

Αρχιτεκτονική Προηγμένων Υπολογιστών και Επιταχυντών – Αναφορά 2^{ου} εργαστηρίου

Δημήτριος Ορέστης Βαγενάς, 10595

Αγγελική Στρατάκη, 10523

Για το δεύτερο εργαστήριο χρησιμοποιήσαμε την εφαρμογή Vitis της Xilinx. Αρχικά, εκτελούμε τις οδηγίες του αρχείου introduction to vitis μια προς μια και τρέχουμε software και hardware emulation. Δεν παρατηρήθηκε κάποιο πρόβλημα κατά την εκτέλεση των οδηγιών, οπότε προχωρήσαμε στον επόμενο βήμα. Στην αρχή εισάγαμε το κώδικα που αναπτύξαμε στο πρώτο εργαστήριο. Ο διαχωρισμός των αρχείων για το testing του προγράμματος διατηρεί παρόμοια λογική, με διαφορετικά αρχεία να διαχειρίζονται τη λειτουργία της CPU και της κάρτας επιτάχυνσης (FPGA). Σε αυτή την περίπτωση, έχουμε δύο ξεχωριστούς τύπους μνήμης: την καθολική μνήμη της FPGA, όπου ο πυρήνας της μπορεί να επεξεργάζεται τα δεδομένα του προγράμματος, και τη RAM της CPU, που φιλοξενεί τα υπόλοιπα δεδομένα του host προγράμματος. Ο στόχος ήταν να εξασφαλιστεί ότι ο κώδικας λειτουργεί σωστά, τουλάχιστον στις δύο από τις τρεις λειτουργίες του Vitis (SW emulation και HW emulation), χωρίς να προκαλούνται σφάλματα. Αφού ολοκληρώθηκαν τα βήματα του παραδείγματος και δημιουργήθηκε το host αρχείο μας, προχωρήσαμε στις αναγκαίες προσαρμογές για να ταιριάζει στη δική μας υλοποίηση.

Στο αρχείο host (host_1) προσθήσαμε 1 (το κομμάτι που θα γεμίζει τους πίνακες με τυχαίες τιμές και 2) το κομμάτι αυτό του κώδικα εκτελεί τον πολλαπλασιασμό των πινάκων και αποθηκεύει τα αποτελέσματα στον πίνακα C.

1)

```
et.add("Fill the buffers");
```

```
std::generate(source_in1.begin(), source_in1.end(),[](){return std::rand() %256;});
```

```
std::generate(source_in2.begin(), source_in2.end(),[](){return std::rand() %256;});
```

```
int result = 0;
```

2)

```
for (int i = 0; i < N; i++) {
```

```
    for (int j = 0; j < P; j++) {
```

```
        result = 0;
```

```
        for (int k = 0; k < M; k++) {
```

```
            result += source_in1[M*i+k]*source_in2[P*k+j];
```

```
        }
```

```
        source_sw_results[P*i+j] = result;
```

```
    }
```

```
}
```

3) Συγκρίνουμε τα αποτελέσματα του host με εκείνα του kernel για να καθορίσουμε αν το test του κώδικα πέρασε (passed) ή απέτυχε (failed).

```

bool match = 1;

for(int i=0; i<N*P; i++){

    if(source_hw_results[i] != source_sw_results[i]) match = false;

}

```

Όσον αφορά των κώδικα του kernel, κρατήθηκαν κάποια κομμάτια του αρχικού κώδικα vadd με κύρια αλλαγή τον data_size στα 16 καθώς είναι η τιμή τη; κάθε διάστασεις των πινάκων μας. Χρησιμοποιήθηκαν τα pragms:

```

#pragma HLS ARRAY_PARTITION variable=A complete

#pragma HLS ARRAY_PARTITION variable=B complete

#pragma HLS ARRAY_PARTITION variable=v1_buffer complete dim=2

#pragma HLS ARRAY_PARTITION variable=v2_buffer complete dim=1

#pragma HLS ARRAY_PARTITION variable=vout_buffer complete dim=2

```

Για τον χωρισμό των πινάκων σε στηλές ή γραμμές (dim=1 αντιστοιχεί στις στηλές και είναι η default τιμή αν δεν οριστεί η διάσταση και dim=2 για τις γραμμές). Στην συνέχεια βάζουμε των αρχικό μας κώδικα και χωρίζεται σε τρία στάδια. Ο πρώτος εμφωλευμένος βρόγχος φορτώνει τα δεδομένα εισόδου τα οποία έρχονται μέσω τις global memory σε δυο buffers, ο δεύτερος βρόγχος υπολογίζει την τιμή κάθε στοιχείου της πράξης σε έναν buffer, ο οποίος θέτει κάθε στοιχείο του στην έξοδο στον επόμενο βρόγχο. Χρησιμοποιήσαμε το pragmas unroll για την παράλληλη επεξεργασία των δεδομένων.

```

    for (int i = 0; i < size; i++) {

#pragma HLS UNROLL

        for (int j = 0; j < size; j++) {

            v1_buffer[i][j] = A[i*size + j];

            v2_buffer[i][j] = B[i*size + j];

        }

    }

    for(int i=0; i < size; i++){

    for (int j = 0; j < size; ++j){

        vout_buffer[i][j] = 0;

        for (int k = 0; k < size; ++k){

#pragma HLS UNROLL

            vout_buffer[i][j] += v1_buffer[i][k] * v2_buffer[k][j];

        }

    }

}

```

```

for(int i=0; i<size ;i++){
    for (int j = 0; j < size; j++) {
#pragma HLS UNROLL
        C[i*size + j] = vout_buffer[i][j];
    }

}


}
}

```

Παρακάτω παρουσιάζονται οι τιμές που συλλέξαμε από τα αποτελέσματα των emulations του κώδικα:


i) Kernel Execution

Kernel Execution (includes estimated device times)

Kernel	Enqueues	Total Time (ms)	Min Time (ms)	Avg Time (ms)	Max Time (ms)
 vadd	1	0.082	0.082	0.082	0.082

ii) Top Kernel Data Transfer

Top Kernel Transfer

Compute Unit	Device	Number of Transfers	Avg Bytes per Transfer	Transfer Efficiency (%)	Total Data Transfer (MB)	Total Write (MB)	Total Read (MB)	Total Transfer Rate (MB/s)
 vadd_1	xilinx_u200_gen3x16_xdma_2_202110_1-0	528	5.000	0.142	0.003	0.001	0.002	858.101

iii) Host Transfer

Data Transfer: Host from/to Global Memory

Context: Number of Devices	Transfer Type	Number of Buffer Transfers	Transfer Rate (MB/s)	Avg Bandwidth Utilization (%)	Avg Size (KB)	Total Time (ms)	Avg Time (ms)
context0:1	READ	1	0.050	N/A	1.024	N/A	N/A
context0:1	WRITE	1	0.050	N/A	2.048	N/A	N/A