Αρχιτεκτονική Προηγμένων Υπολογιστών και Επιταχυντών – Αναφορά 3ου εργαστηρίου

Δημήτριος Ορέστης Βαγενάς, 10595

Αγγελική Στρατάκη, 10523

Στο 3ο εργαστήριο εξετάσαμε μεθόδους βελτιστοποίησης της μεταφοράς δεδομένων από τον x86-host στον accelerator μέσω δύο παραδειγμάτων: 1) την επιτάχυνση πολλαπλασιασμού πινάκων ως διανύσματα 16 στοιχείων, εκμεταλλευόμενοι τη χωρητικότητα του διαύλου επικοινωνίας και 2) την εκμετάλλευση των memory bank DDR μνήμης της πλακέτας Alveo, οι εισόδοι/έξοδοι του accelerator ρυθμίζονται για ταυτόχρονη ανάγνωση/εγγραφή σε διαφορετικές banks, βελτιώνοντας τη μεταφορά δεδομένων.

Δημιουργήσαμε δύο αρχεία host.cpp και wide_vadd.cpp.

Στο αρχείο host.cpp ορίσαμε DATA SIZE 256(16 στοιχεία x 32 bits), δεσμεύσαμε μνήμη RAM που θα χρησιμοποιήσουμε

```
    // Allocate Memory in Host Memory
    size_t vector_size_bytes = sizeof(int) * DATA_SIZE;
    std::vector<unsigned int, aligned_allocator<unsigned int> > source_in1(DATA_SIZE);
    std::vector<unsigned int, aligned_allocator<unsigned int> > source_in2(DATA_SIZE);
    std::vector<unsigned int, aligned_allocator<unsigned int> > source_in2_transposed(DATA_SIZE);
    std::vector<unsigned int, aligned_allocator<unsigned int> > source_hw_results(DATA_SIZE);
    std::vector<unsigned int, aligned_allocator<unsigned int> > source_sw_results(DATA_SIZE);
    et.finish();
```

Στην συνέχεια κάναμε αναστροφή του δεύτερου πίνακα που έπρεπε να πολλαπλασιάσουμε και τον software υπολογισμό του πολλαπλασιασμού(16x16 στοιχεία) των δύων πινάκων , οι οποίοι έχουν οριστεί ως μονοδιάστατοι:

```
  // Software Result
  for (int i = 0; i < 16; i++) {

    for (int j = 0; j < 16; j++) {
        unsigned int result = 0;
        for (int k = 0; k < 16; k++) {
            result += source_in1[16*i+k]*source_in2[16*k+j];
        }
        source_sw_results[16*i+j] = result;
    }
  }


  for (int i = 0; i < 16; i++) {
      for (int j = 0; j < 16; j++) {
          //Transpose source in 2
          source_in2_transposed[i*16+j] = source_in2[j*16+i];
      }
}
```

Δεσμεύσαμε μνήμη BRAM για τον πρώτο πίνακα, τον ανάστροφο του δεύτερου και τον πίνακα που προκύπτει από τον πολλαπλασιασμό τους:

```
et.add("Allocate Buffer in Global Memory");
// Allocate Buffer in Global Memory
OCL_CHECK(err, cl::Buffer buffer_in1(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, vector_size_bytes,
                                     source_in1.data(), &err));
OCL_CHECK(err, cl::Buffer buffer_in2(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, vector_size_bytes,
        source_in2_transposed.data(), &err));
OCL_CHECK(err, cl::Buffer buffer_output(context, CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY, vector_size_bytes,
                                        source_hw_results.data(), &err));
et.finish();
```

Στο αρχείο wide_vadd.cpp για buffer size 16, αφαιρέσαμε τη stream μέθοδο του παραδείγματος και το dataflow directive και υλοποιήσαμε την παρακάτω for loop για τον υπολογισμό των πινάκων στο kernal. Στη πρώτη επανάληψη πήραμε την πρώτη γραμμή του πίνακα A v1_local[0] με την πρώτη γραμμή του B v2_local[0].Μετά στην εσωτερική λούπα του vector ορίζουμε δύο 32bit μεταβλητές και με τη συνάρτηση range διαβάζουμε το αντίστοιχο 32bit στοιχείο των v1_local και v2_local κάνουμε τον πολλαπλασιασμό και αποθηκεύουμε το αποτέλεσμα στο tmpresult , το οποίο μετά αποθηκεύεται σε μια 32bit θέση του tmpOut .Αντίστοιχα για την δεύτερη γραμμή του v2_local προκύπτει το 2ο στοιχείο της πρώτης γραμμής του πίνακα Γ και συνεχίζει μέχρι να συμπληρωθεί η 1η γραμμή του Γ .Συνεχίζοντας παίρνουμε το v1_local[1] και προκύπτει η 2η γραμμή του πίνακα Γ .

```
    for (int j = 0; j < chunk_size; j++) {
    #pragma HLS pipeline
    #pragma HLS LOOP_TRIPCOUNT min = 1 max = 16//buffer size**********
                    tmpV1 = v1_local[j];
                    tmpOut = 0;

                    for (int l = 0; l < chunk_size; l++){
                    tmpV2 = v2_local[l];

                    ap_uint<32> tmpresult = 0;

                    for (int vector = 0; vector < VECTOR_SIZE; vector++) {
                       #pragma HLS UNROLL
                        ap_uint<32> tmp1 = tmpV1.range(32 * (vector + 1) - 1, vector * 32);
                        ap_uint<32> tmp2 = tmpV2.range(32 * (vector + 1) - 1, vector * 32);
                        //tmpOut.range(32 * (vector + 1) - 1, vector * 32) = tmp1 * tmp2;
                        tmpresult += tmp1 * tmp2;
                        }
                    tmpOut.range(32 * (l + 1) - 1, l* 32) = tmpresult;
                    }

            out[i + j] = tmpOut;
    }
```

Έχουμε kernel execution time 0.009, γεγονός που επιβεβαιώνει ότι βελτίωση του performance που πετύχατε συγκριτικά με το 2ο εργαστήριο είναι μεγαλύτερη περίπου κατά 9 φορές (0.082/0.009 = 9.111)

Kernel Execution (includes estimated device times)

Kernel	Enqueues	Total Time (ms)	Min Time (ms)	Avg Time (ms)	Max Time (ms)
vadd	1	0.009	0.009	0.009	0.009