ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ
http://www.cslab.ece.ntua.gr

ΣΥΣΤΗΜΑΤΑ ΠΑΡΑΛΛΗΛΗΣ ΕΠΕΞΕΡΓΑΣΙΑΣ

9ο εξάμηνο ΗΜΜΥ, ακαδημαϊκό έτος 2014-15

ΑΣΚΗΣΗ 2:

Παράλληλος προγραμματισμός σε επεξεργαστές γραφικών

Προθεσμίες παράδοσης:
Έλεγχος προόδου **27 Ιανουαρίου**
Επίδειξη προγραμμάτων **18 Μαρτίου**
Τελική αναφορά **25 Μαρτίου**

I Ο υπολογιστικός πυρήνας Jacobi σε επεξεργαστές γραφικών (GPUs)

```
for (t = 1; t < T && !converged; t++) {
  for (i = 1; i < N; i++) {
    for (j = 1; j < N; j++) {
      A[t][i][j] = 0.25*(A[t—1][i—1][j]+A[t—1][i+1][j]+A[t—1][i][j—1]+A[t—1][i][j+1]);
    }
  }
  converged = check_convergence();
}
```

1 Παραλληλοποίηση και ζητήματα επίδοσης

Ο υπολογιστικός πυρήνας Jacobi, αλλά και γενικότερα οι stencil υπολογισμοί, θέτουν τους παρακάτω βασικούς περιορισμούς σε συσκευές GPUs:

- Η halo περιοχή που αντιστοιχεί σε κάθε thread block πρέπει να ξαναδιαβαστεί από την κύρια μνήμη σε κάθε χρονικό βήμα, εφόσον οι τιμές αυτές τροποποιούνται από γειτονικά thread block.
- Οι νέες τιμές που υπολογίζει κάθε thread block πρέπει να γραφτούν στην κύρια μνήμη σε κάθε χρονικό βήμα, εφόσον οι τιμές αυτές μπορεί να χρησιμοποιηθούν από γειτονικά thread block.
- Απαιτείται global συγχρονισμός στο τέλος κάθε χρονικού βήματος ώστε να εξασφαλίσουμε ότι έχουν υπολογιστεί όλες οι τιμές του πίνακα πριν προχωρήσουμε στο επόμενο χρονικό βήμα.

Αναλογιστείτε πώς οι παραπάνω περιορισμοί επηρεάζουν την υλοποίηση της μεθόδου Jacobi στην αρχιτεκτονική CUDA. Επιπλέον, σχολιάστε κατά πόσο η μέθοδος είναι κατάλληλη για υλοποίηση σε επεξεργαστές γραφικών γενικότερα.

2 Υλοποίηση

Στο πρώτο μέρος αυτής της άσκησης θα πρέπει να υλοποιήσετε τον υπολογιστικό πυρήνα Jacobi για τους επεξεργαστές γραφικών του εργαστηρίου. Ξεκινώντας από μία απλοϊκή αρχική υλοποίηση, στο τέλος της άσκησης θα έχετε πετύχει μία αρκετά αποδοτική υλοποίηση του αλγορίθμου για τους επεξεργαστές γραφικών, χρησιμοποιώντας το προγραμματιστικό περιβάλλον CUDA.

2.1 Βασική υλοποίηση

Υλοποιήστε τον πυρήνα Jacobi για τους επεξεργαστές γραφικών, αναθέτοντας σε κάθε νήμα εκτέλεσης τον υπολογισμό ενός στοιχείου του πίνακα εισόδου.

- Ποιες είναι οι θεωρητικά βέλτιστες διαστάσεις του thread block για την συγκεκριμένη υλοποίηση στην αρχιτεκτονική Fermi; Δικαιολογήστε την απάντησή σας και επιβεβαιώστε ότι ισχύει. Καταγράψτε στην περίπτωση αυτή την χρησιμοποίηση των πολυεπεξεργαστών ροών (SMs). Για ευκολία μπορείτε να κάνετε χρήση του CUDA Occupancy Calculator.

  Hint: μπορεί να υπάρχουν περισσότεροι του ενός συνδυασμοί διαστάσεων thread block που να είναι θεωρητικά βέλτιστοι.

- Εκτελέστε το πρόγραμμά σας για διάφορα μεγέθη πίνακα εισόδου, με διαστάσεις από 1024 έως 16384, και τους θεωρητικά βέλτιστους συνδυασμούς διαστάσεων thread block που υπολογίσατε. Πώς διακυμαίνεται η επίδοση; Συγκρίνετέ την με την βέλτιστη επίδοση της υλοποίησης σε CPU με OpenMP.

- Επιτυγχάνεται συνένωση των προσβάσεων στην κύρια μνήμη (memory access coalescing) με αυτή την υλοποίηση; Σχολιάστε κατά πόσο ο αλγόριθμος ευνοεί τη συνένωση των προσβάσεων.

2.2 Χρήση της μοιραζόμενης μνήμης (shared memory)

Με βάση την προηγούμενη υλοποίηση, χρησιμοποιείστε την τοπική μνήμη των SMs (shared memory), ώστε να προφορτώνετε (prefetch) τμηματικά τα δεδομένα που αντιστοιχούν σε κάθε thread block και στην συνέχεια να εκτελείτε τους υπολογισμούς από αυτή.

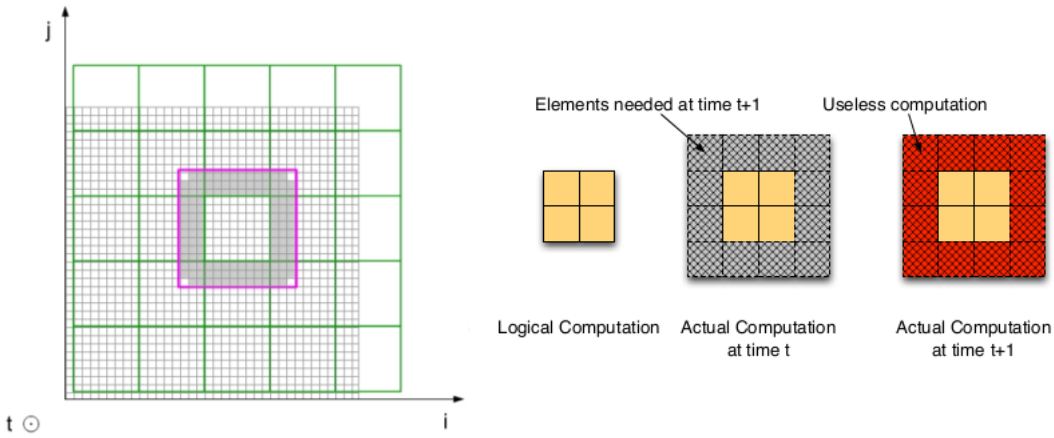
- Εκτελέστε το πρόγραμμά σας για διάφορα μεγέθη πίνακα εισόδου, με διαστάσεις από 1024 έως 16384, και τους βέλτιστους συνδυασμούς διαστάσεων thread block που υπολογίσατε για την προηγούμενη υλοποίηση. Πώς διακυμαίνεται η επίδοση; Γιατί ο κώδικας σας είναι απογοητευτικά αργός σε σχέση με την προηγούμενη υλοποίηση;

- Τροποποιήστε τον κώδικά σας ώστε κάθε νήμα να αναλαμβάνει τον υπολογισμό περισσότερων του ενός στοιχείων. Πού οφείλεται (αν υπάρχει) η βελτίωση στην επίδοση; Συγκρίνετέ την με την βέλτιστη επίδοση της υλοποίησης σε CPU με OpenMP. Καταγράψτε στην περίπτωση αυτή την χρησιμοποίηση των πολυεπεξεργαστών ροών (SMs). Για ευκολία μπορείτε να κάνετε χρήση του CUDA Occupancy Calculator.

2.3 Time-tiled υλοποίηση

Μία τεχνική που έχει προταθεί για τον περιορισμό της ανταλλαγής δεδομένων μεταξύ γειτονικών thread block και, κατ'επέκταση, των προσβάσεων στην κύρια μνήμη, είναι το λεγόμενο *time tiling*. Αντί να επιβάλλουμε καθολικό συγχρονισμό σε κάθε χρονικό βήμα, προκειμένου κάθε thread block να διαβάσει/γράψει τις απαραίτητες τιμές από/προς την κύρια μνήμη, αναθέτουμε σε κάθε thread block να κάνει επιπλέον υπολογισμούς σε μία επεκτεταμένη halo περιοχή, ώστε να εξασφαλίσει τις τιμές που χρειάζεται σε επόμενα χρονικά βήματα. Πιο συγκεκριμένα, αν θέλουμε να γίνεται συγχρονισμός ανά T χρονικά βήματα (*time step*), τότε κάθε thread block κάνει υπολογισμούς σε μία περιοχή με διαστάσεις $(n + 2 * T) \times (n + 2 * T)$, προκειμένου να υπολογίσει το $n \times n$ μπλοκ δεδομένων, χωρίς να πάει στην κύρια μνήμη. Για παράδειγμα, για $T = 2$ έχουμε τους υπολογισμούς που φαίνονται στο δεξί τμήμα του Σχήματος 1.

Αξιοποιώντας τις γνώσεις που αποκομίσατε από τις προηγούμενες υλοποιήσεις, υλοποιήστε την *time-tiled* εκδοχή του πυρήνα Jacobi. Πειραματιστείτε με διάφορα μεγέθη των time step, διαστάσεων tile και διαστάσεων thread block.



Σχήμα 1: Time tiling technique.

- Αναλογιστείτε πώς θα γίνει η διαχείριση της halo περιοχής σε επίπεδο thread block.
- Πόσοι διαφορετικοί πυρήνες απαιτούνται για μία αποδοτική υλοποίηση;
- Για τον καλύτερο συνδυασμό των παραμέτρων time step, διαστάσεων tile και διαστάσεων thread block, εκτελέστε το πρόγραμμά σας για διάφορα μεγέθη πίνακα εισόδου, με διαστάσεις από 1024 έως 16384. Πώς διακυμαίνεται η επίδοση σε σχέση με την προηγούμενη υλοποίηση; Συγκρίνετέ την με την βέλτιστη επίδοση της υλοποίησης σε CPU με OpenMP.

3 Υποδείξεις και διευκρινίσεις

3.1 Διευκρινίσεις για την υλοποίηση

- Η μόνη παραδοχή που μπορείτε να κάνετε για τις διαστάσεις του πίνακα είναι ότι θα είναι αρκετά μικρές ώστε οι υλοποιήσεις σας να μην επηρεάζονται από τις μέγιστες επιτρεπτές διαστάσεις των grid/thread block.
- Η υλοποίηση του ελέγχου σύγκλισης είναι προαιρετική (bonus).
- Σε κάθε branch στον κώδικά σας θα επισημαίνετε με σχόλιο αν προκαλεί warp divergence.

3.2 Υποδείξεις για την υλοποίηση

- Ελαχιστοποίηση των διακλαδώσεων που δημιουργούν divergence στο εσωτερικό των warps.
- Ειδική μεταχείριση της halo περιοχής.

3.3 Ερωτήματα (bonus)

- Στην υλοποίηση που σας παρέχεται, προκειμένου να επιβληθεί global συγχρονισμός μετά από κάθε χρονικό βήμα, εκτελείται ένα kernel launch για κάθε χρονικό βήμα. Επομένως, για n χρονικά βήματα, γίνονται n kernel launches. Αυτό επιφέρει ένα μη αμελητέο κόστος. Υπάρχει κάποια εναλλακτική λύση;

3.4 Δομή κώδικα

Για την διευκόλυνσή σας, αλλά και για να υπάρχει ένας κοινός τρόπος μέτρησης του χρόνου εκτέλεσης, σας δίνετε πλήρης και λειτουργικός σκελετός του κώδικα της άσκησης, καθώς και οι ρουτίνες της σειριακής και παράλληλης έκδοσης του Jacobi με OpenMP (για λόγους αποσφαλμάτωσης αλλά και σύγκρισης επιδόσεων). Ο κώδικας που σας παρέχεται αποτελείται από τέσσερα (4) μέρη:

Κυρίως πρόγραμμα: Πρόκειται για το αρχείο main.cu, το οποίο περιέχει την συνάρτηση main() του προγράμματος, η οποία είναι υπεύθυνη (α΄) για την ανάγνωση των ορισμάτων της γραμμής εντολών και των

μεταβλητών περιβάλλοντος (βλ. παρακάτω), (β′) για την δημιουργία του πίνακα εισόδου, (γ′) για την εκτέλεση και χρονομέτρηση του σειριακού και του παράλληλου με OpenMP πυρήνα, (δ′) για την αρχικοποίηση (παραχωρήσεις μνήμης, παράμετροι πυρήνα), εκτέλεση και χρονομέτρηση του πυρήνα στην GPU, και (ε′) για τον έλεγχο της εγκυρότητας των αποτελεσμάτων.

Υλοποιήσεις για CPU: Πρόκειται για το αρχείο `cpu_kernels.c`, το οποίο περιέχει την σειριακή και παράλληλη υλοποίηση του Jacobi για τις CPUs.

Υλοποιήσεις για GPU: Πρόκειται για τα αρχεία `gpu_kernel_naive.cu`, `gpu_kernel_shmem.cu` και `gpu_kernel_time_tiled_shmem.cu`, τα οποία περιέχουν τις υλοποιήσεις των αντίστοιχων πυρήνων για GPUs.

Βοηθητικές συναρτήσεις: Πρόκειται για τα αρχεία `alloc.c`, `error.c`, `helper.c`, `timer.c` και `gpu_util.cu`, τα οποία περιέχουν βοηθητικές συναρτήσεις για δυναμική παραχώρηση μνήμης διδιάστατου πίνακα, χειρισμού λαθών, συναρτήσεων ελέγχου και αρχικοποίησης των δεδομένων του πυρήνα καθώς και χρονομέτρησης και χειρισμού των GPUs. Υπάρχει επίσης το αρχείο `kernel.h` στο οποίο ορίζονται κάποιες βοηθητικές παράμετροι για τις διαστάσεις του thread block, του tile και του time step.

Σας παρέχετε επιπλέον και το κατάλληλο Makefile για την μεταγλώττιση και την σύνδεση του κώδικά σας. Πληκτρολογείστε "`make help`", ώστε να δείτε τις διάφορες επιλογές που σας δίνονται κατά την μεταγλώττιση.

Τα σημεία του κώδικα, στα οποία θα πρέπει να επέμβετε είναι σημειωμένα με την επιγραφή "`FILLME:`" μαζί με μία σύντομη περιγραφή. Οι ζητούμενες προσθήκες θα γίνουν επί της ουσίας στα αρχεία `gpu_kernel_naive.cu`, `gpu_kernel_shmem.cu`, `gpu_kernel_time_tiled_shmem.cu` και `cpu_kernels.c`. Τέλος, για να δείτε τον τρόπο χρήσης του τελικού εκτέλεσιμου, τρέξτε "`./jacobi`" από την γραμμή εντολών και θα παρουσιαστεί ένα σύντομο μήνυμα βοήθειας για την σωστή χρήση του. Τα σημεία στα οποία αναγράφεται η ένδειξη "`FILLME BONUS:`" είναι προαιρετικά.

3.5 Περιβάλλον και διαδικασία ανάπτυξης

3.5.1 Στο εργαστήριο

Στο εργαστήριο η διαδικασία ανάπτυξης της άσκησης χωρίζεται σε δύο φάσεις:

1. στην φάση ανάπτυξης και αποσφαλμάτωσης (debugging) και
2. στην φάση δοκιμών και πειραμάτων στους επεξεργαστές γραφικών.

Κατά την διάρκεια της πρώτης φάσης μπορείτε να δουλεύετε στα clones (όπως στις προηγούμενες ασκήσεις), όπου υπάρχει εγκατεστημένο το περιβάλλον ανάπτυξης CUDA 2.3. Ωστόσο, τα clones δεν έχουν κάρτα γραφικών, οπότε θα πρέπει να μεταγλωττίζετε και να εκτελείτε κατάλληλα τον κώδικά σας για λειτουργία εξομοίωσης (emulation mode). Αυτό μπορείτε να το πετύχετε εύκολα σε κάποιο clone, εκτελώντας απλά "make EMU=1". Ο λόγος ύπαρξης αυτής της φάσης είναι καθαρά για έλεγχο λαθών στις υλοποιήσεις σας· οι επιδόσεις των πυρήνων για GPU σε λειτουργία εξομοίωσης δεν έχουν καμία σχέση με την πραγματικότητα, οπότε μην προσπαθήσετε να ερμηνεύσετε τα αποτελέσματα.

Κατά την δεύτερη φάση της υλοποίησης θα τρέξετε τον κώδικά σας σε μηχάνημα του εργαστηρίου με εγκατεστημένη κάρτα γραφικών γενιάς 2.0 (NVIDIA Tesla M2050, αρχιτεκτονική Fermi). Για περισσότερες πληροφορίες σχετικά με τα λεπτομερή τεχνικά χαρακτηριστικά της GPU, πληκτρολογείστε "make query" όντας σε ένα μηχάνημα termi.

Σημείωση: στα clones, ίσως χρειαστεί πριν από την εκτέλεση να τρέξετε την εντολή

```
$ export LD_LIBRARY_PATH=/usr/local/cuda/lib64:${LD_LIBRARY_PATH}
```

3.5.2 Στο σπίτι

Μπορείτε να δουλεύετε και στο δικό σας σύστημα είτε με χρήση της λειτουργίας εξομοίωσης, όπως στα clones, είτε –εάν έχετε CUDA-enabled κάρτα γραφικών– απευθείας στην κάρτα γραφικών σας. Εάν πρόκειται να χρησιμοποιήσετε την λειτουργίας εξομοίωσης, ωστόσο, σας προτείνουμε να χρησιμοποιήσετε το περιβάλλον ανάπτυξης CUDA 2.3, καθότι στις επόμενες εκδόσεις η υποστήριξη της συγκεκριμένης λειτουγίας έχει εγκαταλειφθεί. Εναλλακτικά, θα πρέπει να δοκιμάσετε άλλου είδους προσομοιωτές επεξεργαστών γραφικών, π.χ.,

Ocelot. Τέλος, εάν διαλέξετε κάποια δική σας πλατφόρμα για την ανάπτυξη του κώδικά σας, το πιθανότερο είναι να πρέπει να αλλάξετε κάποιες μεταβλητές του Makefile που σας δίνετε. Για περισσότερες πληροφορίες, μπορείτε να επικοινωνήσετε με τους βοηθούς του εργαστηρίου.

4 Πειράματα και μετρήσεις επιδόσεων

4.1 Διαδικασία μετρήσεων

Σκοπός των μετρήσεων της άσκησης αυτής είναι η σύγκριση της επίδοσης των τριών εκδόσεων του Jacobi για GPUs με την OpenMP έκδοση για CPUs που υλοποιήθηκε στην προηγούμενη άσκηση. Η κατάλληλη επιλογή των παραμέτρων για τις εκδόσεις των GPUs, καθώς και τα μεγέθη και σχήματα των μπλοκ νημάτων αφήνονται ελεύθερα για δικό σας πειραματισμό. Επιλέξατε αυτά που σας δίνουν την καλύτερη επίδοση και προσπαθήστε να εξηγήσετε τους λόγους πίσω από τυχόν διαφορές επίδοσης που παρατηρήσατε.

Το μηχάνημα στο οποίο θα εκτελέσετε τα πειράματά σας αποτελείται από δύο επεξεργαστές Intel Xeon X5650 (6 πυρήνες + H/T, συνολικά 12 πυρήνες + H/T) και μία κάρτα γραφικών NVIDIA Tesla M2050 αρχιτεκτονικής Fermi.

4.2 Μεταγλώττιση

Στο Makefile που σας δίνετε, η προεπιλεγμένη ρύθμιση για την μεταγλώττιση είναι σε λειτουργία debug (όλες οι βελτιστοποιήσεις του μεταγλωττιστή απενεργοποιημένες). Για να μετρήσετε τις επιδόσεις των πυρήνων θα πρέπει να απενεργοποιήσετε αυτή την λειτουργία και να μεταγλωττίσετε εξαρχής (“`make clean`”) τον κώδικά σας με “`make DEBUG=0`”.

Προσοχή: Όταν απενεργοποιείτε την λειτουργία debug, απενεργοποιείται και ο έλεγχος του αποτελέσματος του πυρήνα για λόγους συντομίας. Επομένως, προτού τρέξετε την τελική έκδοση, θα πρέπει να έχετε σιγουρευτεί ότι ο πυρήνας σας έχει το σωστό αποτέλεσμα.

4.3 Χρήση υπολογιστών εργαστηρίου

Η χρήση των υπολογιστών του εργαστηρίου (ουρά clones) για την εκτέλεση της άσκησης θα γίνεται μέσω του συστήματος υποβολής εργασιών Torque, όπως και στις προηγούμενες ασκήσεις. Για την εκτέλεση των μετρήσεων σε πραγματική GPU, θα πρέπει να δεσμεύσετε το κατάλληλο μηχάνημα από το σύστημα Torque ως εξής:

```
$ qsub -q termis -l nodes=1:ppn=24:cuda myjob.sh
```