Do something here

    return result;
}

int main() {
    // Example usage
    int size = 5;
    double v1[] = {1.0, 0.0, 0.0, 2.0, 3.0};
    double v2[] = {0.0, 3.0, 0.0, 4.0, 0.0};

    double result = dotProduct(size, v1, v2);

    std::cout << "Dot product result: " << result << std::endl;

    return 0;
}

```

**Follow up**: how would you solve the problem if you stored only the non-zero elements of the vectors? As vectors get large, it is increasingly wasteful to store the zero values. What is done instead is to store a set of indices of the non-zero elements and their associated values. What is the time and space complexity of your solution?

## Task 3: Calculating arithmetic intensity

Arithmetic intensity (AI) is the measure of how many operations are done per bytes loaded or stored from memory.

$$AI = \frac{\#flops}{\#bytes}$$

While floating point calculations such as addition and multiplication require only one CPU cycle, and sometimes even less with fused instructions like FMA and FMAC in x86, the loads and stores can often take orders of magnitude longer.

Since memory operations are typically slow, it is crucial to maximize the number of operations performed on a piece of data once it is loaded, before loading new data. The concept of arithmetic intensity aims to quantify this.

**axpy:** The axpy routine is formally defined as $y = ax + y$ where $a \in R$ and $x, y \in R^n$.
Calculate the AI for axpy as shown in the below code

```
1
2  void daxpy(size_t n, double a, const double *x, double *y) {
3      size_t i;
4      for (i = 0; i < n; i++) {
5          y[i] = a * x[i] + y[i];
6      }
7  }
8
```

**GeMM:** The GeMM (General Matrix Multiplication) is another good example for calculating arithmetic intensity.

Let's just look at simple matrix multiplication: $C = AB$ where $A, B, C \in R^{n \times n}$.

Calculate the AI for GEMM as shown in the below code

```
1
2  void dgemm(size_t n, const double *A, const double *B, const double *C) {
3      size_t i, j, k;
4      double sum;
5      for (i = 0; i < n; i++) {
6          for (j = 0; j < n; j++) {
7              sum = 0.0;
8              for (k = 0; k < n; k++) {
9                  sum = sum + A[i*n+k] * B[k*n+j];
10             }
11             C[i*n+j] = sum;
12         }
13     }
14 }
15
```

*Note: While this is a CPU based implementation of these routines, you will be parallelizing it and converting it to **efficient** GPU code during various parts of the assignment. Hence, it helps to be familiar with such routines.*