Acceleration of a bioinformatics application using high-level synthesis

Naeem Abbas

THÈSE / ENS CACHAN - BRETAGNE

sous le sceau de l'Université européenne de Bretagne

pour obtenir le titre de

DOCTEUR DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

Mention : Informatique

École doctorale MATISSE

présentée par

Naeem Abbas

Préparée à l'Unité Mixte de Recherche 6074
Institut de recherche en informatique
et systèmes aléatoires

Acceleration of a Bioinformatics Application using High-Level Synthesis

Thèse soutenue le 22 mai 2012
devant le jury composé de :

Philippe COUSSY,
Maître de conférences - Université de Bretagne Sud / *rapporteur*

Florent DE DINECHIN,
Maître de conférences - ENS Lyon / *rapporteur*

Rumen ANDONOV,
Professeur des universités - Université de Rennes 1 / *examinateur*

Tanguy RISSET,
Professeur des universités - INSA de Lyon / *examinateur*

Steven DERRIEN,
Maître de conférences - Université de Rennes 1 / *directeur de thèse*

Patrice QUINTON,
Professeur des universités - ENS Cachan-Bretagne / *directeur de thèse*

Résumé

Les avancées dans le domaine de la bioinformatique ont ouvert de nouveaux horizons pour la recherche en biologie et en pharmacologie. Les machines comme les algorithmes utilisées aujourd'hui ne sont cependant plus en mesure de répondre à la demande exponentiellement croissante en puissance de calcul. Il existe donc un besoin pour des plate-formes de calculs spécialisées pour ce types de traitement, qui sauraient tirer partie de l'ensemble des technologie de calcul parallèle actuelles (Grilles, multi-coeurs, GPU, FPGA).

Dans cette thèse nous étudions comment l'utilisation d'outils de synthèse de haut niveau peut aider à la conception d'accélérateurs matériels spécialisés massivement parallèles. Ces outils permettent de réduire considérablement les temps de conception mais ne sont pas conçus pour produire des architectures matérielles massivement parallèles efficaces. Les travaux de cette thèse se sont attachés à dégager des techniques de parallélisation, ainsi que les moyens d'exprimer efficacement ce parallélisme, pour des outils de type HLS.

Nous avons appliqué ces résultats à une application de bioinformatique connue sous le nom de HMMER. Cet algorithme qui pourrait être un bon candidat à une accélération matérielle est très délicat à paralléliser. Nous avons proposé un schéma d'exécution parallèle original, basé sur une réécriture mathématique de l'algorithme, qui a été suivi par une exploration des schéma d'exécution matériels possible sur FPGA. Ce résultat à ensuite donnée lieu à une mise en œuvre sur un accélérateur matériel et a démontré des facteurs d'accélération encourageants.

Les travaux démontre également la pertinence des outils de HLS pour la conception d'accélérateur matériel pour le calcul haute performance en Bioinformatique, à la fois pour réduire les temps de conception, mais aussi pour obtenir des architectures plus efficaces et plus facilement reciblables d'un plateforme à une autre.

Abstract

The revolutionary advancements in the field of bioinformatics have opened new horizons in biological and pharmaceutical research. However, the existing bioinformatics tools are unable to meet the computational demands, due to the recent exponential growth in biological data. So there is a dire need to build future bioinformatics platforms incorporating modern parallel computation techniques.

In this work, we investigate FPGA based acceleration of these applications, using High-Level Synthesis. High-Level Synthesis tools enable automatic translation of abstract specifications to the hardware design, considerably reducing the design efforts. However, the generation of an efficient hardware using these tools is often a challenge for the designers. Our research effort encompasses an exploration of the techniques and practices, that can lead to the generation of an efficient design from these high-level synthesis tools.

We illustrate our methodology by accelerating a widely used application -- HMMER -- in bioinformatics community. HMMER is well-known for its compute-intensive kernels and data dependencies that lead to a sequential execution. We propose an original parallelization scheme based on rewriting of its mathematical formulation, followed by an in-depth exploration of hardware mapping techniques of these kernels, and finally show on-board acceleration results.

Our research work demonstrates designing flexible hardware accelerators for bioinformatics applications, using design methodologies which are more efficient than the traditional ones, and where resulting designs are scalable enough to meet the future requirements.

N° d'ordre :

École normale supérieure de Cachan - Antenne de Bretagne

Campus de Ker Lann - Avenue Robert Schuman - 35170 BRUZ

Tél : +33(0)2 99 05 93 00 - Fax : +33(0)2 99 05 93 29

Contents

1

Introduction

1.1 High Performance Computing for Bioinformatics

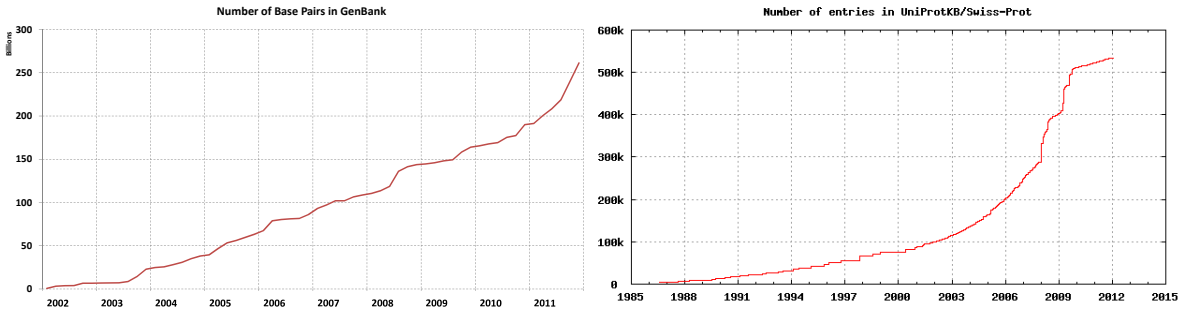
La Bioinformatique est un domaine récent, mais qui suscite depuis une dizaine d'année de plus en plus d'intérêt dans la communauté scientifique. Ce domaine recouvre des champs disciplinaires très variés incluant la biologie, la génétique, l'informatique mais également les mathématiques. L'objectif premier de la bioinformatique est d'offrir aux biologistes des outils informatiques qui leur permettront d'analyser des données issues de séquences génétiques (par exemple de l'ADN, de l'ARN et/ou des protéines) afin d'essayer de découvrir ou de prédire les fonctions biologiques associées à ces séquences.

Les problématiques de la bio-informatique sont nombreuses, parmi celles-ci on peut citer la découverte de gènes dans des séquences d'ADN, la prédiction (et la classification) de la structure et des fonctions de protéines ainsi que la construction automatique d'arbres phylogéniques en vue de l'étude des relations évolutives.

En outre, cette dernière décennie a vue l'apparition de techniques de séquençage d'ADN à haut débit, qui ont permises de grandes avancées (séquençage complet du génome humain [VAM+01], projet d'annotation du génome des plantes [SRV+07]). Ces progrès se sont à leur tour traduits par une explosion du volume de données génomiques (ADN, proteines) disponibles pour la communauté, comme l'illustre la figure 2.1, qui montre l'évolution des banques NCBI GenBank [NCB11] (ADN) UniProt [INT] (proteines).

Il est à noter que les nouvelles générations de technologies de séquençage, facilitent encore plus l'extraction d'énormes quantités de séquences, et vont certainement accentuer cette croissance exponentielle.

Les chercheurs sont de fait désormais confrontés à un défi majeur : extraire de ces volumes de données gigantesques des informations utiles à la compréhension de phénomènes biologiques. Les outils traditionnellement utilisés par la communauté bioinformatique ne sont en effet pas conçus pour fonctionner sur de telles masses de données, et les volumes de calculs mis en jeux dans ces outils d'analyses sont devenus trop importants au point de devenir un goulot d'étranglement.



(a) (b)

Figure 1.1: The exponential growth of the (a) GenBank and (b) UniPortKB databases [NCB11, **?**].

De nombreux travaux se sont donc intéressés à l'utilisation de machines parallèles pour réduire ces temps de calcul. Si les premier travaux ciblaient essentiellement des architectures de super-calculateurs classiques [SRG03, YHK09, CCSV04, GCBT10] (grilles, clusters), la démocratisation des architectures multi-cœurs [Edd, LBP+08] et l'émergence du GPGPU[1] ont rendu ces travaux plus populaires. Outre ces travaux portant sur des architecture généralistes programmables, il faut également mentionner l'utilisation d'accélérateur matériels spécialisés à base de logique programmable [HMS+07, SKD06, DQ07, LST] qui a démontré qu'il était possible de profiter de capacités d'accélération très élevées pour tout en restant à des niveaux de consommation électriques et donc des coûts de maintenance très raisonnables.

L'augmentation de la densité et de la vitesse des circuits FPGA a ainsi favorisé l'émergence d'accélérateurs matériels reconfigurables orientés vers le domaine du calcul haute performance (HPC), avec des applications en calculs financier [ZLH+05, WV08], simulations météorologiques [AT01], traitements vidéo [LSK+05] mais également en bioinformatique[DQ07, SKD06].

Les accélérateurs FPGAs se sont ainsi avérés être des architectures matérielles bien adaptées à la mise en œuvre de traitements de type bioinformatique. Ceux-ci offrent souvent la possibilité d'exposer un un niveau important de parallélisme à grain fin dans l'algorithme, lequel peut ensuite être exploité très efficacement par une mise en œuvre sur FPGA. Une part importante des algorithmes de bioinformatique repose en effet sur l'utilisation de techniques à base de programmation dynamique, en autre pour la comparaison de séquence (Smith-Waterman [SW81], Needleman-Wunsch [NW70] and BLAST [AGM+90]), l'alignement multiple de séquences (CLUSTALW [THG94]), la recherche sur profil (HMMER [Edd]), le repliage de séquences de RNA (MFOLD [Zuk03]) et même la construction d'arbres phylogéniques (PHYLIP [Fel93]). Le caractère *régulier* des traitements effectués dans ces algorithmes se prête ainsi facilement à une parallélisation sur un architecture de type réseau régulier disposant de communication locales.

[1] Qui vise à utiliser les capacités de calculs très importantes des cartes graphiques pour accélérer des calculs scientifiques

1.2 FPGA based Hardware Acceleration

Les circuits FPGAs se présentent comme un gigantesque matrice de cellules logiques programmables, ils peuvent donc être configurés pour implémenter un nombre élevé de chemins de données matériels spécialisés et fonctionnant en parallèle. Les développeurs peuvent ainsi directement implémenter un accélérateur matériel dédié à l'application et tirer parti des gains en performance dus au parallélisme et à la spécialisation.

Dans un FPGA, l'expression de ce parallélisme peut prendre de nombreuses plusieurs formes : parallélisme de tâches en implantant plusieurs cœurs de calculs opérant en parallèle, parallélisme d'opérations au travers de l'utilisation de chemins de données pipelinés complexes. Parce que les FPGAs fonctionnent à des fréquences d'horloges bien plus faibles que les processeurs (en moyenne par un facteur 10), ils doivent compenser leur lenteur relative en exploitant un niveau de parallélisme massif au sein du circuit, tout en s'assurant de la possibilité d'alimenter le circuit en donnée à une cadence suffisante.

Une des techniques utilisées pour améliorer à la fois le degré de parallélisme et la fréquence de fonctionnement des circuits implantés sur le FPGA est d'utiliser des encodages de données à précision réduite (entiers à précision arbitraire, et codage virgule fixe en lieu et place des flottants).

Ici encore les algorithmes de bioinformatique se prêtent très bien à ce genre d'optimisations (par exemple, le codage d'un base ADN peut se faire sur 2 bits au lieu d'un octet complet). Ces caractéristiques en font donc de très bon candidats à une accélération matérielle sur FPGA, en particulier comparé à des machine de type GPUs plus orienté vers le calcul flottant. De nombreux travaux se sont donc intéressés à la mise en uvre, sur FPGA, daccélérateur matériels pour les algorithmes les plus couramment utilisés [HMS+07, SKD06, DQ07].

Ces implémentations, qui ont démontrés des facteurs daccélération très encourageant, se basent sur des spécifications du circuit écrites en VHDL ou Verilog, et très fortement optimisées pour une technologie FPGA donnée. Ce type dapproche pose de fait des problèmes de portabilité, et passer dun accélérateur FPGA à un autre nécessite souvent de reprendre la conception du circuit à zéro. Le section suivante aborde ce problème et discute de la pertinence des outils de synthèse de haut niveau dans ce contexte.

1.3 FPGA Design Flow

Le flot de conception standard pour circuit FPGA se base en grand partie sur celui d'un ASIC. Les principales étapes de ce flot sont représentées dans la Figure 2.2a, elles ne concernent cependant que la partie matérielle d'un co-design logiciel-matériel, le logiciel embarqué étant développé à l'aide de chaînes de compilation classiques.

La première étape de ce flot consiste à définir les spécifications fonctionnelles des composants dans des langages de haut niveau (C, C++, Matlab) afin de déterminer le comportement exact du système. Une fois validée, le concepteur doit définir une

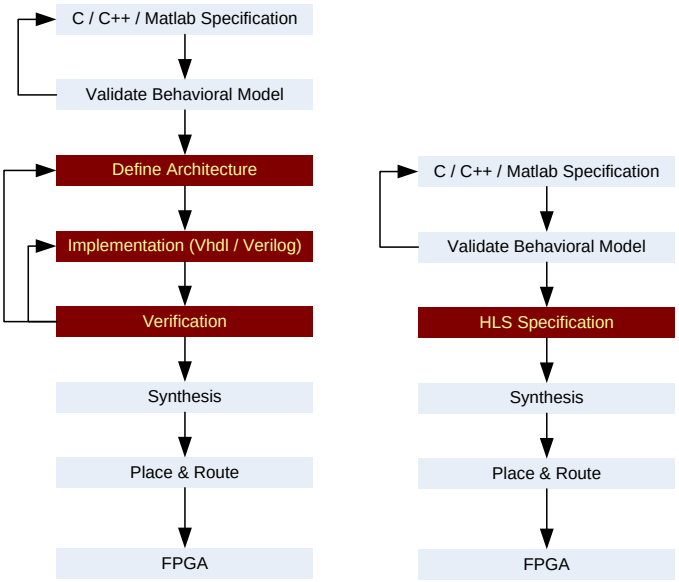


Figure 1.2: Flot de conception FPGA: (a) Flot de conception traditionnel basé sur l'utilisation de langages de description de matériel (HDLs). La description d'une application en HDL est délicate et nécessite un effeort de vérifictaion important. (b) Flot de synthèse basé sur l'utilisation d'outils de synthèse de haut niveau: l'étape de conception manuelle au niveau RTL est remplaée par une description comportementale de haut niveau, suivie d'un phase de généraion automatique de description RTL.

architecture matérielle qui sera en mesure de satisfaire les contraintes de performance, de coût et de consommation électrique imposés par le cahier des charges.

Une fois l'architecture définie, les concepteurs doivent décrire cette architecture au niveau RTL (Register to Logic) à l'aide de langages de description de matériel (Verilog ou VHDL) ou de spécifications schématiques. Cette description est ensuite validée à l'aide de simulations, afin de garantir sa correction.

Une fois vérifiée, la description du circuit est alors synthétisée, c'est-à-dire transformée en une représentation à base de primitives logiques du FPGA ciblé appelée *netlist*. Cette représentation est ensuite placée et routée sur le circuit FPGA ciblé, en permet de dériver un fichier *bitstream* qui servira à configurer le FPGA.

Ce flot de conception reste cependant très complexe et nécessite souvent de nombreuses itérations avant d'obtenir une configuration matérielle opérationnelle.

La première difficulté est de bien choisir la cible architecturale (type de FPGA, capacités de traitement, de mémorisation, etc.), car celle-ci va conditionner une grande partie des choix de conception ultérieurs. Un mauvais choix initial peut ainsi avoir un impact très important sur l'effort de conception global. Le seconde (et principale) difficulté est la spécification au niveau RTL (Register to Logic) de l'architecture de l'accélérateur, qui se fait à l'aide de langage de description matériel tels VHDL ou Verilog. Cette étape est très fastidieuse et nécessite une étape de débogage très longue, avec de nombreuses itérations entre les étapes de spécification et de validation.

La complexité toujours croissante des systèmes électroniques, qui s'illustre par une constante augmentation des fonctionnalités intégrées sur un seul circuit FPGA, rend cette étape de conception RTL de plus en plus critique [CD08]. De fait, les outils de conceptions utilisés pour la mise en œuvre de systèmes de communication sans-fils 4G sont les même que pour le standard GSM, et ce malgré l'énorme écart de complexité entre ces deux standards.

De nombreux travaux se sont donc intéressés à ce problème, en proposant de relever le niveau d'abstraction utilisé la spécification de composants. L'objectif est d'offrir des outils de génération automatique de description RTL à partir de spécification algorithmiques dans des langages de plus haut niveau tel C ou SystemC. On parle alors d'outils de *synthèse de haut niveau*.

1.3.1 Synthèse de haut niveau

Les outils de synthèse de haut niveau (High Level Synthesis) visent principalement à réduire les délais de conception, en utilisant des spécifications de plus haut-niveau que celles offertes par les approches basés sur des descriptions RTL. En plus de réduire le temps de conception à proprement parler, les outils d'HLS permettent également de fortement réduire le temps de vérification, en diminuant le nombre d'itération nécessaire pour obtenir un composant fonctionnel. Par ailleurs en libérant le concepteur de la gestion des horloges, du partage de ressource et de l'interfaçage mémoire, ces outils réduisent également les risques d'erreurs.

Le portage de spécification RTL d'une technologie à une autre se fait souvent au prix d'une baisse des performances et d'une augmentation du coût en ressource et en consommation énergétique [Fin10].

Au contraire, parce que la spécification HLS se fait au niveau fonctionnel, le portage d'une IP matérielle d'une plate-forme à une autre est simplifié, puisque c'est l'outil d'HLS qui va se charger de réaliser le mapping technologique.

Pour autant, les architectures matérielles générées automatiquement à partir d'un niveau de spécification plus abstrait ne sont que rarement aussi efficaces que des implémentations manuelles. En conséquence, les faibles performances obtenues par une utilisation naïve de ces outils limitent l'intérêt des FPGAs dans un contexte de calcul « haute performance ».

Ces faibles performances s'expliquent par l'incapacité de ces outils à extraire un niveau de parallélisme suffisamment élevé. Les accélérateurs matériels issus de ces outils peinent de fait à rivaliser avec des architecture GPU et multi-cœurs, et ce d'autant plus qu'il doivent compter sur une fréquence de fonctionnement plus faible.

Il est possible de lever cette difficulté, en modifiant directement le code source de l'application de manière à faire apparaître un niveau de parallélisme qui sera exploitable par l'outil. Ce type de technique est très efficace dès lors que l'on cherche à accélérer des calculs réguliers, ayant la forme de nids de boucles. En effet, il est possible de d'appuyer sur la grande quantité de travaux issus de la communauté de parallélisation

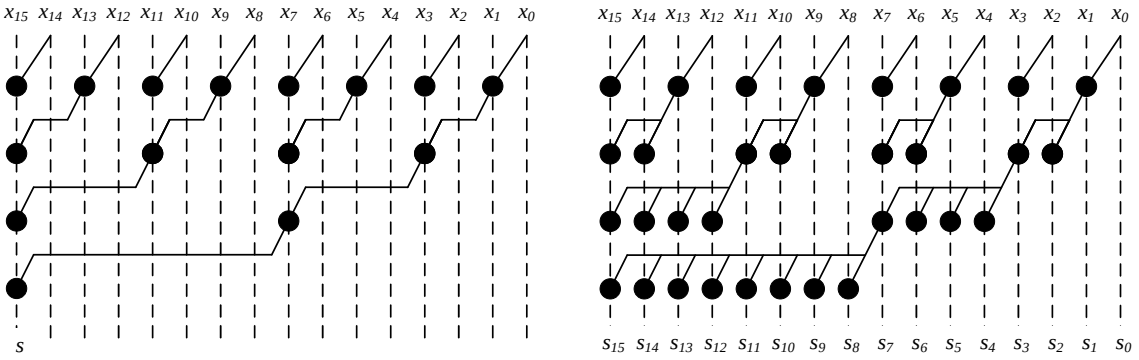


Figure 1.3: Examples of Reduction, (a), and Scan, (b), are shown here, with a possible order of computation.

automatique [Wol90, Wol96].

Outre les aspects liés à la parallélisation des calculs proprement dits, l'obtention de bonne performances nécessite également de prendre en compte de manière très fine la gestion des données dans les différents niveaux de hiérarchie mémoire du système (mémoire hôte, mémoire locale sur la carte, mémoire embarquée). Une des contributions de ce travail est de présenter une revue d'ensemble des transformations clés permettant d'obtenir, grâce à des outils de synthèse de haut niveau, des architectures matérielles spécialisées exploitant efficacement les possibilités des accélérateurs FPGAs actuels.

1.4 Parallélisation à l'aide de réductions et de préfixes parallèles

Les algorithmes élémentaires utilisés en algèbre linéaire peuvent être classés en deux catégories. Dans la première, la taille du résultat d'un calcul est du même ordre que la taille de ces opérandes; c'est par exemple le cas de l'addition de deux vecteurs. Dans la seconde la taille du résultat est plus beaucoup plus petite (en général une valeur scalaire), d'où le terme de réduction proposé par Iverson [Ive62], et qui correspond par exemple à l'opération de sommation des éléments d'un vecteur ou d'une matrice.

Dans ce travail, nous nous sommes intéressés à deux types de calculs : les opérations de *réduction* et les opérations de *préfixes*[2]. Ces opérations, qui opèrent sur des collections d'objets, sont basées sur l'utilisation d'un opérateur élémentaire disposant de propriétés de commutativité et d'associativité.

Soit $\oplus$ le symbole identifiant cet opérateur élémentaire, une réduction sur un vecteur

[2]également connus sous le terme de scan

opérande $(x_1, x_2, \ldots, x_n)$ s'écrit comme:

$$s = \bigoplus_{i=0}^{n} x_i = x_0 \oplus x_1 \oplus \ldots \oplus x_n \tag{1.1}$$

Pour l'opération de préfixe, la taille du résultat est la même que celle de l'opérande, et se définit, pour vecteur opérande $(x_1, x_2, \ldots, x_n)$ et pour un vecteur résultat $(s_1, s_2, \ldots, s_n)$ comme:

$$s_k = \bigoplus_{i=0}^{k} x_i = x_0 \oplus x_1 \oplus \ldots \oplus x_k \tag{1.2}$$

Ces deux types d'opérations sont représentées sur la Figure 2.3 pour $n = 16$. Il est important de remarquer que ces opérations, a priori séquentielles dans leur définition, peuvent être réalisées de manière parallèle en réorganisant les calculs de manière plus ou moins complexe. En particulier, la mise en œuvre efficace d'opérations de type préfixes sur circuits VLSI est un sujet qui a reçu beaucoup d'attention [3], et ce depuis le début des années 60. De nombreuses structures matérielles permettant d'explorer des compromis entre rapidité et coût en surface ont ainsi été proposées [LF80, BK82, KS73, HC87, Skl60].

La mise en œuvre matérielle d'un algorithme utilisant des opérations de préfixes peut profiter de ces résultats, en explorant les différentes possibilités de réaliser le traitement pour choisir la plus efficace. Cette exploration est d'autant plus facile lorsque la conception se fait à haut niveau d'abstraction, par exemple en utilisant des outils de synthèse de haut niveau.

Les algorithmes d'alignement de séquences utilisés en bioinformatique, sont basés sur des algorithmes de programmation dynamique, et exposent des schémas de calcul se prêtant justement assez bien à des reformulations mathématiques permettant de faire ressortir des opérations de réductions et/ou de préfixe.

Dans le chapitre **??**, nous montrons comment certains des traitements mis en jeu dans l'outil HMMER [Edd] peuvent être reformulées comme des opérations de réductions et/ou de préfixes, lesquelles permettent une parallélisation plus efficace.

1.5 Contributions de cette thèse

Le chapitre 3 propose une courte introduction au domaine de la bioinformatique, et à ses enjeux. Nous détaillons en particulier les principaux algorithmes utilisés pour l'alignement la comparaison et le repliement de séquences, en mettant l'accent sur leur coût en termes de traitements et sur leur capacité à passer à l'échelle sur de gros volumes de données. Nous montrons en particulier que la plupart des approches utilisés ne passent pas à l'échelle, et nécessitent de recourir à des architectures matérielle exploitant des niveaux de parallélisme important.

[3] Cet intérêt s'explique par le fait que l'opération d'addition binaire est une opération de préfixe

Le chapitre 4 présente ensuite un survol des techniques et outils de synthèse de haut niveau. Ces outils permettent de dériver une architecture matérielle spécialisée directement à partir d'une spécification algorithmique (par exemple en C). Ils permettent ainsi de réduire de manière drastique les temps de conception. Le chapitre présente les différentes étapes mises en jeu dans un flot de synthèse HLS, et propose un état de l'art des techniques utilisées dans ces outils. Le chapitre se termine par une revue des outils de HLS académiques et commerciaux actuellement disponibles.

Le chapitre 5 s'intéresse quant à lui aux techniques de transformation de code permettant d'améliorer les performances des architectures obtenues par synthèse HLS. Cette partie du manuscrit s'intéresse en particulier aux transformations de boucles pour la parallélisation et à l'optimisation des accès à la mémoire, qui sont des points cruciaux pour l'obtention d'accélérateurs efficaces.

Les chapitres 6 & 7 présentent quant à eux les contributions de ce travail, qui portent sur l'utilisation de transformations de programme complexes, en vue de l'accélération matériel du programme HMMER. Cet outil, très utilisé dans la communauté bioinformatique, repose sur deux noyaux de calculs (`MSV` et `P7Viterbi`) réputés difficiles à accélérer du fait de la présence de dépendances de données qui empêchent a priori toute parallélisation. Dans le chapitre 6, nous présentons l'état de l'art concernant la parallélisation de HMMER sur FPGA et proposons une reformulation des noyaux `MSV` et `P7Viterbi` qui permet de mettre en évidence un niveau important de parallélisme au travers d'opérations de réductions et de préfixes.

Le chapitre 7 s'intéresse quand à lui à la mise en œuvre, sur un accélérateur FPGA et à l'aide d'un outil HLS commercial, d'une architecture de co-processeur parallèle pour HMMER. L'originalité de l'approche vient de l'utilisation d'un schéma de calcul complexe, exploitant du parallélisme à grain fin (boucles vectorisées) et à gros grain (utilisation d'un macro-pipeline de tâche). Ces schémas ont donnés lieu à une mise en œuvre matérielle sur une carte FPGA (XtremeData), et nous a permis de démontrer des facteur d'accélération intéressant par rapport à une mise en œuvre optimisée exploitant de manière très fine les extension SIMD des processeurs multi-cœurs Intel.

2

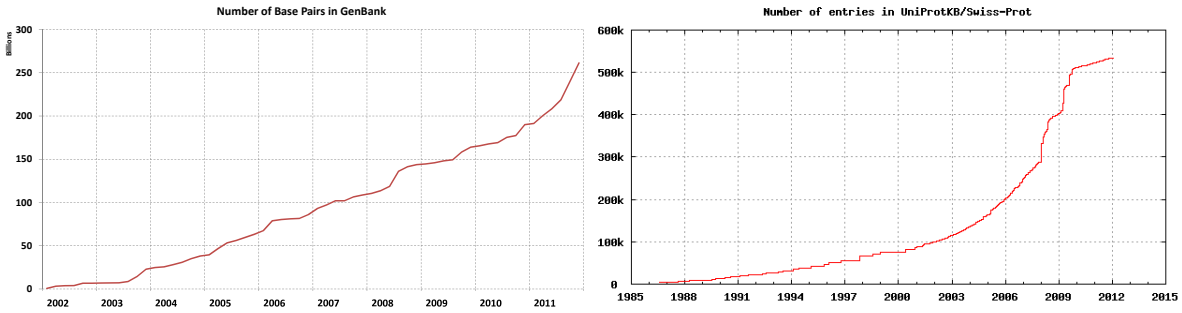
Introduction

2.1 High Performance Computing for Bioinformatics

Bioinformatics can be defined as an application of concepts from computer science, mathematics and statistics to analyze biological data (e.g. DNA, RNA and Proteins) and to predict their the functions and structures. The typical problems found in bioinformatics consist in finding genes in DNA sequences, analyzing new proteins, aligning similar proteins into families and generating phylogenetic trees to expose evolutionary relationships.

In the last decade, there has been a rapid growth in the amount of available digital biological data with the advancement in DNA sequencing techniques, and particularly the success of projects such as The Human Genome Project [VAM+01] and genome annotation projects for plants [SRV+07]. The noticeable examples are the growth of DNA sequence information in NCBI's GenBank [NCB11] database and the growth of protein sequences in the UniProt [**?**] database, as shown in Figure 2.1. Furthermore, the next-generation sequencing technologies have enabled the extraction of genome sequence data in huge quantities, and this will result in further growth of these databases.

Computer scientists and biomedical researchers are now facing a major challenge of transforming this enormous amount of genomic data into biological understanding. The traditional tools and algorithms in bioinformatics were designed to handle very small databases, hence a bottleneck in terms of computational time has arisen when scaled up to facilitate analyses of large data-sets and databases. Recently, a lot of research efforts have been done enabling modern bioinformatics tools to take advantage of parallel computing environments. The implementation of bioinformatic applications on modern multicore general-purpose processors [Edd, LBP+08], General Purpose Graphic Processors (GPGPU) [WBKC09b, VS11, MV08], grid technology [SRG03, YHK09, CCSV04, GCBT10] and reconfigurable platforms, such as field-programmable gate arrays (FPGAs) [HMS+07, SKD06, DQ07, LST] have shown promising acceleration and have significantly reduced the runtime of many biological algorithms while operating on the



(a) (b)

Figure 2.1: The exponential growth of the (a) GenBank and (b) UniPortKB databases [NCB11, **?**].

enormous databases.

The considerable increase in logic density and clock speed of FPGAs, in recent years, have in turn increased the trend of using FPGAs to implement compute intensive algorithms from various domains, including finance [ZLH+05, WV08], weather forecast [AT01], video encoding [LSK+05] and bioinformatics[DQ07, SKD06]. FPGAs are an attractive target architecture for bioinformatics applications, considering their cost-effectiveness as customized accelerators and their ability to exploit the fine-grain parallelism available in many bioinformatics applications. A large class of bioinformatics applications rely on dynamic programming algorithms or a fast approximation of one, including sequence database search programs (Smith-Waterman [SW81], Needleman-Wunsch [NW70] and BLAST [AGM+90]), multiple sequence alignment programs (CLUSTALW [THG94]), profile based search programs (HMMER [Edd]), RNA-folding programs (MFOLD [Zuk03]) and even phylogenetic inference programs (PHYLIP [Fel93]). The FPGA architecture is very well suited for such dynamic programming algorithms, since it has a regular structure, similar to the data dependencies in dynamic programming algorithms, with a communication network to close neighbors.

2.2 FPGA based Hardware Acceleration

FPGAs are simply large fields of programmable gates, so they can be programmed into many parallel hardware execution paths. Due to their parallel nature, different processing operations do not have to compete for the same resources. The designer can map any number of task-specific cores on an FPGA, that all run as simultaneous parallel circuits.

On an FPGA, a designer can exhibit parallelism with the help of a variety of computation granularities (i.e. fine and coarse-grain parallelism), pipelining the long computation paths and through data parallelism. The parallelism granularity may range from very fine-grain computations (e.g. bit-level operations), to fine-grain operations, as in a SIMD architecture (e.g. word- and instruction-level operations) and to coarse-grain computations (e.g. many independent instances of a highly compute intensive kernel,

operating in parallel).

Since FPGAs operate on a very low frequency (about 10 × low) in comparison with a CPU, so in order to outperform the CPU based performance, there should be enough computations to be computed in parallel. Hence compute intensive applications with massive inherent parallelism (e.g. converting each pixel of a color image to grayscale) are highly suitable for FPGA based implementation. Similarly applications with reduced bit-width data are appropriate for FPGAs, due to their ability to compute custom bit-width operations. The majority of bioinformatics algorithms do not require even the full integer precision, thus floating point arithmetic on a modern CPUs will be not valuable. Therefore, FPGA based implementation of such applications can exploit the customizable precision and parallelism, and can result in improved speed and better utilization of the available resources.

The properties held by bioinformatic applications make them viable for FPGA based acceleration in comparison with other acceleration approaches, such as clusters and GPUs. And a lot of research work has been done to accelerate these applications on FPGAs using traditional hardware languages (VHDL and Verilog) [HMS+07, SKD06, DQ07]. The resulting implementations are very efficient and the obtained speedup is highly valuable. However, there are few issues with FPGA based implementations that hinders the designer to opt for an FPGA based implementation, e.g. the design flow is highly error prone and lengthy verification phase often becomes the bottleneck in design projects. In next section, we will highlight these issues by discussing the traditional FPGA design flow and a possible solution to these issues through high-level synthesis.

2.3 FPGA Design Flow

The standard design flow for FPGA designs is borrowed from ASICs, as shown in Figure 2.2a. In practice, a design is usually partitioned into hardware & software parts. The steps shown in Figure 2.2a are related only to the implementation of hardware blocks in such a design, while the software blocks will be implemented using standard software development techniques.

The first step in design flow is to define functional specifications in C, C++, Matlab or any other language in order to validate and fine-tune the desired behavior. Once tested, the designer needs to define an optimal architecture to implement the desired functionality. The architecture selection defines the performance, area and power consumption goals to be met. After the architecture is defined, the design team hand-codes these decisions in the form of a Hardware Description language (Verilog or Vhdl) or in the form of a schematic design. At this stage, functional simulation is carried out to verify the correctness of the described functionality.

After functional verification, the design can be synthesized, i.e. mapping boolean operators on lookup tables (LUTs) modules, shown in Figure 4.1b. The result of logic synthesis is called the *netlist*, a file describing the modules to be used for the

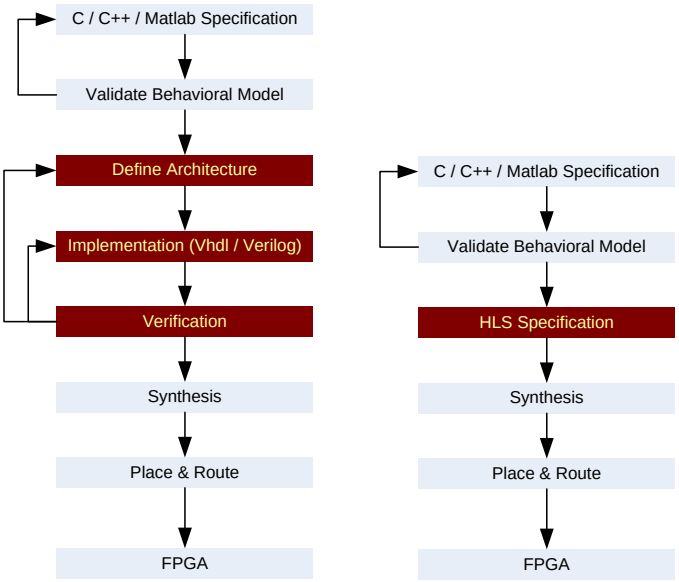


Figure 2.2: FPGA Design Flow: (a) Traditional FPGA design flow using Hardware Description languages (HDLs). The application description in HDL is very error prone and requires a lot of verification efforts. Similarly, it is not easy to port design to other FPGA architectures. (b) High-level Synthesis based FPGA design flow: The manual RTL based design steps are replaced with high-level behavioral description of design following by an automatic generation of RTL design.

implementation of the design and its interconnections. In next step, we place and route the design on FPGA, i.e. the operators (LUTs, Flip-Flops, Multiplexers, etc.) described in the netlist will be now placed on the FPGA fabric and will be connected together through routing. This step is normally done by the CAD tool provided by the FPGA vendor. The CAD tool generates a file called *bitstream*. The bitstream file contains the description of all the bits to be configured, in order to configure LUTs, the interconnect matrices, multiplexers and I/O of the FPGA. Now, by loading the bitstream file on the FPGA, the hardware will be configured according to the functional specifications of the application.

However, the design flow is not that straightforward and often involves a lot of iterative development steps. First problem is to find a suitable architecture, since the following design steps closely related to the selected architecture. An inadequate choice of underlying architecture will prolong the development cycle greatly. The biggest problem in the design flow is the manual RTL description, as when the design is tested after first implementation, bugs are reported and a lot of development time is usually spent in hunting down and fixing the bugs individually. The iterative process of fixing bugs, generating new bugs and fixing them again, prolongs the time-to-market.

One major issue with HDL based implementation is the ever-increasing complexity of electronic designs. The increase in device capacity only exacerbates this issue, as programmers seek to map increasingly complex computations to even larger devices [CD08]. The reality is that we are trying to develop 4G broadband modems and H264 decoders

with tools and methods inherited from era, when GSM and VGA controllers were popular technologies [Fin10]. Eventually, creating RTL design triggers bug and cause the verification phase to be the bottleneck of any ASIC project.

Many research efforts have been done to ameliorate this issue by offering higher-level programming abstractions combined with an automatic RTL generation from popular high-level languages such as C or Matlab, known as *High-level Synthesis* (HLS) tools.

2.3.1 High-level Synthesis

High-level synthesis addresses the root cause of the problem, posed by HDL based design flow, by providing an error-free path from abstract specification to RTL. HLS reduces the implementation time, while also reduces the overall verification effort. The high-level of abstraction needs a lot less detail for the description, and the designer can only focus on describing the desired behavior. With fewer lines of code, when there are no such details as clocks, technology or micro-architecture specifications in side the sources, the risk of errors is greatly reduced. Similarly with fewer blocks to verify, the design can be exhaustively verified.

The abstract functional specifications in HLS, makes the design reuse more effective. Since the design sources are now the abstract specification of the design, retargeting to other architectures is easier. Similarly, the concepts of IP and reuse, which have been promoted to address the design complexity challenge with RTL design, are often unhelpful. The retargeting of legacy RTL is usually done at the expense of power, performance and area [Fin10]. However, in HLS, we are dealing with pure functional specifications and technology specific information is added later by HLS tool automatically. This makes the IP reuse and change in existing functionality, easy to implement and verify.

For biocomputing applications, HLS framework simplify the complex algorithmic description phase and also maximize the design portability. However, the abstract specification of a design may lack several design optimization details, which also expands the hardware mapping possibilities. This can lead to a less efficient design through automatic RTL design generation, in comparison with the efficiency of a highly detailed manual RTL design. Consequently, the resulting performance of HLS based design is often not good enough to justify the use of an FPGA based acceleration. Most of the research efforts in development of these HLS tools, are dedicated to an efficient translation of the given input C code into a hardware design, and this task has been accomplished quite effectively. However, there has been a very little focus on automatic parallelization extraction from the input C code. Therefore, the designer needs to pay a lot of attention on ‘what’ kind of C code will generate ‘what’ kind of circuit.

To tackle this problem, the HLS input needs to be reformed by exposing the hidden parallelism in the algorithm. This task can be accomplished with a prior dependency analysis of input design and based on this analysis parallelism can be expressed with the help of modern high performance compiler optimization techniques [Wol90, Wol96]. The input code should also manage memory resources in an efficient way (i.e. minimizing

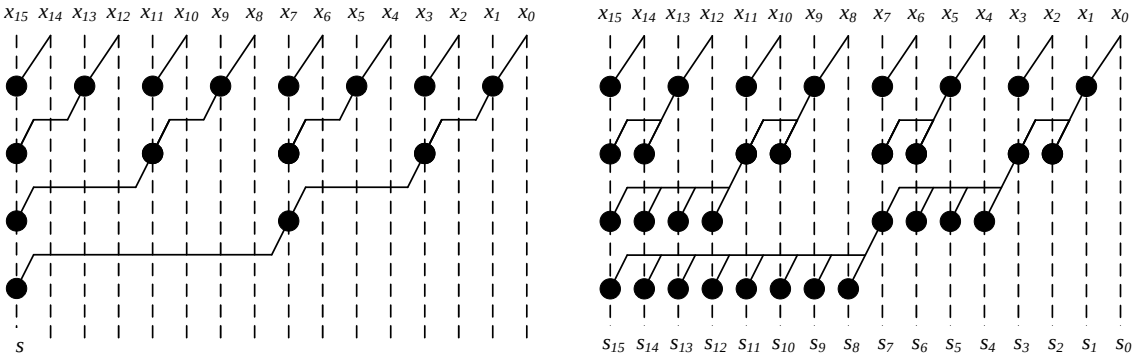


Figure 2.3: Examples of Reduction, (a), and Scan, (b), are shown here, with a possible order of computation.

data communication overhead and maximizing data reuse). Hence, there is a dire need to identify, analyze and layout the rules and guidelines, a designer should keep in mind, while designing for hardware using high-level synthesis tools.

The leitmotiv of this thesis consists in a critical analysis of state of the art HLS tools, identifying their capabilities and shortcomings, formalize techniques to craft an efficient hardware using these tools and exercise these strategies on a well-known, compute-intensive and naively sequential bioinformatic application (i.e. HMMER).

2.4 Exploiting Parallelism with Reductions and Prefixes

The basic algorithms of linear algebra and matrix computation fall into two broad classes. In the first one, the output of a computation is of the same size or bigger than the input data. This is the case, for instance, for vector operations. In the second class, the output is much smaller, typically only one value, than the input data, hence the name reduction which has been coined by Iverson [Ive62].

Here, we are interested in two special kind of such computations, namely *reduction* and *scans* or *prefix* computations, where operations hold associativity and commutative properties. Let say, $\oplus$ represents such an operation, then a reduction can be defined, over a input vector $(x_1, x_2, \ldots, x_n)$, as:

$$s = \bigoplus_{i=0}^{n} x_i = x_0 \oplus x_1 \oplus \ldots \oplus x_n \tag{2.1}$$

A prefix operation belongs to the first class of computations, where output is exactly the same size as the input, and can be defined for an output vector $(s_1, s_2, \ldots, s_n)$ as:

$$s_k = \bigoplus_{i=0}^{k} x_i = x_0 \oplus x_1 \oplus \ldots \oplus x_k \tag{2.2}$$

The operations can be visualized in Figure 2.3 for $n = 16$. The possibility to compute these operations in parallel and in numerous order of executions, has given significant importance to these computations. While targeting FPGA, a designer can easily devise a compromise between the speed and area.

The parallel implementation of prefix networks (Parallel Prefix) has received a wealth of attention from VLSI community going back almost 50 years and various network topologies have been proposed [LF80, BK82, KS73, HC87, Skl60]. These network topologies allow a variety of hardware implementations of a prefix operation, managing various design trade-offs, such as speed, area, wiring and fan-out. Thus, expressing parallelism in the form of prefix operations allows to utilize these previously developed network topologies. Furthermore, the high-level synthesis based implementation of such networks simplifies the design exploration task.

Sequence alignment techniques, based on dynamic programming algorithms, in bioinformatic applications generally compute a best score for a comparison and the computations involved usually hold the above mentioned algebraic properties. So there is a strong tendency that reduction and prefix computations can be detected in these algorithms and it will lead to parallel implementation of the algorithms. In Chapter **??**, we demonstrate how algorithmic dependencies in HMMER [Edd] can be transformed into reductions and prefixes through algorithmic rewriting and which ultimately help to accelerate the execution.

2.5 Contributions of this work

Chapter 3 provides an brief introduction to bioinformatics field and common practices in this field. We highlight some important algorithms for sequence alignment and RNA folding. A review of these algorithms provides a fair insight to the algorithmic complexities and also highlights the challenge being faced by biologists and computer scientists, i.e. exercising these algorithms on constantly growing size of genome databases in becoming time prohibitive. There is a pressing need to utilize the advancements in computation platforms and accelerate bioinformatics applications.

Chapter 4 discusses how bioinformatics applications are viable for FPGA based acceleration. It also reasons the importance of high-level synthesis in FPGA based implementation, in comparison with traditional RTL based designs. The chapter introduces to the design flow inside an HLS tool and discusses the state of the art techniques applied in each step of the design flow. It also provides an overview of few well-known HLS tools in market, investigates their handling of input code and identify the basis of performance degradation.

Chapter 5 is dedicated to design techniques and code transformations, a designer needs to bear in mind while designing hardware from high-level specifications (i.e. C code). The sole idea is to highlight that ‘what’ kind of C code will be translated to ‘what’ kind of hardware, and ‘what’ kind of transformations may help to accomplish design goals

(Speed/Area).

Chapter 6 & 7 presents the research work carried out to accelerate HMMER application by exercising the previously discussed techniques for efficient hardware design using HLS. HMMER is a widely usied tool in bioinformatics for sequence homology searching. The computation kernels of HMMER, namely MSV and P7Viterbi are very compute-intensive, and their data dependencies, if interpreted naively, lead to a purely sequential execution. We propose an original parallelization scheme for HMMER based on rewriting its mathematical formulation, in order to expose hidden potential parallelization opportunities by transforming computations into well-known architectures, i.e.parallel prefix networks & reduction trees. Besides exploring fine-grain parallelization possibilities, we employ and compare coarse-grain parallelization through different system-level implementations of the complete execution pipeline, based on either several independent pipelines or a large aggregated pipeline. We implement our parllelization scheme on FPGA, and then present and compare our speedup with the latest HMMER3 SSE version on a Quad-core Intel Xeon machine. Our results show that a careful HLS based implementation can fairly compete an RTL based design in terms of performance and holds a definite edge in terms of time-to-market and design efforts.

3

An Introduction to Bioinformatics Algorithms

Bioinformatics can be defined as the science of developing computer systems and algorithms for the purpose of spreading up and enhancing biological research [Aga08]. To understand bioinformatics in a meaningful way, it is necessary for a computer scientist to understand some basic biology. This chapter provides a short introduction to those fundamental concepts in biology and highlights some common algorithms being used in bioinformatics.

3.1 DNA, RNA & Proteins:

Cells are the smallest structural unit of life that has all the basic characteristics of a living organism, such as maintaining life and reproducing it [SEQb]. A cell contains all the necessary *information* as well as the required *equipment* to not only produce a replica of itself, but also helps its offspring start functioning [JP04]. Each cell in a human body contains 23 pairs of chromosomes, consisting of 30,000 genes in each of them. There are around 10^{12} cells in a body, which gives an estimate of approximately 3 billion pairs of DNA bases [oEGP08]. Similarly, the plant genome-sequencing project reports more than 40,000 genes in average plants [SRV$^+$07].

The three primary types of molecules studied by biologists are DNA, RNA and proteins. The relationship between these molecules is the transfer of information from DNA to proteins through RNA, as shown in Figure 3.1. DNA encodes RNA that produces the proteins, where proteins are responsible for managing and performing different biological processes inside the cell. A DNA within a cell holds the complete information describing the functionality of the cell. RNA transfers short pieces of this information to different places within the cell, where this information is used to produce proteins [JP04].

DNA is a long molecule forming a chain, where the links of the chain are pieces called

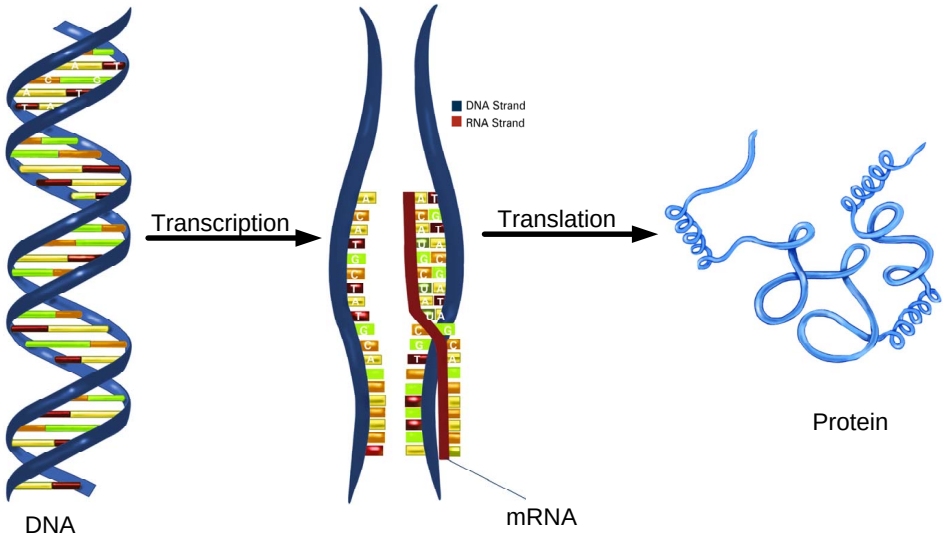


Figure 3.1: The relationship between DNA, RNA and Proteins is refereed as the central dogma of life. [Courtesy of NIGMS Image Gallery [Gal]]

nucleotides, or 'bases', named '`A`', '`C`', '`G`' and '`T`'. DNA encodes the information necessary to build a cell. Most of the cell activities, e.g. breaking down the food as enzymes, building new cell fragments, cell signaling and signal transduction, are carried out by proteins. However, a DNA sequence must be decoded to make a protein and the decoding process requires the creations of an RNA template [Wil03]. The creation of "messenger RNA" or mRNA is called *transcription*, while the process of creating proteins from the mRNA is called *translation*.

The discovery of DNA is probably the most influential discovery of the 20th century, that led to extraordinary breakthroughs in the field of science and medicine. The discovery of DNA has enabled the identification of genes, diagnosing of diseases and developing treatments for them.

Why Bioinformatics?

The information that biologists have collected about gene sequences needs to be processed, in order to completely understand their function and roles, e.g. how a specific gene is related to a specific disease, or what are the functions of thousands of proteins and how proteins can be classified, in accordance to the functionalities. The field of *Bioinformatics* is a collection of such tools and methods that are used to collect, store, analyze and classify this huge amount of biological data.

As mentioned by Thampi [Tha09] regarding the history of bioninformatics, it began in the 1960s with the efforts of Margaret O. Dayhoff, Walter M. Fitch, Russell F. Doolittle and others. Since then it has evolved into a much developed discipline, having strong infulence on modern biology research. In 1970, Saul B. Needleman and Christian D. Wunsch [NW70], proposed the first DNA sequence matching algorithm. However, during the 1990s few major steps brought revolution in bioinformatics study, e.g. the start of

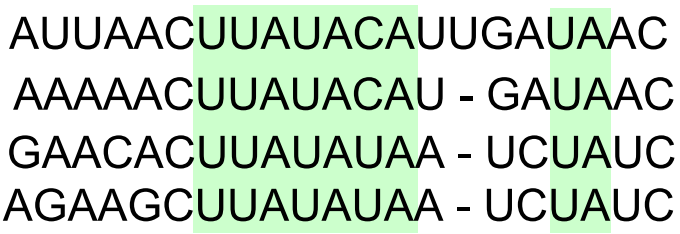


Figure 3.2: An example for multiple sequence alignment: The region of convergence is the shaded part where exact matches are found in all sequences.

Human Genome Project Bioinformatics, the availability of new analysis, services and the availability of data through Internet. Huge databases, such as GenBank and EMBL were designed to store, compare and analyze the biological sequence data that is being produced at an enormous rate. Today, bioinformatics field involves structural and functional analysis of proteins and genes, drug development and pre-clinical and clinical trials [Tha09].

The field of bioinformatic encompasses the use of tools and techniques from three separate disciplines; the source of the data to be analyzed is related to molecular biology, the platform and resources to analyze this data are borrowed from computer science, and the techniques and tools that analyze this data are based on data analysis algorithms [Ric]. The common activities in bioinformatics are hence storing DNA and protein sequences, analyzing, aligning or comparing, classifying protein families and finding new members, predicting structures of RNAs and constructing phylogenetic trees or evolutionary trees. In this chapter, we will focus on algorithms related to general sequence alignments [NW70, SW81, AGM$^+$90, THG94, Edd11a] and RNA folding [NPGK78, ZS81].

3.2 Sequence Alignment

Sequence alignment is an arrangement of two sequences which shows where the two sequences are similar, and where they differ. Sequence alignment techniques are used to discover structural and functional properties of the biological data and characterizing evolutionary relationship in sequences. The identical characters are identified as matches, while nonidentical characters are mentioned as gaps. The regions with identical characters are known as *conserved region*, as shown in Figure 3.2. To discover this information it is important to obtain the "optimal" alignment, which is the one that exhibits the most significant similarities, and the fewer differences.

A similarity between two sequences suggests a similarity in the function or the structure of these sequences. Additionally, strong similarities between two sequences may also show the evolutionary relationship between them, assuming that there might be a common ancestor sequence. The alignment indicates the changes that could have occurred between the two homologous sequences w.r.t. a common ancestor sequence during evolution.

There are two types of sequence alignments: **global alignments** try to align the sequences from end to end for each sequence. Sequences that are similar and that are

approximately the same length are suitable candidates for global alignment. On the contrary **local alignments** search for segments of the two sequences that are similar. Local Alignment does not force the entire sequence into alignment, instead it only aligns the regions with the highest density of matches. It hence generates one or more sub-alignments in the aligned sequences. Local alignments are more suitable for aligning sequences which are different in length, or sequences that have a strong conserved region but not located at same position in both sequences.

In the following section we will show a comparison of both type of alignments and how for the same sequence pair, alignment result can differ.

Sequences are usually either aligned in pairwise manner, i.e. through a *Pairwise sequence alignment*, to compare and identify similarities in two sequences. In some other cases, three or more sequences are aligned, i.e. through a *Multiple sequence alignment*. The latter ones are used to show similarities conserved by most of the sequences and to construct families of these sequences. New members of such families can then be found by searching sequence databases for other sequences exhibiting these same conserved regions.

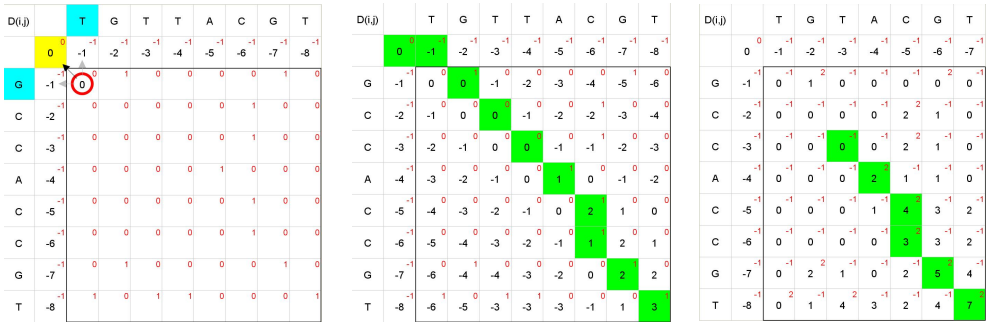
3.2.1 Pairwise Sequence Alignment

Pairwise alignment methods are used to find optimal local or global alignment of two query sequences. The most common methods for pairwise alignment are *dot matrix*, *dynamic programming* and *word* or *k-tuple* methods. The most famous dynamic programming algorithms for pairwise alignment are Smith-Waterman [SW81] and Needleman-Wunsch [NW70] algorithms. BLAST [AGM$^+$90], one of the most widely used bioinformatic tool, is based on a word method.

Needleman-Wunsch: Needleman-Wunsch algorithm performs global alignment for a pair of sequences. The algorithm was proposed in 1970 by Saul B. Needleman and Christian D. Wunsch [NW70], and was the first application of dynamic programming to biological sequence comparison. To find the alignment with the highest score, a two-dimensional array (or matrix) D is allocated. The entry in row i and column j is denoted by $D_{i,j}$. There is one column for each character in sequence A, and one row for each character in sequence B. Each cell of matrix D will be computed using following formula:

$$D_{i,j} = \max \begin{cases} D_{i-1,j-1} + \delta(A_i, B_j) \\ D_{i-1,j} + \delta(A_i, -) \\ D_{i,j-1} + \delta(-, B_j) \end{cases} \tag{3.1}$$

Figure 3.3a shows the initialized matrix and the data dependency, as depicted by the formula above. The numbers in small font, in first row and first column mentions the gap penalty while in rest of the matrix they shows the matching and penalty scores. The matching score, $\delta(A_i, B_j)$ is equal to 1 when A_i and B_j are same characters. Otherwise, the penalty is set to 0 for any mismatch. Figre 3.3b shows the global alignment from



(a) (b) (c)

Figure 3.3: Pair-wise Sequence Alignment: (a) Matrix initialization & computation dependencies , (b) Global alignment with Needleman-Wunsch, (c) Local alignment with Smith-Waterman. The green trail in (b) and (c) shows the alignment. [Figures generated using Basic-Algorithms-of-Bioinformatics Applet [Cas]]

Needleman-Wunsch algorithm. The final alignment for this example:

```
-  G  C  C  A  C  C  G  T
   |           |  |     |  |
T  G  T  T  A  C  -  G  T
```

Smith-Waterman: The Smith-Waterman algorithm, also based on dynamic programming techniques, computes the optimal local alignment of two sequences. Instead of looking at the entire sequence length, the Smith-Waterman algorithm compares only segments (for all possible lengths) of the input sequences and try to optimizes the similarity score. The main difference with Needleman-Wunsch is that Needleman-Wunsch allows negative scoring, whereas Smith-Waterman forces negative values to zero. This choice of positive scoring makes local alignment visible. The Smith-Waterman algoritm computes the matrix D as:

$$D_{i,j} = \max \begin{cases} D_{i-1,j-1} + \delta(A_i, B_j) \\ D_{i-1,j} + \delta(A_i, -) \\ D_{i,j-1} + \delta(-, B_j) \\ 0 \end{cases} \tag{3.2}$$

Figure 3.3c shows the local alignment, where matching score, $\delta(A_i, B_j)$ is set to 2 and all penalties are set to -1. The final alignment in this case is:

```
A  C  C  G  T
|  |     |  |
A  C  -  G  T
```

Local vs. Global Alignment: From the above two alignments, it can be seen that global alignment can align even less conserved regions in comparison with local alignment that only aligns the regions that are well conserved by the two sequences. Similarly, local

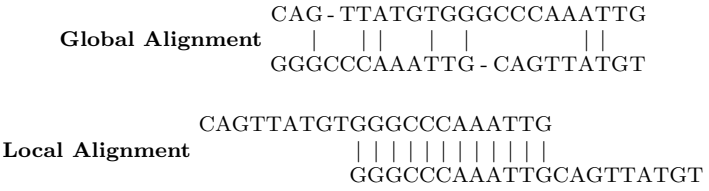


Figure 3.4: Local Alignment aligns very significant regions, apart from the region location in two sequences, While global alignment aligns even small, not very significant, regions.

alignment can align well conserved regions, apart from their location in the two sequences. The example in Figure 3.4 shows how different results can be obtained from global and local alignments. In this example, local alignment aligns the starting region of the one sequence to the end region of the other sequence. On the other hand, global alignment aligns sequences from end to end and the example demonstrates the "gappy" nature of global alignment when sequences are insufficiently similar. Global alignments are most useful when query sequences are similar and of roughly equal size, e.g. protein sequences from the same protein family are often very conserved, and hence have almost the same length [JP04].

A hybrid method, known as "glocal" (short for **glo**bal-lo**cal**), presented by Brudno et al. [BMP$^+$03] attempts to combine features of both kind of alignments. Glocal alignment aligns two sequences by transforming one sequence into the other by a series of operations. The set of supported additional operations are not limited to insertion, deletion and substitution, but also include other possible types of mutations, e.g. inversion (a small segment of the sequence is first removed and then inserted back at the same location but in the opposite direction), translocation (a small segment is removed from one location and inserted into another, without changing the orientation) and duplication (a copy of a segment is inserted into the sequence without making any change to the original segment).

BLAST: The **B**asic **L**ocal **A**lignment **S**earch **T**ool, or BLAST, was developed by Altschul et al. [AGM$^+$90]. This method is widely used from the Web site of the National Center for Biotechnology Information at the National Library of Medicine in Washington, DC (`http://www.ncbi.nlm.nih.gov/BLAST`). The BLAST server is probably the most widely used sequence analysis facility, where alignments can be performed against all currently available sequences. BLAST is fast enough to search an entire database in a reasonable time. Before the development of fast algorithms such as BLAST and FASTA (another k-tuple based tool), database searches were very time consuming, because they had to rely on a full alignment procedure such as Smith-Waterman. However, BLAST algorithm emphasizes on speed rather than sensitivity, in comparison with traditional tools.

BLAST aligns two sequences by first searching for very short identical words (known as *tuples* or *k-mers*) and then by combining these words into an alignment. The length of the word is fixed at 3 for proteins and 11 for nucleic acids. In the first step, the

algorithm creates a word list in the query sequence and then refines the word list to only very significant words, whose possible matching score is higher than a threshold, as shown in Figure 3.5a. Then, BLAST scans the database for the exact match of these high-scoring words, as described in Figure 3.5b. In the third step, these matches are extended in the right and left directions, from the position of the match, as shown in Figure 3.5c. The extension process in each direction stops when the accumulated score ends increasing and is just about to start fall a small amount below the best score found for shorter extensions. This extension phase may find a larger stretch of sequence, known as high-scoring segment pair (HSP), with higher score than the original word. A newer version of BLAST, called BLAST2 [TM99] attempts to accelerate the alignment process by finding word pairs on the same diagonal, which are within distance A from each other, as shown in Figure 3.6. It extends only such word pairs, instead of all words. In order to maintain the alignment sensitivity, BLAST2 lowers down the initial threshold that results in greater number of candidate words. However, since the extension is done only on a few of them, the computation time of overall alignment decreases.

3.2.2 Multiple Sequence Alignment

Given a family of functionally related biological sequences, searching for new homolog sequences in an important application in biocomputing. The new members can be explored using pairwise alignments between family members and sequences from the database. However, this approach may fail to identify distantly related sequences, due to weak similarities to individual family members. A sequence having weak similarities with many family members is likely to belong to the family, but pair-wise matching will be unable to detect it. A solution can be to align the sequence to all family members at once.

Multiple sequence alignment (MSA) is an extension of pairwise alignment, that aligns more than two sequences at a time. Multiple alignment methods try to align all of the sequences in a given query set in order to identify conserved sequence regions across the group of sequences. In proteins, such regions may represent conserved functional or structural domains, thus such alignment can be used to identify and classify protein families.

Computationally, MSA presents several difficult challenges. The optimal alignment of more than two sequences at the same time, considering all possible matches, insertions and deletions, is a difficult problem. Dynamic programming algorithms used for pair-wise alignment, can be extended for MSA, but for aligning n individual sequences of length l, the search space increases exponentially and computational complexity is $O((2l)^n)$ [WP84]. Such algorithms can be used to align 3 sequences, in a cubical score matrix, or a small number of relatively short sequences [Mou04]. Other methods in use for multiple sequence alignment are (1) Progressive alignment [THG94, WP84], (2) Iterative alignment [MFDW98] and (3) Statistical methods [KBM$^+$94, Edd].

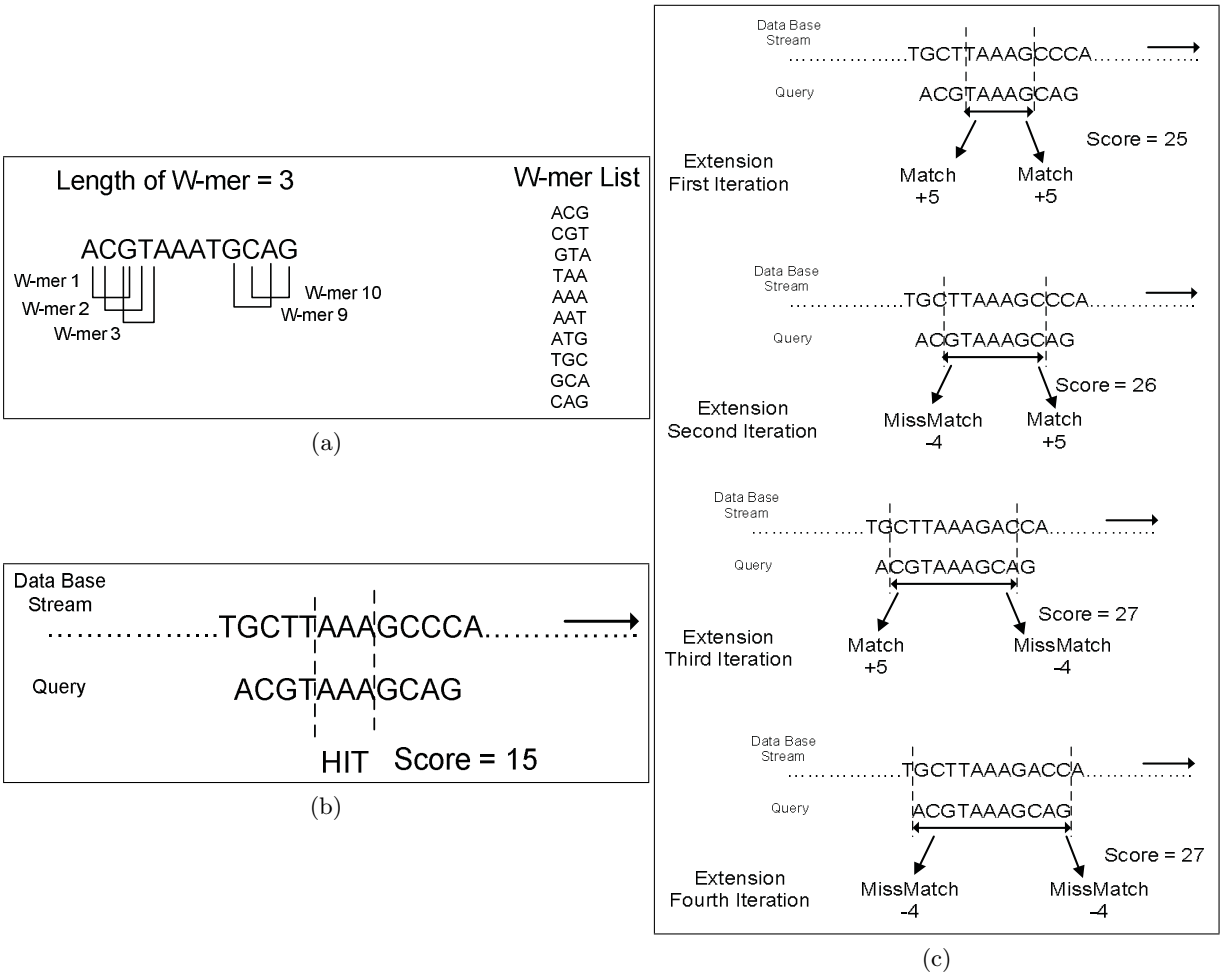


Figure 3.5: A graphical illustration of the BLAST algorithm: (a) In the first step, BLAST creates a list of words from the sequence, (b) In the second step, it searches against the database for exact word matches, (c) Then the third step extends the match in both directions. [Example borrowed from [SKD06]]

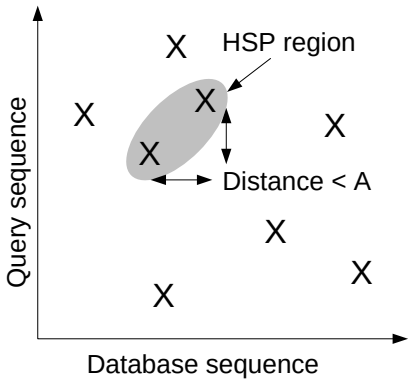


Figure 3.6: BLAST: The X's mark shows the position of the high scoring words. The elliptic region shows the newly joined region

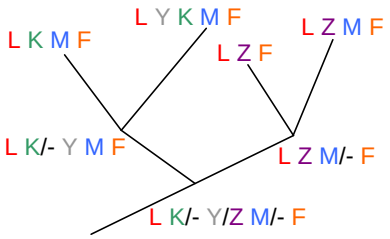


Figure 3.7: Progressive sequence alignment

Progressive Alignment: Progressive alignment techniques based on the dynamic programming method try to build an MSA by first aligning the most similar sequences and then by progressively adding groups of sequences to the initial alignment, reducing the complexity to $O(nl^2)$ [WP84]. Relationships among the sequences are modeled by an evolutionary tree in which the outer branches or leaves are the sequences. The closely related sequences are aligned first and then aligned with other pairs in subsequent tree levels, as shown in Figure 3.7. The most notable program based on progressive methods is CLUSTALW [THG94]. Progressive alignments are not guaranteed to be globally optimal. The major problem is that when distantly related sequences are aligned during the first stage, errors can be made, which may propagate to the final result. A second problem with the progressive alignment method is the choice of suitable scoring matrices and gap penalties that apply to the set of sequences [Mou04].

Iterative Alignment: Iterative alignment methods attempt to correct the key issue of progressive methods, i.e. the fact that errors in the initial alignments propagate through MSA. The problem is addressed by repeatedly realigning subgroups of the sequences and then by aligning these subgroups into a global alignment of all of the sequences. The goal is to improve the overall alignment score [Mou04].

The DIALIGN program [MFDW98], based on an iterative alignment technique, performs MSA through segment-to-segment comparisons rather than residue-to-residue comparisons. Pairs of sequences are aligned by locating aligned regions, i.e. the regions that do not include gaps, called “diagonals”. These diagonals are then used to generate an alignment. The alignment is generated using a greedy method, i.e. the diagonal with the highest weight is selected first and then, the next diagonal from the list is added iteratively to the alignment, if the new diagonal is consistent with the existing alignment, i.e. if there is no conflict due to the double presence of a single residue or cross-over assignments of residues. The algorithm proceeds until the whole list of diagonals has been processed.

Hidden Markov Models (HMMs) [RJ86] are a popular machine learning approach used for sequence homology searching. HMM is a statistical model that take into account all possible combinations of matches, mismatches, and gaps in a set of query sequences, to generate an alignment. HMMs are widely used for finding homologous sequence by comparing a profile-HMM to either a single sequence or a database of sequences. The profile-HMM is first built with prior information about the sequence family and trained

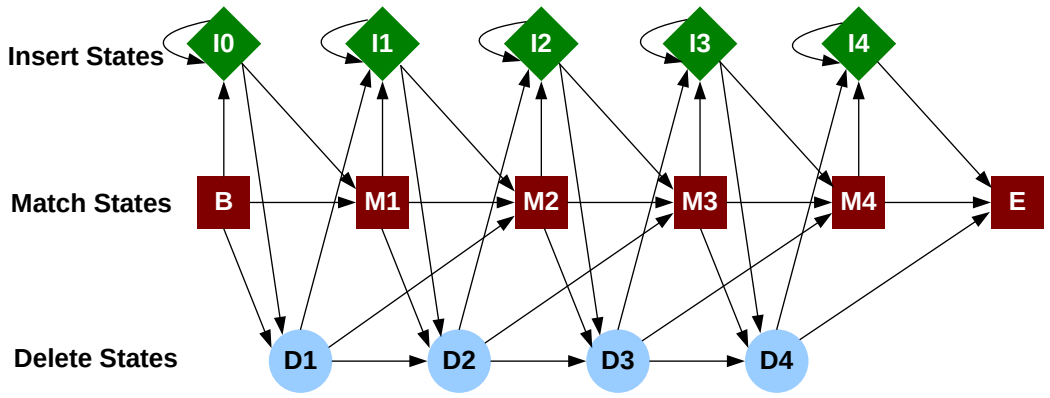


Figure 3.8: Hidden Markov Model for sequence alignment. [KBM+94]

with a set of data sequences. Comparing a query sequence against the profile-HMM allows one to find out if the query sequence is an additional member to the family or not. A different profile HMM is produced for each set of sequences. The intuition behind the profile-based matching is that the multiple alignment of a sequence family reveals regions that are more conserved by the family and the regions that seem to tolerate insertion and deletion more than the conserved ones. Thus position-specific informations must be utilized when searching for homologous sequences. Profile-based methods build position-specific scoring models from multiple alignments, e.g. there will be a higher penalty for insertion/deletion in a conserved region than in a region of tolerance. Krogh et al. [KBM+94] were the first to introduce HMM to computational biology (see Figure 3.8). For each column of the multiple alignment, a 'match' state models the distribution of residues allowed in the column, while 'insert' and 'delete' states allow insertion and deletion of residues between two columns. Profile HMM diverges from standard sequence alignment scoring by including non-affine gap penalty scores. Traditionally, an 'insert' of x residues is scored as $a + b(x - 1)$, where a is the score of first residue and b is the score for each subsequent residues in the insertion. In profile-HMM, insertion of a residue x is modeled using state transitions from state 'match' to 'insert', from state 'insert' to 'insert', and from state 'insert' to 'match'.

Profile Hidden Markov Model Packages: There are several software packages implementing profile HMMs or HMM-like models. SAM [Hug96], HMMER [Edd10], PFTOOLS [BKMH96] and HMMpro [BCHM94] implement models based on the original profile HMMs of Krogh et al [KBM+94]. While PROBE [NLLL97] and BLOCKS [HPH98] assume different models, where alignments consist of one or more ungapped blocks, separated by intervening random sequences blocks. In the next section, we will discuss HMMER tool suite in detail.

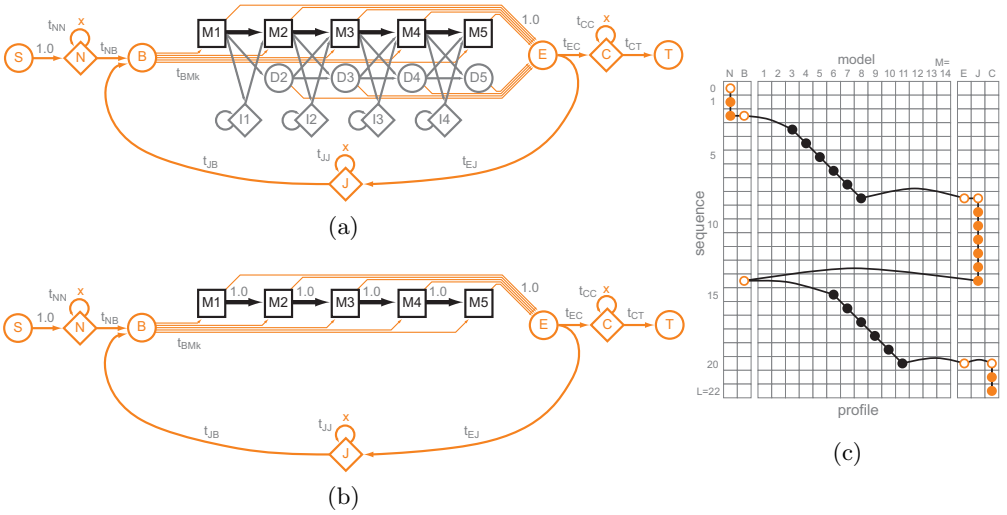


Figure 3.9: HMMER: (a) HMM Plan7 model, (b) MSV filter, (c) Example of MSV path in DP matrix. [Courtesy [Edd11a]]

3.2.3 The HMMER tool suit

One of the most commonly used program for profile-HMM analysis is the open source software suite HMMER, developed at Washington University, St. Louis by Sean Eddy [Edd]. In comparison with other sequence alignment tools (e.g. BLAST and FASTA), HMMER intends to produce more accurate results and is able to detect more distant homologs, because it is based on a probability model instead of heuristic filters. Due to this additional sensitivity, HMMER was previously running about 100x slower than a comparable BLAST search. However, with HMMER3, this tool suite is now essentially as fast as BLAST [Edd10].

Figure 3.9a presents the Plan7 HMM model, used by HMMER. The Plan7 HMM differs from the Krogh et al. HMM model in a few ways. The Plan7 HMM does not have $D \rightarrow I$ and $I \rightarrow D$ transitions, which reduces transitions per node from 9 to 7, one of the origins of the name Plan7. Similarly, a feedback loop from state E, through J to B can be seen, which isn't present in the Krogh et al. profile HMM model. The feedback loop gives HMMER the ability to perform multiple hit alignments. More than one segment per sequence can be aligned to the core section of the model. The self-loop over J provides the separating sequence between two aligned segments.

Figure 3.9c shows how a model can identify two high-scoring alignment segments, due to the presence of the feedback loop. In HMMER, the P7Viterbi kernel solves the Plan7 HMM model through the well known Viterbi dynamic programming algorithm. The P7Viterbi kernel was the most time consuming kernel of HMMER. In order to accelerate the tool suite and to reduce the workload of P7Viterbi, a new heuristic filter (*called Multi ungapped Segment Viterbi*) was designed, that feeds P7Viterbi with the most relevant sequences and filters out the redundant ones. The MSV filter, shown in Figure 3.9c, in an ungapped local alignment version of the P7Viterbi kernel, where delete and insert states

are removed. Figure 3.10 shows the execution pipeline of HMMER3, with the percentage of filtered query data on input of each kernel. It can be seen that the MSV filter handles most of the input query requests, taking approximately 75% of the total execution time. The following section discusses the P7Viterbi and the MSV algorithms in detail.

3.2.3.1 The Viterbi Algorithm

The architecture of the Plan7 model is shown in Figure 3.9a. The M (Matching), I (Insertion) and D (Deletion) states constitute the core section of the model. States B and E are non-emitting states, representing the start and end of the model. The other states (S,N,C,T,J) are called "special states". These "special states" combined with entry and exit probabilities, control some algorithm dependent features of the model. For example, they control the generation of different types of local and multi-hit alignments. The parameters are normally set by user in order to specify an alignment style. The P7Viterbi algorithm follows following equations:

$$M_i[k] = \max \begin{cases} e_M(\text{seq}_i, k) + \max \begin{cases} M_{i-1}[k-1] + \text{TMM}[k] \\ I_{i-1}[k-1] + \text{TIM}[k] \\ D_{i-1}[k-1] + \text{TDM}[k] \\ B_{i-1} + \text{TBM}[k] \end{cases} \\ -\infty \end{cases} \tag{3.3}$$

$$I_i[k] = \max \begin{cases} e_I(\text{seq}_i, k) + \max \begin{cases} M_{i-1}[k] + \text{TMI}[k] \\ I_{i-1}[k] + \text{TII}[k] \end{cases} \\ -\infty \end{cases} \tag{3.4}$$

$$D_i[k] = \max \begin{cases} M_i[k-1] + \text{TMD}[k] \\ D_i[k-1] + \text{TDD}[k] \\ -\infty \end{cases} \tag{3.5}$$

$$E_i = \max \begin{cases} M_i[k] + \text{TME}[k] \\ -\infty \end{cases} \tag{3.6}$$

$$N_i = \max \begin{cases} N_{i-1} + t_{NN} \\ -\infty \end{cases} \tag{3.7}$$

$$J_i = \max \begin{cases} E_i + t_{EJ} \\ J_{i-1} + t_{JJ} \\ -\infty \end{cases} \tag{3.8}$$

$$B_i = \max \begin{cases} N_i + t_{NB} \\ J_i + t_{JB} \\ -\infty \end{cases} \tag{3.9}$$

$$C_i = \max \begin{cases} C_{i-1} + t_{CC} \\ E_i + t_{EC} \\ -\infty \end{cases} \tag{3.10}$$

Variables e_M, e_I, TMM, TIM, TDM, TBM, TMI, TII, TMD, TDD, TME, are the transition memories (e.g. TIM[k] holds the transition value from state I to state M during column k), while t_{NN}, t_{EJ}, t_{JJ}, t_{NB}, t_{JB}, t_{CC}, t_{EC}, are set of constants. In Eq.(3.3) and

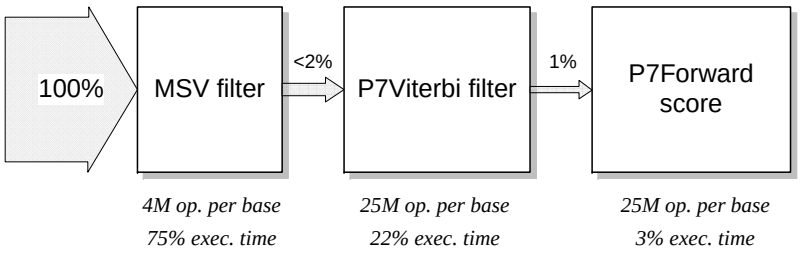


Figure 3.10: HMMER3 execution pipeline, with profiling data

Eq.(3.4), Seq_i represents the current sequence character being aligned.

3.2.3.2 The MSV Kernel

As mentioned earlier, the main computation in the MSV kernel is a dynamic programming algorithm, computing only the *match* state ($M_i[k]$, with i as the column index, and k as the row index) together with boundary and special states. The values are computed iteratively, depending on values computed in previous iteration using the following equations:

$$M_i[k] = e_M(\text{seq}_i, k) + \max \begin{cases} M_{i-1}[k-1] \\ B_{i-1} + t_{\text{BMK}} \end{cases} \qquad (3.11)$$

$$E_i = \max_k \left(M_i[k], -\infty\right) \qquad (3.12)$$

$$J_i = \max\left(J_{i-1} + t_{\text{loop}}, E_i + t_{\text{EJ}}\right) \qquad (3.13)$$

$$C_i = \max\left(C_{i-1} + t_{\text{loop}}, E_i + t_{\text{EC}}\right) \qquad (3.14)$$

$$N_i = \max\left(N_{i-1} + t_{\text{loop}}\right) \qquad (3.15)$$

$$B_i = \max\left(N_i + t_{\text{move}}, J_i + t_{\text{move}}\right) \qquad (3.16)$$

Here, t_{loop}, t_{move}, t_{EJ}, t_{EC} & t_{BMK} are calculated based on the constant size of the current model and the length of the current input sequence.

An MSV score is quite comparable to BLAST's score of one or more ungapped HSPs. Since the MSV score is computed directly by dynamic programming, and not by heuristics used by BLAST (i.e. word hit and hit extension heuristics), it is claimed to be potentially more sensitive than BLAST's approach [Edd11a].

3.2.4 Computational Complexity

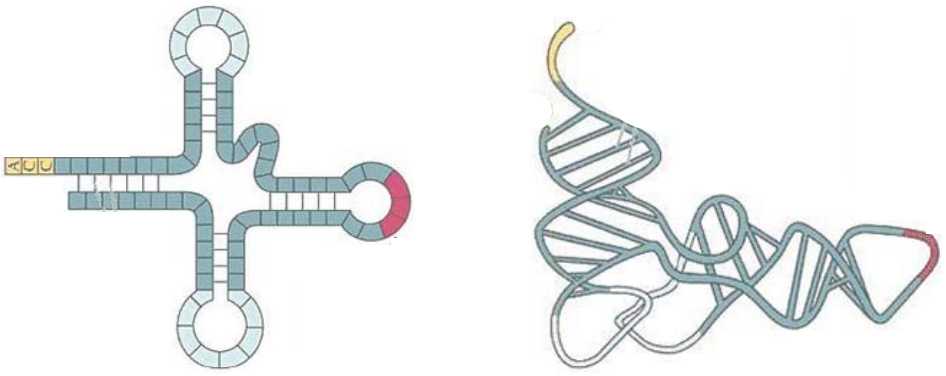
Alignment methods can be compared on the basis of several criteria as shown in Table 3.1. It is interesting to note that most of the global and local sequence alignment methods essentially have the same computational complexity of $O(L_Q L_D)$, where L_Q and L_D are the lengths of the query and database sequences, respectively. Yet despite this, each of the algorithms has very different running times, with BLAST being the fastest and dynamic programming algorithms being the slowest. Using the statistically significant elimination of HSPs and words, BLAST significantly lowers the numbers of segments which need to be extended and thus make the algorithm faster than all the previous algorithms.

Method	**Type**	**Accuracy**	**Search**	**Complexity**
Needleman-Wunsch	Global	Exact	Dynamic Programming	$O(L_Q L_D)$
Smith-Waterman	Local	Exact	Dynamic Programming	$O(L_Q L_D)$
BLAST	Local	Approximate	Heuristic	$O(L_Q L_D)$
HMMER	Multiple	Approximate	Heuristic	$O(L_Q L_D^2)$
ClustalW	Multiple	Approximate	Heuristic	$O(L_Q^2 L_D^2)$

Table 3.1: Comparison of various sequence alignment methods, according to their type, accuracy, search method and the complexity [HAA11].

3.3 RNA Structure Prediction

RNA is a long chain of molecules, although much shorter than DNA. Although an RNA is a linear sequence of bases `A`,`C`,`G` and `U`, they have intra-chain base pairing that produce structures known as secondary and tertiary structures, such as the one shown in Figure 3.11. Tertiary structures determine the biochemical activity of the RNA sequence.



(a) Secondary Structure of a tRNA.

(b) The actual structure of a tRNA is a three-dimensional L shape.

Figure 3.11: An example of RNA secondary structure and its corresponding tertiary structure. [Courtesy [SEQc]]

Investigation of such structures, based on X-ray diffraction or biochemical probes, are extremely costly and time consuming. Thus biologists have simplified the study of complex three-dimensional tertiary structures by focusing attention simply on what base pairs are involved from the secondary structure [ZS84]. A common problem for researchers working with RNA is first to predict the secondary structure of the molecule, in order to analyze the resulting tertiary structure from it.

In order to predict secondary structures, algorithms based on dynamic programming, compute the free energies of the different possible folded structures and attempt to find a structure with minimum free energy. The first algorithm proposed by Nussinov et al. [NPGK78] tries to maximize the number of base pairs in the structure. Later on, Zuker and Stiegler [ZS81] refined this algorithm with a more accurate energy model. In this section we briefly describe these algorithms.

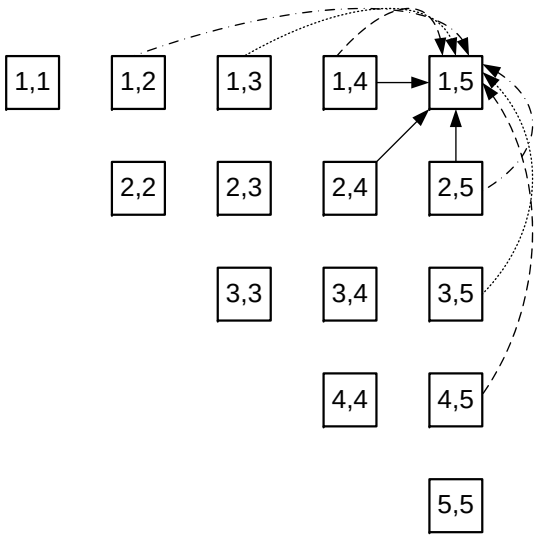


Figure 3.12: Nussinov data dependencies for the computation of $X[1,5]$, where $N = 5$. The cells corresponding to identical arrows are being added using the 4th term of Equation (3.17)

3.3.1 The Nussinov Algorithm

The Nussinov algorithm tries to maximize the number of base pairs in a given RNA sequence. The underlying assumption is that the higher number of base pairs is, the more the structure is stable. For a sequence S with N bases, the Nussinov algorithm attempts to maximize the base pair score $X[i,j]$ in a folded structure of a subsequence $S[i \dots j]$ using following equation, as defined by Jacob et al. [JBC08]:

$$X[i,j] = \max \begin{cases} X[i+1,j] \\ X[i,j-1] \\ X[i+1,j-1] + \delta(i,j) \\ \max\limits_{i<k<j} \{ \; X[i,k] + X[k+1,j] \end{cases} \tag{3.17}$$

where variable X is defined over the domain $1 \leq i \leq j \leq N$. Figure 3.12 shows the data dependencies of the Nussinov algorithm. The score $\delta(i,j)$ can be a constant or a value of the free energy. It allows admissible base pairs to be considered:

$$\delta(i,j) = \begin{cases} 1 & \text{if } (i,j) = (A,U) \text{ or } (C,G) \\ 0 & \text{otherwise} \end{cases} \tag{3.18}$$

After the matrix has been filled, the solution can be recovered by backtracking beginning from the score of the best structure at $X[1,N]$. Sometimes, there are several structures with the same number of base pairs. However, the back tracking algorithm only traces one of the best structures. The overall time complexity of this algorithm is $O(N^3)$ and its space complexity is $O(N^2)$.

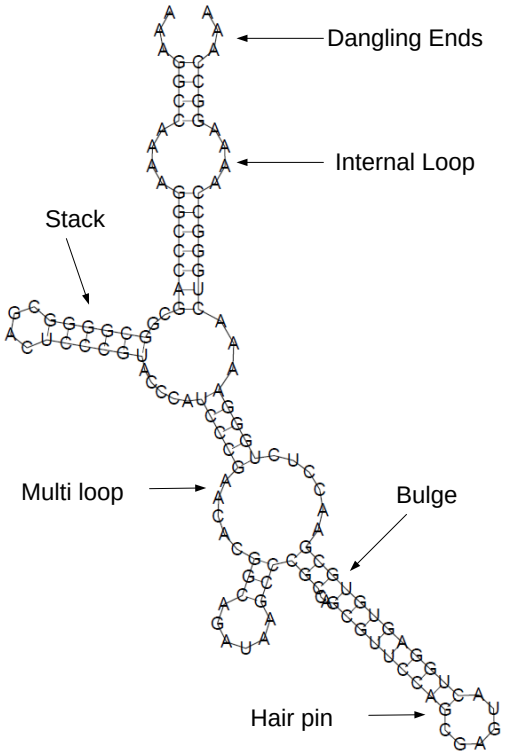


Figure 3.13: An example of an RNA folded into its secondary structure, showing different types of structural features. [Generated with Vienna RNA Websuite [GLB+08]]

3.3.2 The Zuker Algorithm

The Nussinov algorithm does not deal with most of the pseudoknots, the structural models shown in Figure 3.13. Moreover, maximizing the number of base pairs is an overly simplistic criterion, which can not give an accurate prediction.

The Zuker algorithm [ZS81] is also a dynamical programming algorithm and it works on the basis of identifying the globally minimal energy structure for a sequence. The Zuker algorithm is more sophisticated than the Nussinov algorithm, since for every structural element, an individual energy is calculated, which then contributes to the overall energy of the structure.

For an RNA sequence S with N bases, the Zuker algorithm recursively computes three data variables W, V, and VBI according to the following equations, as defined by Jacob et al. [JBC10]:

$$W[i,j] = \min \begin{cases} W[i+1,j] + b \\ W[i,j-1] + b \\ V[i,j] + \delta[S_i, S_j] \\ \min\limits_{i<k<j} \{W[i,k] + W[k+1,j]\} \end{cases} \tag{3.19}$$

$$V[i,j] = \min \begin{cases} \infty \qquad \text{if } (S_i, S_j) \text{ is not a base pair} \\ eh(i,j) \qquad \text{otherwise} \\ V[i+1, j-1] + es(i,j) \\ VBI[i,j] \\ \min\limits_{i<k<j-1} \{W[i+1,k] + W[k+1, j-1] + c\} \end{cases} \tag{3.20}$$

$$VBI[i,j] = \min_{i<i'<j'<j} \{V[i',j'] + ebi(i,j,i',j')\} \tag{3.21}$$

$V[i,j]$ represents the minimum energy of a subsequence $S_{i...j}$, given that S_i and S_j form a base pair. Variable eh represents a hairpin loop, es represents a stack, VBI represents an internal loop (or bulge), and W represents a multi loop, as shown in Figure 3.13. The complexity is the same as for the Nussinov algorithm, namely the time complexity is $O(N^3)$ and the space complexity is $O(n^2)$.

Although the Zuker algorithm is a very powerful framework for RNA secondary structure predictions, it is still only an approximation of RNA folding, which involves a more complicated processes and factors [Sch08].

The common tools in use for RNA secondary structure prediction are Vienna RNA Web server [Hof03], RNAsoft [AAHCH03], pfold [KH03] and MFold [Zuk03].

3.4 High Performance Bioinformatics

The genetic sequence information in the National Center for Biotechnology Information GenBank database has nearly doubled in size every eighteen months, with more than 146 million sequence records as of December 2011 [NCB11]. Performing sequence alignment and homology searching at this large scale is thus becoming prohibitive. For example, Venter et al. [VAM+01] mentions that the assembly of the human genome, from many short segments of sequence data, required approximately 10,000 CPU hours.

Most of the tools currently being used in bioinformatics were not designed to deal with such enormous data sets, but for very small databases of few decades ago. As a result, the tools, which were adequate in past, are very slow and are incapable of a successful analysis. In order to cope with this problem, high performance computing techniques have been experimented for bioinformatic algorithms.

Many solutions have been presented during the last few years, involving a variety of computational frameworks, e.g. grid computing [CCSV04, SRG03, YHK09], cloud computing [QEB+09, MTF08, DTOG+10], SIMD based algorithmic rewriting [Edd11b], and usage of hardware accelerators such as GPUs [MV08, VS11, WBKC09b], FPGAs [HMS+07, SKD06, DQ07] and ASIC [GJL97, LL85]. BIOWIC [SEQa], a workflow for intensive bioinformatics computing, provides access to various acceleration platforms, consisting of clusters, FPGAs and GPUs, through a single framework interface.

3.5 Conclusion

The field of bioinformatics is going through a revolutionary phase. Advancements in bioinformatics improve our understanding of the genes and of their function in diseases. This helps to identify new potential directions in the pharmaceutical industry.

Although the field of bioinformatics is still in developing stage, its importance is evident from the increasingly dependence of modern biology and other related fields on it. Bioinformatics has shown the real potential to lead and play a major role in the future biological research.

However, the traditional analysis tools in bioinformatics are unable to handle the exponential growth in available digital biological data to be processed. This requires an urgent attention from the parallel computing community to develop such platforms for bioinformatics that are fast enough to process the enormous size of biological databases in agreeable time and scalable enough to handle the constant expansion of datasets.

4

HLS Based Acceleration: From C to Circuit

The performance requirements of compute intensive applications, such as bioinformatics and image processing, have increased the demands on computation power. Many acceleration platforms, such as grid computing, cloud computing, GPUs, ASICs and FPGAs, are being actively used. In this research work, we focus on FPGA-based acceleration that has been widely as dedicated accelerators, to satisfy the computation requirements of such applications.

In this chapter, we discuss the characteristics of FPGA based acceleration in general and show how FPGAs are well suited for bioinformatics applications. In section 4.3 we support the idea of High-Level Synthesis based FPGA design, considering the fast and error-free design flow in comparison with the traditional RTL-based design flow. Section 4.4 discusses in detail various steps involved in automatic generation of RTL design from abstract specifications, such as a `C` program.

4.1 Reconfigurable Computing

Reconfigurable computing (RC) has received a lot of attention from the research community, due to its potential to accelerate a wide variety of applications and its ability to fill the gap between hardware and software designs. It is believed that reconfigurable computing is able to achieve potentially much higher performance than a software design by performing computations in hardware, while retaining much of the flexibility of a software solution in comparison with rigid hardware designs such as ASIC (**A**pplication **S**pecific **I**ntegrated **C**ircuits). Reconfigurable devices, including **F**ield-**P**rogrammable **G**ate **A**rrays (FPGAs), consist of an array of computational elements, usually known as logic blocks. The functionality of these logic blocks is programmable through programmable configuration bits. The routing resources that connects these logic

blocks are also programmable. Thus the configurable logic blocks and routing resources, combinedly perform the functionality of the mapped application.

The history of FPGA dates back to the 1970s with the commercial development of programmable logic array (PLA) devices. The first commercially successful FPGA was developed by Xilinx in 1985. Since then, FPGAs have increased considerably in their logic capacity and clock speed. This in turn increased the trend of using FPGAs for high performance computing applications.

FPGAs have been extensively used for ASIC prototyping, which contain fixed hardware configurations and the implementation can not be altered or updated if a bug is found. With their increasing logic density, FPGAs are now also a low-cost alternatives of ASICs, for mapping complex applications. Similarly, with their reprogrammability and massive parallel processing, FPGAs have emerged as a viable alternative for microprocessors. Indeed, FPGAs have brought distinct domains of hardware and software close together and emerged as a platform holding the advantages of both, ASICs and microprocessors [MoH05].

A general comparison between FPGAs and other technologies (ASICs and general purpose CPUs), shows that FPGAs lies as an intermediate option for different design goals, such as speed, power, flexibility and cost.

Speed: FPGAs are inherently parallel and well suited to take advantage of fine grain parallelism. They operate at very low clock frequency, compared to CPUs, but they can perform sometimes tens of thousands of calculations per clock cycle and thus outperform CPUs in speed. The increased logic capacity of FPGAs has also added the ability to accommodate large complex algorithms, which wasn't possible for first generation FPGAs.

In terms of speed, the flexibility of programmable logic always results in a slower implementation and ASIC based design can easily outperform FPGA. It is acknowledge that an ASIC is typically 3 to 10 times faster than an FPGA for the same level of technology [KR07]. So, it can be concluded that generally FPGAs can provide better speed than CPUs, but hardly perform better than ASICs.

Power: CPUs are operating at high clock frequencies (1.5-2GHz), which results in high power dissipation levels. However, because FPGAs operate at low clock speed, they consume very low "tens of watts" of power, while providing a better speed. However, FPGAs can not perform better in power consumption, in comparison with ASICs. FPGAs have been reported to consume 9-12 times more power than ASIC based design [KR07]. The main cause of this large disparity in power consumption is the increased capacitance and the larger number of transistors that must be switched.

Cost: The initial design and production cost of an FPGA unit is much lower than for an ASIC, since the non-recurring engineering (NRE) cost of an ASIC can reach millions of dollars. NRE is the one-time cost corresponding to the design and test of a new chip.

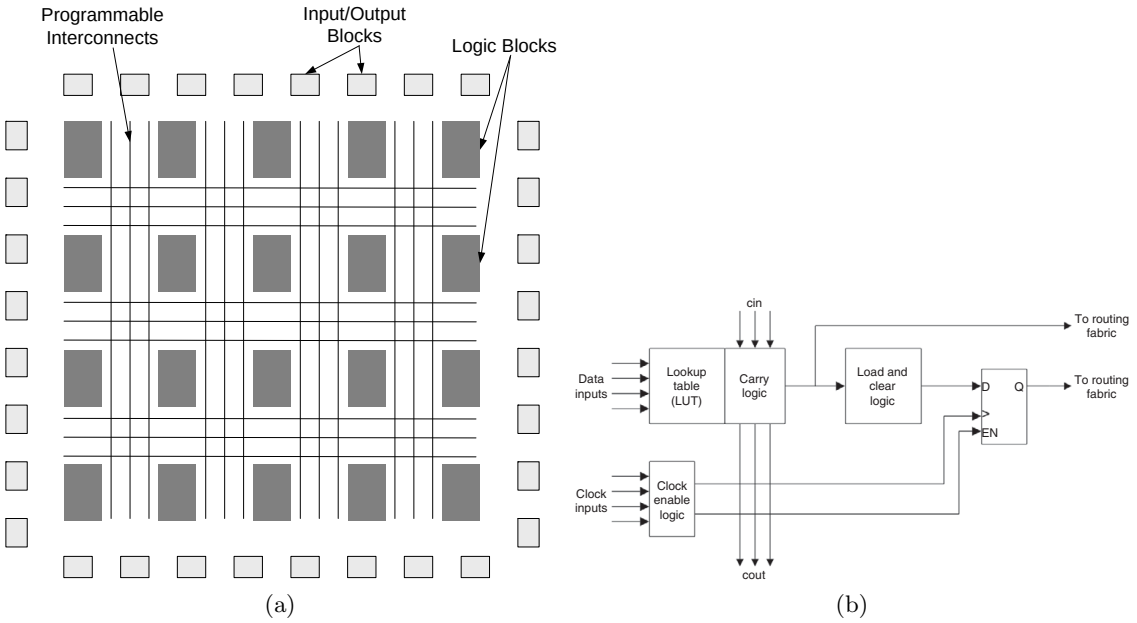


Figure 4.1: (a) An island-style FPGA architecture. The logic blocks (LBs) implements part of application functionality and interconnects connect different LBs to implement the complete logic. (b) A simplified Altera Stratix logic element.

For lower production volumes, FPGAs can be more cost effective than an ASIC design. However, high volume production of custom circuit units can lower cost per unit.

Flexibility: FPGAs are highly flexible due to the reconfigurable nature of the device, contrary to ASIC where the functionality is fixed at the time of production. This is why FPGAs have been extensively used to prototype ASIC design, in order to estimate the performance and more importantly to fix bugs before tape-out. The reconfigurability of an FPGA can also be utilized to run different versions of a hardware architecture for different set of input data sets, as shown in section 7.3.5.

Comparison with multi-core: As the demand for computing resources increases, central processing unit (CPU) development has relied on a combination of an increase in clock speed and change in the micro architecture to improve instruction level parallelism. However, there is a growing performance gap between the capabilities of the microprocessor and the data transfer rate of memory. To fill this gap, techniques such as caching, pipelining, out-of-order execution, and branch prediction have been pushed to their limits. These microarchitectural changes have come at a cost, as running a system with extra circuitry and at a higher frequency consumes more power and dissipates more heat. As a result, CPU designers have moved toward multicore and many-core technologies to improve performance while keeping thermal dissipation within manageable limits. Unfortunately, multi-core architectures have not really addressed the memory bottleneck issue and have introduced a completely new set of problems.

On the other end, FPGAs with reduced power consumption and heat dissipation,

can provide parallel access to on chip memory banks and can address part of the memory bottleneck issue in multi-core systems. Similarly a lot of applications, such as bioinformatic applications for sequence matching do not require the full integer precision available on modern CPUs, so exploiting this fact on FPGA may result in a better utilization of available resources.

4.2 Accelerators for Biocomputing

Sequence homology detection (or sequence alignment) is a pervasive compute operation carried out in almost all bioinformatic sequence analysis applications. However, performing this operation at a large scale is becoming prohibitive due to the exponentially growing sequence databases, doubling about every eighteen months [NCB11]. The bioinformatic applications, e.g. Smith-Waterman (SW), Needleman-Wunsch (NW) and HMMER, are very good candidates for hardware acceleration due to several reasons:

1. **Fine-grain Parallelism:** SW, NW and a simplified version of HMMER (without feedback loop) provide very apparent and significant amount of opportunities for acceleration through fine-grain parallelism, as anti-diagonal cells have no dependency on each other and can be computed in parallel.

2. **Coarse-grain Parallelism:** In practice, SW and NW are often executed by aligning a query sequence against a database. In such scenario, multiple pairwise alignments can be done by performing several independent alignments of input sequence against several sequences from the database in parallel. This coarse grained level of parallelism achieves linear speedup. Similarly, HMMER aligns a database of sequences against a profile and multiple sequences can be aligned in parallel on FPGA.

3. **Bit-width Optimization:** A biological sequence is a set of characters, and on FPGA a character can be represented, and subsequently operated, with reduced number of bits (e.g. 5 bits are sufficient to represent 24 distinct characters in protein sequences). This bit-level optimization enables huge acceleration of alignment computations and permits implementation of more coarse grained nodes.

The above reasons show that these applications are very well suited for FPGA-based acceleration. However, most of these algorithms were originally developed by biologists as software applications, focusing entirely on the accuracy of the results, operating on very small datasets. In order to accelerate these algorithms, a fair amount of efforts is required to first make these algorithms amenable for parallel implementation, and then to map them from software to hardware. A variety of hardware mapping techniques need to be experimented to make sure an efficient utilization of the underlying architecture. Traditionally, FPGA designers would develop a behavioral register transfer level (RTL) description of the required circuit using a hardware description language (HDL), such as

VHDL or Verilog. So for, most of the acceleration efforts of bioinformatic applications has been done at RTL level. Although the results have provided very good speedup, designing at RTL level is laborious. Since the RTL designs are often architecture specific, it is usually hard to retarget such designs. Additionally, at the HDL level, the designs become larger and more complicated and it becomes more difficult to manage this complexity. Any modification in such complex designs is error-prone and may force a very long verification period. To meet these challenges, High Level Synthesis (HLS) tools have been developed to make system-level design easier.

4.3 High Level Synthesis

Logic synthesis is the process of generating circuit implementations from elementary circuit component's descriptions. High-level Synthesis (HLS) refers to circuit synthesis from algorithmic or behavioral description. The input behavioral description is usually a program written in a high level language (HLL), e.g. C, MATLAB, SystemC or any other imperative language. The generated circuit, as output, is an RTL design consisting of a data path and a control unit in HDL.

4.3.1 Advantages of HLS over RTL coding

As mentioned by Michael Fingeroff [Fin10] about RTL, "We are still trying to develop 4G broadband modems with CAD tools whose principles date back from mid-90s, when GSM was a focus of research, or trying to design H264 decoders with languages used to design VGA controllers". The steadily growing design sizes and increasing complexity of applications, amplifies reasons to design with HLS. Here, we discuss a few important motives to adopt HLS over RTL.

High productivity: Design abstraction is generally helpful for the designer in order to control the design complexity and to improve the design productivity. A study from NEC [Wak04] shows that a 1M-gate design typically requires about 300K lines of RTL coding, in comparison with about 40K lines of code in a high-level design specification language. This means the productivity per line of code for behavioral description is about 7.5× thats of RTL coding.

Verification Efforts: With increasing application complexity, the designer's tool should also evolve. Designing modern applications through RTL will eventually trigger bugs and problems will arise during the verification phase. In such a scenario, HLS addresses the root cause of this problem by enabling abstract description of design as an input and providing an error free path from functional specification to RTL. This simplifies the design verification phase.

IP reuse: The RTL designs are targeted to a specific technology and retargeting is usually done by compromising power, performance and area. Even small changes to existing RTL IP to create a derivative can be a tedious job and a complete rewrite might be required for considerable performance [Fin10]. On the other hand, with high-level specification, the design sources become truly generic. The functional specifications in HLS can be easily targeted to available technologies and similarly new functionalities in the existing IP can be added and verified at abstract level.

Architecture Exploration: High-level synthesis tools can provide a quick feedback of performance, area and power for the design, which allow designers to explore different available architectures to finally pick the best one. As estimations are being done at an abstract level, designer can explore different design partitioning opportunities, i.e. with a very low effort, compared to RTL design, designers can explore which parts of the application should be accelerated on hardware and which parts are better suited for a software implementation. High-level synthesis allows the designer to experiment a variety code transformations, individually or usually a combination, to observe their effectiveness on circuit performance criteria. The code transformations range from bit-level to instruction and loop-level transformation (discussed in detail in next chapter).

Instruction-level transformations: Beside simple algebraic transformations, such as *constant folding, constant propagation* and *common subexpression elimination*, High-Level Synthesis tools perform transformations such as *operator strength reduction* and *tree-height reduction*. In the operator strength reduction transformation some operations are replaced with a sequence of less expensive operations in order to reduce the area cost and operation delay. Similarly, height reduction rearranges the order of arithmetic operations by exploiting their commutative, associative and distributive properties. Tree-height reduction (THR) organizes the operations in the form of a tree to reduce the latency of the resulting hardware. In the best case, THR reduces the height from $O(n)$ to $O(\log n)$ where n represents the number of computations in the expression [CD08].

Loop-level transformations : Generally, the most time consuming parts of a program are within loops. The goal of loop transformations is to enable parallel execution, improve data-reuse and data locality to help improving the memory hierarchy, or help the compiler implementing software pipelining. High-level synthesis provides the ability to quickly observe the impact of loop transformations on the resulting hardware design implementation.

4.4 HLS Design Steps

4.4.1 Compilation

The compilation stage transforms the input code to a formal intermediate representation. During this stage, the compiler validates the syntax of the input program, applies syntactic and code transformations e.g. function inlining, dead-code elimination, false data dependency elimination, constant folding & loop transformations [CGMT09] and then transforms the input program into an intermediate representation. The optimizations can be architecture-independent transformations (e.g. elimination of redundant memory accesses) and architecture-dependent transformations (e.g. array data partitioning). The most popular intermediate representations for high-level synthesis are Data flow graphs (DFGs) and control-data flow graphs (CDFGs). However, some HLS tools use additional representations, e.g. SPARK [GGDN04] uses *Hierarchical Task Graphs* (HTGs) [GP92].

4.4.2 Operation Scheduling

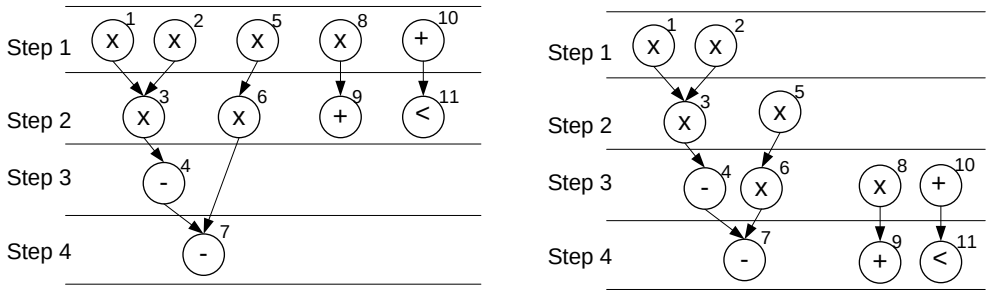
Operation scheduling is one of the most important task in high level synthesis. By scheduling operations in a design, the speed/area tradeoffs can be met, hence an inappropriate scheduling can prevent from exploiting the full potential of the system. The input to a scheduler is a data flow graph (DFG) or control data flow graph(CDFG).

Operation scheduling techniques can be classified in to two categories, namely *resource constrained* and *time constrained*. Given a DFG, a clock speed constraint, a total number of available resources and their associated delays, a resource constrained schedule tries to minimize the number of clock cycles needed to execute the set of operations in DFG. Conversely, a scheduler with timing constraint attempts to minimize the number of resources needed for a given number of clock cycles.

ASAP & ALAP: The simplest scheduling task consist in trying to minimize the latency in the presence of unlimited computing resources. In this scenario, an operation will be scheduled as soon as all of its predecessors have completed, which gives it the name *As Soon As Possible*. ASAP schedule provides the lower bound of the overall application latency.

For a given latency, *As Late As Possible* scheduling tries to schedule an operation at the latest possible time step. The result of such a scheduling provides the upper bound for the starting time of each operation.

The schedules derived from ASAP and ALAP combinedly provide the scheduling range for each operation. Such scheduling range is often used as an initial exploration step, in more advanced scheduling methods [WGK08]. For instance, Hwang et. al. [HLH91] and Lee et. al. [LHL89] restrict the search space for the schedule of each operation using both ASAP and ALAP scheduling and use ILP to minimize the resource cost. In Figure 4.2, ASAP and ALAP schedules define the earliest and latest scheduling steps for each operation, also known as *mobility* of an operation. For instance, operation 5 can be

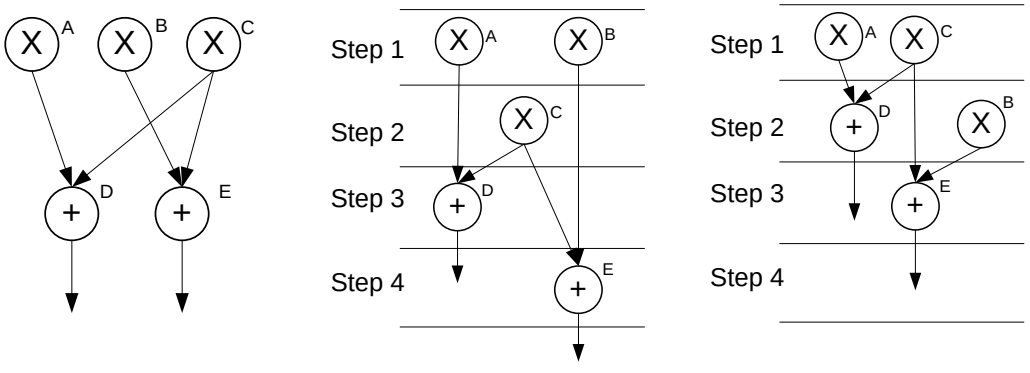


(a) ASAP Scheduling (b) ALAP Scheduling

Figure 4.2: An illustration of ASAP and ALAP scheduling. [example borrowed from [PK87]]

scheduled at step 1 to step 2 and operation 8 & 10 can be scheduled between step 1 to 3, and thus limit the scheduling search space for the following ILP. In list-based scheduling, the mobility information is also used as a priority function for operation scheduling [Gaj92].

List-Based Scheduling: List scheduling is one of the most popular scheduling method for resource constraint scheduling. At each time step, the list scheduling selects and schedules operations from a list of ready operations, based on the priority, as far as resources are available. A ready operation is the one whose all predecessors are scheduled at previous time steps. A ready operation which isn't scheduled due to low priority and resource constraint will be deferred to later time steps. A new scheduled operation may result in changing some other non-ready operations into the ready state and these should be added in the priority list. In absence of resource constraints, list scheduling will generate an ASAP schedule.



(a) DFG (b) List scheduling with mobility (c) List scheduling by [SD02]

Figure 4.3: List Scheduling with different priority functions, with 2 multipliers and 1 adder.

The priority list is sorted according to a priority function, thus the quality of list-based schedule largely depends on its priority function. A common priority function can

be the mobility of an operation, i.e. lower mobility causes a higher priority. However, a scheduler based only on mobility will have no further decision information for operations with equal priority and incorrect ordering may generate a sub-optimal schedule. Sllame and Drabek [SD02] use mobility as the main priority function and then for those operations that have equal priority value the scheduler selects the operations that belong to the same path, using tree-id information collected at a preprocessing phase. Figure 4.3 shows a comparison of both priority functions, where a time step has been saved by giving priority to C, over B, as it belongs to the same path as A. Similarly, Beidas et al. [BMZ11] performed register pressure aware list scheduling. In order to reduce the total number of required registers, they try to minimize the live range cut, i.e. the number of live values after each time step. The live range values are captured through an extended data flow graph and the scheduler tries to minimize the sum of the lengths of live range edges in the DFG. SPARK [GGDN04] uses a priority-based global list scheduling, where the priority function tries to minimize the longest delay within the design. The priorities are assigned to each operation based on their distance from the primary outputs of the design. Hence, the output operations carry a priority of zero and operations whose results are read by output operations have a priority of one and so on.

ILP: Integer Linear Programming (ILP) expresses the scheduling problem through a mathematical description and tries to solve the problem by minimizing or maximizing an objective cost function (e.g. resource or time). For a time constrained schedule, a formulation can be derived as:

$$\min \left\{ \sum_{k=1}^{m} C_{tk} * M_{tk} \right\} \tag{4.1}$$

subject to

$$\sum_{j=E_i}^{L_i} x_{i,j} = 1 \qquad \text{for } 1 \leq i \leq n \tag{4.2}$$

Where M_{tk} is number of resources of type t_k and C_{tk} is the associated cost of the resource unit. The variable $x_{i,j}$ will be 1 only if operation o_i is scheduled at step j else it will be 0, and j is bounded by $E_i \leq j \leq Li$, where E_i and L_i are the earliest and latest scheduling time steps for operation o_i derived from ASAP and ALAP schedules [LHL89]. In order to limit the number of resources to M_{tk}, at each time step, an additional constraint will be:

$$\sum_{i=1}^{n} x_{i,j} \leq M_{tk} \tag{4.3}$$

The data and control dependencies between two operation o_i and o_k, i.e. $o_i \rightarrow o_k$, can

be enforced by:

$$\sum_{j=E_i}^{L_i} (j * x_{i,j}) - \sum_{j=E_k}^{L_k} (j * x_{k,j}) \leq -1 \tag{4.4}$$

The above set of equations defines the scheduling problem and minimizing the Equation 4.1 will lead to the optimal schedule for the given problem. However, the complexity of ILP formulation increases exponentially with the number of time steps, i.e. a unit increase in time step will lead to n additional x variables. This factor rapidly increases the algorithm execution time and limits the applicability of ILP only to very small examples [WGK08].

Force-directed scheduling: The force-directed scheduling (FDS) [PK89] is a popular heuristic for time constraint scheduling. The goal is to reduce the total number of functional units by distributing similar operations into all available time steps. The uniform distribution of resources leads to high utilization and low idle time for each resource. The algorithm is iterative, scheduling one operation at each iteration by balancing the distribution of operations within each time step.

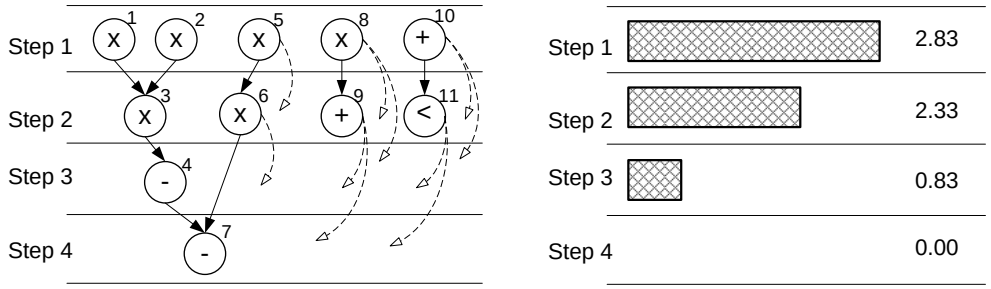
Similar to ILP and list-based scheduling, FDS relies on ASAP and ALAP to determine the time frame of each operation. The time frame is the probability of scheduling an operation in a time step. An operation has a uniform probability of being scheduled into any time step within the mobility period and zero outside this period. This probability will be $1/(L_i - E_i + 1)$, where L_i and E_i are the latest and earliest possible time step for operation o_i respectively, e.g. in Figure 4.2 operation 8 has the probability of $1/3$ for time step 1 to 3 and 0 for step 4.

The next step is to construct the distribution graph by taking the summation of the probabilities of each type of operations for each time step. The distribution graph DG of operation type t_k for step i can be expressed as:

$$DG(i) = \sum_{tk} Prob(opn, i) \tag{4.5}$$

For the example previously discussed in Figure 4.2, Figure 4.4 shows the time frame for each operation and the DG for the multiplication operation. The DG provides an expected operator cost, i.e. the maximum of DG over all i steps. For example, the operator cost for above example is $2.83 \times \text{Cost}_{mul}$. The FDS algorithm attempts to minimize this cost by distributing the probability over all possible states. The algorithm computes the force or impact of scheduling an operation in one of the possible time steps j as:

$$F(j) = \sum_{i=E_i}^{L_i} DG(i) * x(i) \qquad \text{where } E_i \leq j \leq L_i \tag{4.6}$$



(a) The dotted arrows shows the mobility of an operation

(b) The bar graph shows the probability of multiplication operation scheduling for a time step

Figure 4.4: Distribution graphs for multiplication operation

Where $x(i)$ corresponds to a change in the probability by scheduling the operation at step i, e.g. scheduling operation 5 at step 1 will result in $x(1) = 0.5$ and $x(2) = -0.5$. A positive $F(j)$ corresponds to an increase in operation concurrency and negative for a decrease, hence for efficient sharing of functional units across all states, a negative $F(j)$ is desired. For instance, scheduling operation 5 at step 1 results $F(1) = 0.25$ due to higher probability at step 1 in the DG, while scheduling at step 2 will result in $F(2) = -0.75$, which is more effective. FDS schedules an operation based on already computed partial schedule and force associated to the current operation and iterates until all operations are scheduled.

Constraint based scheduling: Constraint Programming (CP) is increasingly being used as a problem-solving tool to solve scheduling problems due to its ability to define precisely the problem in the form of constraints, finding the solution domain and enabling the selection of an optimal solution based on a variety of algorithms [BPN01]. Kuchcinski [Kuc03] used CP over finite domains for scheduling and resource assignment in high-level designs. The prototype constraint solver JaCoP consists of a constraint solver that can find different optimal and suboptimal solutions and optimization algorithms.

According to Kuchcinski, a constraint satisfaction problem (CSP) can be defined as a 3-tuple function $\mathcal{S} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$, where $\mathcal{V}$ is a finite set of variables (FDVs), $\mathcal{D}$ is a finite set of domains (FD) for each FDV, and $\mathcal{C}$ is a set of constraints defining the range of values that can be assigned for each variable. Here, we recall some scheduling function defined by Kuchcinski [Kuc03], in order to show how constraint based scheduling can be useful. For instance, scheduling problem can then be formulated for a partial ordered task graph, by defining FDVs of starting time T, delay D and assigned resource R for each task. The precedence relation can be modeled by inequality constraints:

$$\begin{aligned} &\textbf{for each} (i,j) \in \textbf{Dependencies} \\ &\qquad \textbf{impose } T_i + D_i \leq T_j \end{aligned} \tag{4.7}$$

For instance, in Figure 4.2 the precedence relations can be written as:

$$T_1 + 1 \leq T_3 \qquad T_2 + 1 \leq T_3 \qquad T_5 + 1 \leq T_6 \qquad T_8 + 1 \leq T_9 \tag{4.8}$$

$$T_{10} + 1 \leq T_{11} \qquad T_3 + 1 \leq T_4 \qquad T_4 + 1 \leq T_7 \qquad T_6 + 1 \leq T_7 \tag{4.9}$$

Similarly, a resource sharing constraint can be imposed for tasks whose execution do not overlap. The following constraint will ensure that a resource is not being shared simultaneously:

$$\begin{aligned} &\textbf{for each } (i,j) \text{ where } i \neq j \\ &\qquad \textbf{impose } T_i + D_i \leq T_j \quad \vee \quad T_j + D_j \leq T_i \quad \vee \quad R_i \neq R_j \end{aligned} \tag{4.10}$$

In order to implement the resource constraint, the `cumulative` constraint can be used to express the limit on available resources at any time instance. For the following constraint, AR_i is the amount of required resources by each task and *Limit* is the amount of available resources.

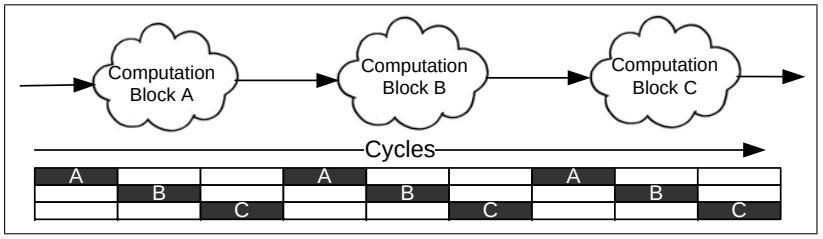
$$\texttt{cumulative}([T_1, \cdots, T_n], [D_1, \cdots, D_n], [AR_1, \cdots, AR_n], Limit) \tag{4.11}$$

Furthermore, we can find an optimal solution from the available solution domain by defining a design goal in terms of a cost function. A cost function can be the schedule length, or the number of resources or power consumption. For example, the schedule length can be defined with the `maximum` constraint.

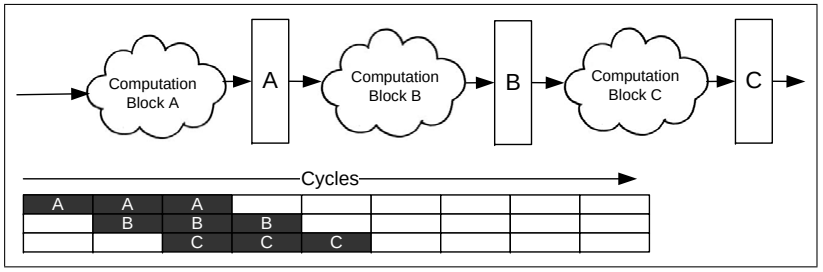
$$\begin{aligned} &\textbf{for each } i \\ &\qquad \textbf{impose } E_i = T_i + D_i \\ &\textbf{impose } \texttt{maximum}(EndTime, [E_1, \cdots, E_n]) \end{aligned} \tag{4.12}$$

By minimizing the variable *EndTime*, the shortest schedule can be found. The above examples show the capability of constraint based schedulers to precisely define the design dependencies, the design constraints and the cost functions for the design goals. The complex scheduling constraints for pipelining and chaining operations can be also be specified.

Pipelining: Pipelining [Lam88] is one of the most effective techniques used to improve the design throughput, when a set of instructions being executed iteratively and not much parallelism is available to execute instructions in parallel. Most of the scheduling techniques, discussed above, can generate schedules for pipelined data flow graphs. Loop pipelining provides a way to increase the throughput of a circuit defined by loop by initiating the following iteration of the loop before the current iteration has completed. The overlapped execution of subsequent iterations takes advantage of the parallelism across loop iterations. The amount of overlap is mentioned as the *Intiation Interval*.



(a) An un pipelined loop. Each loop iteration starts after the completion of previous iteration. It can be noted that only one computation block is active at any clock cycle.



(b) The pipelined architectural view of the loop. The loop iterations are completely overlapped and II=1. The Computation blocks are now separated with storage resources and all computation blocks are now active at each clock cycle, operating on different loop iterations.

Figure 4.5: Loop pipelining impact on resulting architecture, for a loop body with 3 computation blocks or instructions.

Loop pipelining goes through three phases: **prolog**, when new iterations are entering the pipeline and no iteration is completed yet, **kernel**, when the pipeline is in steady state, i.e. new iterations are entering while previous iterations are being completed, and **epilog**, when the pipeline is being flushed and no new iteration is being started. Maximum instruction-level parallelism is obtained during the kernel phase, hence pipelining is most beneficial when most of the execution time is spent in the kernel phase. The *Intiation Interval* is the number of cycles it takes before starting the next iteration, thus an II=1 means a new iteration is started every clock cycle, as shown in Figure 4.5. The demand for computation resources will increase in accordance to the number of overlapping operators and the increase in storage resources will grow with the number of stages in the pipeline.

4.4.3 Allocation & Binding

Allocation defines the type and number of resources needed to satisfy the design constraints. The resources include computational, storage and connectivity components. These components are selected from the RTL component library. The RTL library also includes the characteristics (e.g. area, delay and power) of each component, which are used to construct an early estimation of total area and cost of the design.

The *Binding* stage involves the mapping of the variables and operations in the scheduled CDFG into function, storage and connectivity units. Every operation needs to be bound to a functional unit that can execute the operation. If a variable is used

across several time steps, a storage unit also needs to be bound to hold the data values during the variable lifetime. Finally, a set of connectivity components are required for every data transfer in the schedule.

Scheduling, allocation and binding phases are deeply interrelated and decisions made in one phase impose constraints on the following phases. For instance, if a scheduler decides for a concurrent execution of two operations, then the binding phase will be constrained to allocate separate resources for these operations. Many research works perform the three tasks simultaneously [KAH97, CFH+05, LMD94] and others usually take the scheduled DFG as an input for the allocation and binding phase [RJDL92, SDLS11].

The allocation and binding steps are usually also formulated as an optimization problem, where the main goal is to minimize the number of resources (or the overall resource cost) while fulfilling the area/performance constraints.

ILP: Rim et al. and Landwehr et al. used ILP based formulation to achieve optimal allocation and binding solution [RJDL92, LMD94]. The binding problem can be defined with following constraints:

$$\sum_{j=1}^{\text{Res}} B_{i,j} = 1 \qquad \text{for} \quad 1 \leq i \leq \text{Ops} \tag{4.13}$$

$$\sum_{i=1}^{\text{Ops}} B_{i,j}.S_{i,k} \leq 1 \qquad \text{for} \quad 1 \leq j \leq \text{Res}; 1 \leq k \leq \text{Time} \tag{4.14}$$

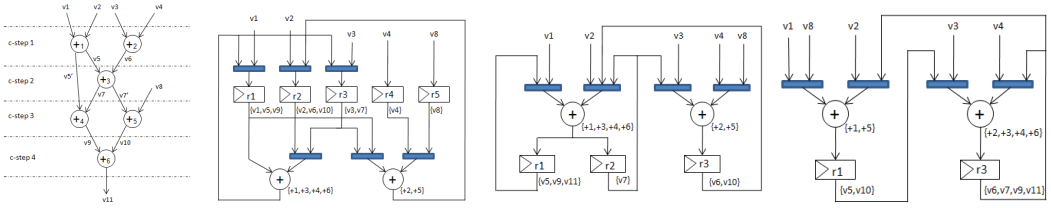
The first constraint ensures that an operation can only be assigned to one resource, while the second constraint impose that at most one operation can be scheduled on a resource during any time step. The ILP-based problem definition can be extended in many ways to include other complex parameters. For instance, in order to reduce the wiring area, the following constraint can be minimized [RJDL92]:

$$\sum_{j_1=1}^{\text{Res}} \sum_{j_2=1}^{\text{Res}} \text{Cost}_{j_1,j_2} \text{Transfer}_{j_1,j_2} \tag{4.15}$$

Where $\text{Transfer}_{j_1,j_2} \in \{0,1\}$ will be 1, if there is a value transfer from resource j_1 to j_2 and Cost_{j_1,j_2} is the associated cost of connecting those resources.

Compatibility Graphs: Allocation and binding can be defined as graph problem as finding cliques in a compatibility graph. A compatibility graph is used to express the resource sharing. Two operations are compatible if they can be executed on same resource and belong to different time steps in the schedule. A compatible graph can be defined as graph $G(V, E)$ where set V represents operations and E is a set of edges representing compatibility among operations.

To solve the binding problem, using a compatibility graph, we have to find a maximal set of compatible operations by formulating a maximal clique partitioning problem, where



(a) (b) (c) (d)

Figure 4.6: Binding Example: (a) Scheduled DFG, (b) Binding results from WBM [HCLH90], (c) Binding results from CPB [KL07], (d) Binding results from [DSLS10]. [Example borrowed from [DSLS10]]

a clique is defined as a subgraph where all nodes are connected to each other. A clique is then said to be maximal if it is not contained by any other clique. An early work on clique partitioning is presented by Tseng et al. [TS86].

More recent results [KL07, DSLS10, SDLS11] on binding algorithms adopt a modified form of compatibility graphs, i.e. *Weighted and ordered compatibility graph (WOCG)*, where an edge $u \xrightarrow{w_{uv}} v$ represents that operations u and v are compatible and u is scheduled earlier than v. The weight w_{uv} represents the strength of the compatibility.

Kim and Liu [KL07] try to reduce the interconnect cost by the reduction of multiplexer inputs. In their work, w_{uv} represents the flow dependency between u and v and how many common inputs the two operations have. This leads to schedule operations holding a dependency or common inputs on same functional unit. The weight is calculated as:

$$W_{uv} = \alpha F_{uv} + N_{uv} + 1 \tag{4.16}$$

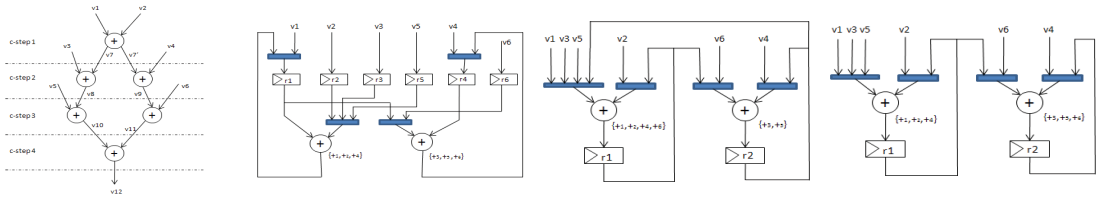
The F_{uv} is a boolean variable which indicates if there is a flow dependency between u and v. N_{uv} is the number of common inputs of the vertices. The coefficient α is used to tune the binding criteria. After generating WOCGs for all FU types, they iteratively search for the longest path in the WOCG, remove operations and associated edges from the graph and finally bind operations inside the path to single FU. Figure 4.6c and 4.7c show the binding results of their algorithm and a comparison with the results of other algorithms.

Dhawan et al. extended this work with a modification in the operation compatibility criteria [DSLS10]. The modified weight function is given as:

$$W_{uv} = \alpha F_{uv} + \beta N_{uv} + \gamma R_{uv} + 1 \tag{4.17}$$

Where F_{uv} and N_{uv} has similar definitions, as above. R_{uv} is a boolean variable that represents the possibility of operation u and v to store their output variable in the same register. R will be 1, if output variable lifetime do not overlap, else it is 0. Figure 4.6d shows the binding results of this algorithm and compare the difference it made with CPB results.

The longest path-based approach proposed by Kim et al. and Dhawan et al. [KL07, DSLS10] reduces the MUX input count, but can result in a design with a few FUs suffering from a large MUX input count. Sinha et al. [SDLS11] proposed an algorithm to divide the number of operations equally among the FUs. This may results in an increase in number of MUXes, but due to a smaller MUX input count, a better critical path delay can be achieved. They used weight relation similar to [DSLS10] but instead of following a path-based approach, they formulated an upper bound for the number of operations for each FU. The upper bound is based on maximum possible delay of FU+MUX, when operations are equally distributed among the available FUs. Figure 4.7 shows a comparison of binding results with existing techniques, discussed earlier. The longest path search binds the last add operation to the left sided FU, which results a MUX with 4 inputs, shown in Figure 4.7c. However, shifting this operation to right sided FU reduces the maximum MUX input count to 3, shown in Figure 4.7d.



(a) (b) (c) (d)

Figure 4.7: Binding Example from [SDLS11]: (a) Scheduled DFG, (b) Binding results from WBM [HCLH90], (c) Binding results from CPB and ECPB [KL07, DSLS10], (d) Binding results from [SDLS11]. [Example borrowed from [SDLS11]]

4.4.4 Generation

During the RTL generation step, the decisions made by previous steps such as scheduling, allocation and binding, are applied in a synthesizeable RTL model. A finite state machine (FSM) implements the scheduling decision and controls which operations are to be executed in which state. The RTL design inside each state can be generated with different levels of binding details. For example, Figure 4.8 shows different types of generated RTL code for an addition operation in `state n` of the FSM. Binding details can be completely omitted, or partially assigned such as mentioning the storage location of each variable, or completely assigned such as binding variables to storage locations and operations to specific operators. The binding tasks that are not performed in generated design, are performed in logic synthesis step that follows HLS.

4.5 High Level Synthesis Tools: An Overview

This section will discuss briefly some state of the art HLS tools available on the market, both from commercial and academic ends.

```
-- Without any binding:
  state(n): a = b + c;
  goto state(n+1);

-- With storage binding:
-- RF represents a storage location
  state(n): RF(1) = RF(2) + RF(3);
  goto state(n+1);

-- With storage and FU binding:
-- RF represents a storage location and ALU is a functional unit
  state(n): RF(1) = ALU1(+, RF(2), RF(3));
  goto state(n+1);
```

Figure 4.8: RTL description written with different binding details. [example borrowed from [CGMT09]]

4.5.1 Impulse C

Impulse C from Impulse Accelerated [HLSa] uses extended version of standard ANSI C language and is based on communication sequential processes (CSP) programming model. An Impulse C program consists of processes and streams, where processes are concurrently operating segments of the application, and data flows from process to process through streams, constructed with FIFOs. A distinct configuration function instantiates instances of all processes and connects them together. Impulse C provides predefined unsigned and signed integer data types for bit widths ranging from 1 to 64. However, the compiler doesn't include any bit-width analysis (discussed in the following chapter) for resource optimizations. Besides ANSI C functions, Impulse C permit to embed custom hardware functions written in HDL languages. Impulse C also provides Platform Support Packages (PSPs) for various target platforms to simplify the creation of mixed software/hardware applications, by providing the necessary hardware/software interfaces for both the hardware (FPGA) and software (microprocessor) elements of the platform [PT05]. Impulse C provides `pragmas` for loop unrolling and pipelining. The limitations involved in Impulse C based hardware design, are:

- **Unrolling:** Loop unrolling is limited to only `for` loop with a constant bound.
- **Pipelining:** pipelining can only be performed to inner-most loop in a loop nest. This can lead to inefficient solutions in cases where there are dependencies that cross multiple iterations at the innermost loop, or if the innermost loop has few iterations. In the latter case, the pipeline will be mainly in a *prolog* or an *epilog* stage throughout the execution time.
- **Partial unrolling:** Partial unrolling and subsequently memory partitioning is not supported, which are one of the most important transformations to exhibit adequate

parallelism, when a complete unroll is not possible due to resource constraints or non-constant loop bounds.

– **Array Configuration:** Impulse C does not support an automatic configuration of local array variables, i.e. an array being accessed twice per cycle will not be automatically configured as dual-port. The designer is suppose to manually configure each array variable with an array configuration function.

– **Non Recursive Memory Access:** When an array is accessed, multiple times, in a pipeline, non-recursively e.g.:

```
for(i=8;i>0;i--){
  A[i+1]=A[i];
}
```

The Impulse C compiler conservatively assumes false dependencies between the array accesses and will not allow parallel read & write operations on a same memory bank, hence will increase the *Iteration Interval* of the pipeline. In order to suppress these dependencies, the designer needs to specifically describe that there will be no *aliasing* of memory accesses, i.e. two addresses that are only known at runtime will not refer to same memory location. Impulse C allows to specify the non-recursive accesses of such memories with the help of a `#pragma`, e.g. `#pragma CO NONRECURSIVE A` will allow parallel access to array `A`. However, if designer specify a memory non-recursive erroneously, the data inside pipeline will be corrupted without any compiler's warning.

– **Loop Transformations:** most of the loop transformations discussed in following chapter have to be implemented manually.

4.5.2 Catapult C

Catapult C developed by Mentor Graphics [HLSb] is one of the most prominent tool in the market and leading market by holding 50% market share in HLS market [HLS09a]. It supports both ANSI C++ and SystemC. Catapult C uses Mentor Graphics Algorithmic C bit accurate data types and it also try to optimize the bit width of variables with bit width defined more than required. Catapult C supports full and partial loop unrolling for all kind of loops, i.e. the inner-most loop or a loop containing a nested loop and partial unrolling for parametrized bounded loops. Similarly, pipelining is not restricted to only inner-most loop, as compared with Impulse C. Loop merging can be also be carried out automatically, if allowed by the designer. Partial loop unrolling combined with memory partitioning (in both block and interleaved manner) can be useful to express parallelism in a scenario where resources are limited or a full unroll is not desired. The designer can derive several solutions for the same input design and Catapult C encourages the designer to iteratively improve the generated design, tuning up different design constraints. However, one of the

limitation observed is that Catapult C takes a lot of time for scheduling very large designs, which hinders the design refinement if performed in an iterative manner.

In terms of loop pipelining, the data dependency analysis in Catapult C is to some extent very conservative, enforcing false memory access dependencies and thus generating a pipeline with low throughput. On the other hand, Catapult C allows nested loop pipelining with unknown loop bounds, but it may generate false schedule, and subsequently corrupt design, problem highlighted by Morvan et al. [MDQ11]. For example, for a loop nest with loop bounds unknown at compile time, e.g.:

```
for(i=0;i<M;i++){
  for(j=0;j<N;j++){
    S[i][j]= func(S[i-1][j]);
  }
}
```

The Catapult C allows to pipeline the complete loop nest with initiation interval of 1. However, for situation where pipeline latency Δ is greater than the inner loop count `N`, if computation `S[i-1][j]` starts at time t, it will be available at $t + \Delta$. While the computation of `S[i][j]` will read this value at $t + N$, where $t + N < t + \Delta$, hence will read the incorrect value. The Catapult C does not generate any guard for such situation, thus may generate a fallacious hardware design.

4.5.3 MMAlpha

MMAlpha is an academic HLS tool developed at IRISA, Rennes. It is aimed at compiling parallel circuits from high level specifications. MMAlpha generates systolic like architectures that are well suited regular computations in signal processing applications. It manipulates and transforms Alpha programs. Alpha is a functional language developed for the synthesis of regular architectures from recurrence equations [LVMQ91]. The transformations implemented in MMAlpha are based on research work on automatic synthesis of systolic arrays by Quinton and Robert [QR91]. Figure 4.9 shows the design flow with MMAlpha. A computation intensive part of original program (usually nested loops) is translated into Alpha program (either manually or using automatic translators), which serves as input to MMAlpha. The initial specification is translated into an internal representation in the form of recurrence equations. Analysis based on polyhedral model is used to check properties of input equations, translated from original loop nests. The translation into Polyhedral model can be helpful to perform most of the loop transformations, discussed in next chapter, to expose parallelism. MMAlpha generates a hardware description at the Register Transfer Level along with a `C` code to interface with the rest of software program. Scheduling is performed by solving an integer linear problem using parametrized integer programming [DRQR08]. MMAlpha also allows automatic derivation of a HDL test-bench for simulation and test purpose [GQRM00].

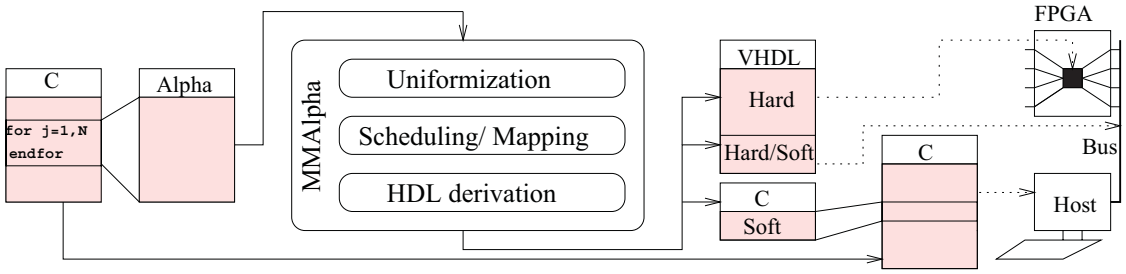


Figure 4.9: Design flow with MMAlpha: Square shaded boxes represent programs of various languages, and round boxes represent transformations. [Courtesy [DR00]]

4.5.4 C2H

C2H [HLS09b] by Altera Corporation is integrated in the development flow of NIOS II embedded software development (EDS) tools. C2H enables creation of a custom hardware accelerator (described as a ANSI-C function) that communicate with the NIOS II processor and other modules through `Avalon` system interconnect fabric and FIFOs. C2H performs somehow direct syntactic translation to hardware and lacks most of the automatic code optimizations. For example, a scalar variable is mapped to a register and an array to a local memory with no memory reorganization. The arithmetic and logical operations are mapped to hardware without resource sharing. Similarly, a very long expression is performed by chaining these resources and C2H does not cut such paths into smaller paths, with intermediate storage. C2H offer loop pipelining that is not limited to only innermost loop, but can also be applied to nested loops. A more detailed review can be found in thesis work of Alexandru Plesco [Ple10].

4.6 Conclusion

In the last decade, FPGAs have greatly prospered in their power of computation, with more computational resources operating at higher speed. This emerged the trend of using FPGAs for high performance computing. Bioinformatics applications are viable for FPGA based acceleration, since they can benefit from fine & coarse grain parallelism on FPGAs. Similarly, bioinformatics applications usually require low bit-width representation, and an FPGA based implementation of such application can benefit from bit-width optimizations. Bit-level optimizations may help to reduce the required resources for a design and may increase the clock speed. Similarly, the reduced resource requirement may help to increase coarse-grain parallelism.

However, the traditional hardware development languages (HDLs) are not efficient for designing large and complex circuit designs of today. HDL based implementations are time consuming, architecture specific and highly error-prone, and finally a laborious verification phase becomes the bottle-neck in the design cycle. High-Level Synthesis tools address these limitations by automating this manual and error-prone design path. The designer provides the abstract design specification and the tool can generate the error-free

HDL design automatically. Since the design specification are abstract and truly generic, it can be easily re-targeted to a different platform and any changes to an existing design are easily manageable. The design description at abstract level can lead to many possible hardware implementations. In next chapter we show how different code transformations can help to generate an efficient hardware design.

5

Efficient Hardware Generation with HLS

The third generation of HLS tools has achieved a reasonable success in comparison with previous generations. One of the several reasons for this success is that tools can take advantage of research into compiler based optimizations rather than relying solely on HDL-driven improvements. Another strong reason is the rise of FPGAs in the same time period [MS09]. High level synthesis targeting FPGA quickly maps an algorithm to hardware and helps enormously to reduce the time-to-market.

Although the current HLS tools deliver a reliable quality of synthesis results, the outcome largely depends on the input functional description. The input language used by most of HLS tools is a variant of the `C` language. An important point here is that `C` is not being used as a programming language, but as a sort of circuit description language. For an efficient hardware generation, the designer needs to understand how an abstract description will be translated into hardware. Here, by efficient hardware generation, we mean an effort to maximize parallelism using minimum resources. As a correct functional description doesn't guarantee an optimal hardware to be found, the designer needs to keep in mind the target hardware.

This chapter will cover most common code transformations, which might help a designer to improve the quality of the synthesis results.

5.1 Bit-Level Transformations

In order to be consistent with RTL data types, many HLS tools provide predefined bit-accurate data types instead of standard HLL data types. In this section we discuss some common bit-level transformations.

```
for(int i=0;i<=31;i++){
  ...
}
```

(a) Standard data type

```
for(unsigned_int<5> i=0;i<=31;i++){
  ...
}
```

(b) Bit-accurate data type

Figure 5.1: Bit-width Narrowing example

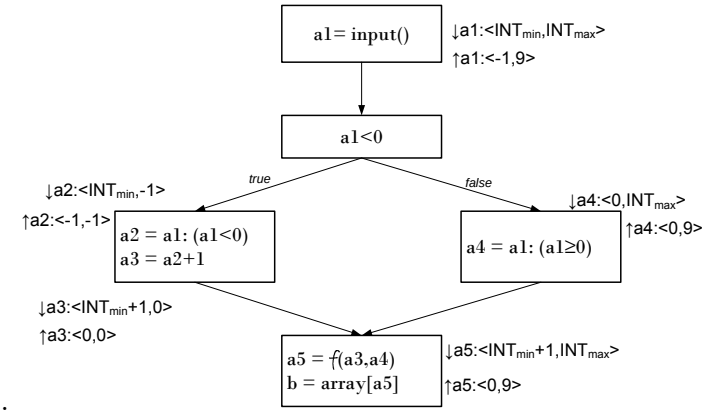


Figure 5.2: The Data propagation analysis with *Bitwise*: The forward propagation is denoted with ↓ and the backward with ↑. The control information is utilized to collect the value ranges of the variables in each branch, and the backward analysis makes use of the array bound informations to tighten the bounds. [Example borrowed from [SBA00]]

5.1.1 Bit-Width Narrowing

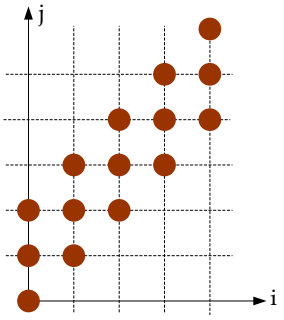
For a numerical data type, the declared bit-width should be consistent with the actual required bit-width, in order to store the data correctly. In Figure 5.1a, the loop control variable is 32-bit wide, although it requires only 5 bits to accommodate the possible values of i, i.e. (0-31). HLS tools allow the designer to explicitly define the bit-width of the variables, as shown in Figure 5.1b. However, this feature still transfers the responsibility for an accurate bit-width analysis to the programmer. A few HLS compilers try to optimize the control structures, but most of the time it is the responsibility of the programmer.

An alternative approach is based on automatic bit-width analysis inside the HLS compiler. With such analysis, the number of required bits for representing a variable, can be estimated. Static bit-width analysis has been implemented in a variety of ways. Budiu et al. [BSWG00] provide a compiler algorithm that tries to determine the possible values of each individual bit. They formulate the problem as a data flow analysis, and propagate possible bit-width values both forward and backward iteratively.

Another variant of bit-width analysis is value range analysis. Range analysis involves studying the data range of each variable and ensuring that the design has enough bits to accommodate the range. Most of the research work for range analysis is based on *Interval Analysis* invented by Moore in 1960s [Moo66]. The *Bitwise* [SBA00] compiler performs the data range propagation both forward and backward over a program control

```
for(i=0;i<1024;i++){
  for(j=i;j<=i+2;j++){
    ... = x[j-i]
  }
}
```

(a)



(b)

Figure 5.3: Bit-width Analysis under the Polyhedral Model: (a) A sample C code, (b) Iteration domain for memory read index $k = j - i$, for array access `x[j-i]`.

flow graph, where all variables are in SSA form. The Control dependent information allows a more accurate collection of data ranges to be found, (See Figure 5.2). Similarly, the array bound information can be utilized in the backward analysis. In the example of Figure 5.2 a constant propagation replaces all occurrences of `a3` with the constant 0. Value propagation analysis for operations inside loop bodies is carried out by finding a closed-form solution for each strongly connected component. Another very powerful bit-width analysis technique is based on the polyhedral mode, [MKFD11], where the iteration domain of each statement is taken into account. In Figure 5.3a, a naive interval analysis for k can lead to the range $< -1024, 1024 >$, although from the iteration domain representation in Figure 5.3b it can be seen that the range of k is $< 0, 2 >$.

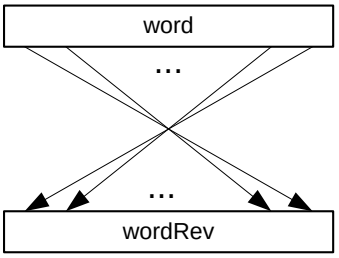
5.1.2 Bit-level Optimization

Bit-wise operations are used extensively in many application domains, e.g. cryptography and telecommunication. Bit-level optimization is an attempt to simplify the logic functions with traditional boolean minimization techniques. For example, a simple operation of $r = a|b\&1$ can be naively translated to an OR gate followed by an AND gate. However, based on constant integer value of 1, compiler can simplify it to a 1-bit OR gate for the lowest bit of a and b and simply wire rest of the bits a to resulting r.

Another expressive example is the bit-reversal, (Figure 5.4), where by fully unrolling the loop, the compiler may greatly simplify the design. It can be noticed that bit-wise operations are tricky to be represented in `C`, using load/shift/mask/store instructions, but since hardware directly supports bit-value accessing and storing, such optimizations are easy at the RTL level. Zhang et. al [ZZZ+10] proposed a new intermediate representation for bit-wise operations, named bit-flow graph (BFG), which contains only the basic logical, shift and conversion operations. They convert a simple DFG node into several BFG nodes, each one representing a bit operation in the original graph. After the BFG construction, redundant operations can be eliminated and the BFG can be converted back to the extended DFG, for further processing.

```
for(i=0;i<32;i++){
  wordRev |= (((word>>i)&1)<<(31-i))
}
```

(a)



(b)

Figure 5.4: Optimized implementation of bit reversal function: (a) C code for Bit Reversal function, (b) Optimized implementation in hardware using wires.

Algebraic Simplifications	Original	Simplified
Multiplication/division with 1/-1	$-1 \times x$	$-x$
Multiplication with 0	$0 \times x$	0
Addition with 0	$0 + x$	x
Common Subexpression Elimination	$a = b \times c + d$ $e = b \times c + g$	$t = b \times c$ $a = t + d$ $e = t + g$
Constant Folding	$3 + (6 * (5 - 2))/2$	12
Constant Propagation	$x = 4,\ y = x + 7$	$x = 4,\ y = 11$

Table 5.1: Simple Algebraic Transformations

5.2 Instruction-level transformations

Simple algebraic transformations, e.g. common subexpression elimination, constant folding and constant propagation, simplify the program and may improve the timing and reduce the resource usage of the design. Most common simplifications are listed in Table. 5.1. In this section we will discuss three key transformations which can be very helpful to expose parallelism for HLS.

5.2.1 Operator Strength Reduction

Strength reduction is an optimization technique where the compiler replaces expensive operators with a sequence of less expensive operators. The replacement can be performed in order to either reduce the time/area cost of the design, or substitute more suitable operator available on hardware. For example, FPGA chips usually include dedicated blocks for particular bit-width multipliers, multiply-accumulate operators and FIR filters, and a compiler can perform strength reduction for a specific FPGA to utilize these accelerators.

A typical example of strength reduction can be replacement of $2 * x$ by $x << 1$, as a shift operation is less costly than a multiplication. Generally, multiplications and divisions by constants can be replaced with a combination of shift and add operations [MPPZ87], e.g. $5 * x$ can be replaced by $x + (x << 2)$. Strength reduction can also simplify the memory addressing inside a loop body, as shown in Figure 5.5. During each iteration,

```
i=0;
while(i<100){
  a[i] = 0;
  i = i + 1;
}
```

(a)

```
i = 0;
L0: t1 = i < 100;
 IfZero(t1) Goto L1;
 t2 = 4 * i ;
 t3 = arr + t2 ;
 *(t3) = 0 ;
 i = i + 1 ;
L1:
```

(b)

```
i = 0;
t0 = arr;
L0: t1 = i < 100;
 IfZero(t1) Goto L1;
 *(t0)=0;
 t0 = t0 + 4;
 i = i + 1 ;
L1:
```

(c)

Figure 5.5: Operator Strength Reduction for memory access: (a) Original Code, (b) Symbolic IR with OSR, (c) Symbolic IR after OSR.

OSR Scenarios	Original	Simplified
Multiplications	$7 \times x$	$x + (x << 3)$
Multiplication with powers of 2	$2 \times x$	$x << 1$
Division with powers of 2	$\lfloor x/2 \rfloor$	$x >> 1$
Square to multiplication	x^2	$x \times x$
induction variable	`for(i=0;i<100;i++) j=i*15;`	`for(i=0;i<100;i++) j=j+15;`
Constant accumulation of form 2^N	$Y_{31:0} = X_{31:0} + 2^{16}$ (32-bit adder)	$Y_{15:0} = X_{15:0}$ $Y_{31:16} = X_{31:16} + 1$ (16-bit adder)

Table 5.2: Operator Strength Reduction

element `a[i]` is accessed by multiplying `i` by 4, but this multiplication can be obviously replaced by an addition.

5.2.2 Height Reduction

A well-known technique to increase ILP is height reduction, in which the compiler exploits commutativity, associativity, and distributivity of arithmetic operations to reduce the number of time steps required to compute the expression. The objective is to reduce the height of expression trees by balancing the operation nodes, where the height represents the number of cycles required to compute the expression. Figure 5.6 shows how rearranging the addition operation reduces the data path height by one cycle.

In *tree-height reduction* (THR) the compiler tries to minimize the expression tree height, by exploiting commutativity and associativity of arithmetic operations. A summation operation of N inputs can be reduced to $\log_2(N)$ steps, forming a binary tree, as it can be seen in Figure 5.7. However, such an optimization can only be beneficial when the memory bandwidth can support the parallel data accesses. In Figure 5.7, If the memory can only be accessed once per cycle, then the THR transformation will produce a worse hardware implementation, in terms of required number of cycles, than the original one, since the memory accesses would have been pipelined otherwise.

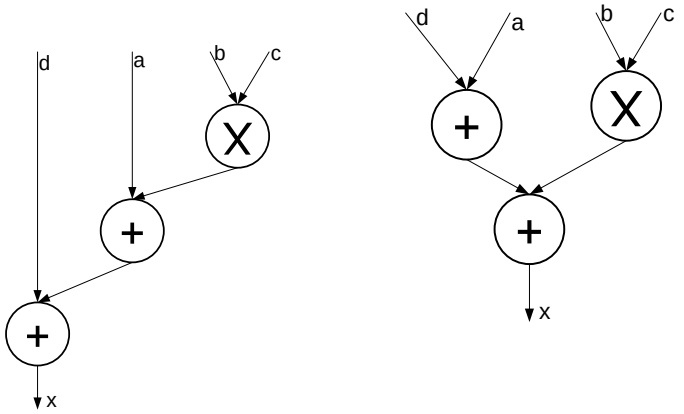


Figure 5.6: Height Reduction: (a) $x = a + b * c + d$, (b) $x = (a + d) + (b * c)$.

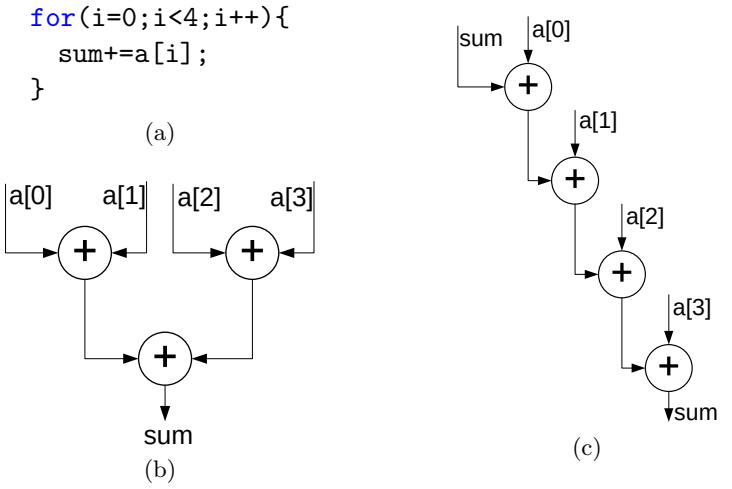
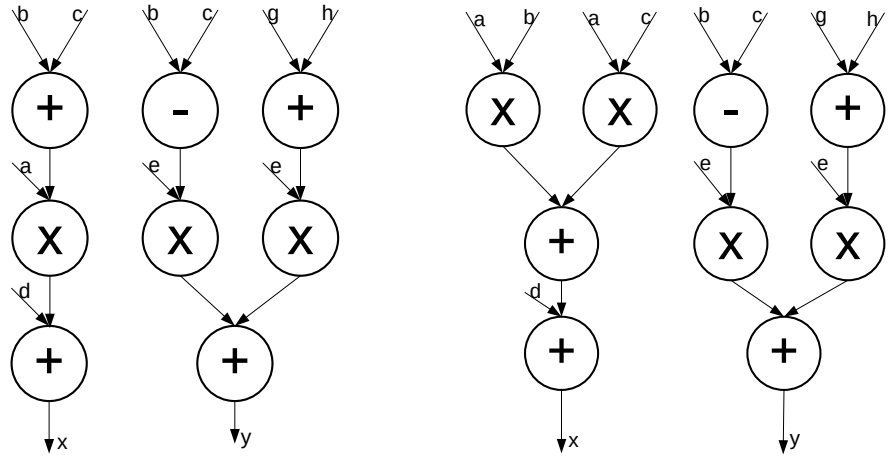


Figure 5.7: Tree Height Reduction: The associativity of addition allows one to compute summation in parallel and in this manner, N steps can be reduced to $\log_2(N)$ steps, provided the input data is available: (a) Original Code, (b) Before THR, (c) After THR. [Example borrowed from [CD08]]

Exploiting the distributivity may increase the number of operations in the expression, but in some cases it may break dependencies thus leading to a better schedule, or it may reduce the required resources for a resource-sharing scenario. In Figure 5.8, distributivity helps to remove the self-inserted dependency of addition before multiplication for x, which results in utilizing two multipliers instead of three in the original version.

Similarly, distributivity can expose common subexpression in a set of expressions [CD08]. For instance, applying the distributivity transformation to the following example:

$$x = a * (b + c + d) \longrightarrow x = a * b + a * c + a * d$$
$$y = a * (b - e) + a * d \longrightarrow y = a * b - a * e + a * d$$



(a) $x = a * (b + c) + d$
$y = e * (g + h) + e * (b - c)$

(b) $x = a * b + a * c + d$
$y = e * (g + h) + e * (b - c)$

Figure 5.8: Exploiting distributivity: The transformation implements x and y with 2 multipliers and 2 add/sub blocks, where as the original code needed 3 multipliers.

reveals that the subexpression $a * b + a * d$ is common to both instructions and can be computed one cycle earlier.

5.2.3 Code Motion

Code motion allows us to change the order of execution of the instructions in a program in order to optimize both area and timing. Code motion is often helpful for expressions inside a loop body, whose value does not change from iteration to iteration, known as *loop-invariant code motion*. For instance, in Figure 5.9b, the computations `c+d` and `b*b` can be computed before the loop, as their value is constant throughout the loop iterations. This kind of code motion results in a shorter schedule length for the loop body. However code motion can sometimes increase the latency of a loop when memory accesses are being moved [CD08]. For example, in Figure 5.9c the access to array `c` is loop invariant for loop `j`, but assuming that memories `b` and `c` can be read in same cycle, the loop invariant code motion does not affect the schedule length. However, in Figure 5.9d, moving `c[i]` outside the inner loop reduces the number of memory accesses to `c` but increases the latency of the loop, as now `b` and `c` will be accessed in separate cycles for the first iteration of `j`.

Code motion techniques, e.g. speculation and reverse speculation, have been extensively studied for control-flow branches in order to expose instruction level parallelism across the conditional blocks [GDGN03, RFJ95]. Code motion can be applied to move a code segment:

- upward from inside a branch block to a split block, i.e. *speculation*.
- downward from a split block to a branch block, i.e. *reverse speculation*.
- upward from a join block to branch blocks, i.e. *conditional speculation* or *duplicating*

```
for(i=0;i<100;i++){
  b=c+d;
  a[i]=2*i+(b*b);
}
```

(a)

```
b=c+d;
tmp=b*b
for(i=0;i<100;i++){
  a[i]=2*i+tmp;
}
```

(b)

```
for(i=0;i<100;i++){
  for(j=0;j<100;j++){
    a[i][j] = b[j]*c[i];
  }
}
```

(c)

```
for(i=0;i<100;i++){
  tmp = c[i];
  for(j=0;j<100;j++){
    a[i][j] = b[j]*tmp;
  }
}
```

(d)

Figure 5.9: Code Motion Examples: (a) Original Code, (b) Transformed Code: moving loop invariant computations `c*d` and `b*b` outside shortens the critical path length, (c) Original Code: array `a,b` and `c` are mapped to disjoint memories and can be accessed during the same clock cycle, (d) Transformed Code: moving `c[i]` outside reduces the number of memory accesses to `c[i]`, but increases the loop latency.

up

– upward across the conditional block, i.e. *useful motion* [RFJ95]

Figure 5.10 shows all four types of speculative code motions.

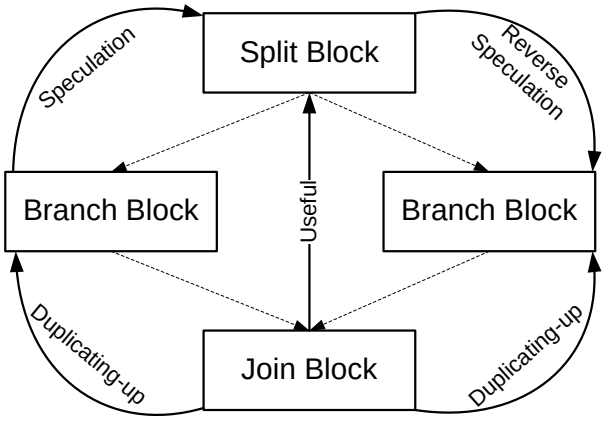
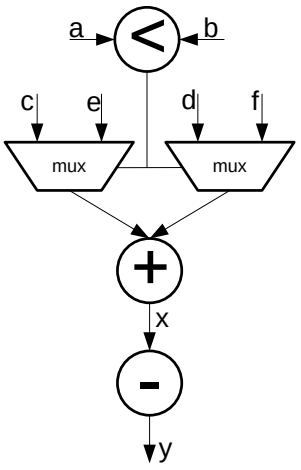


Figure 5.10: Various types of conditional block code motion

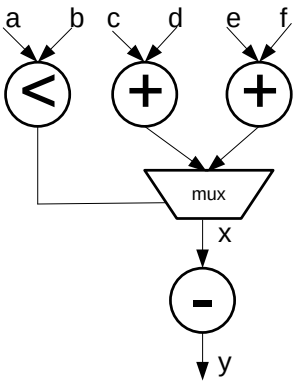
Speculative code motion may shorten the critical path at the price of using extra resources, as shown in Figure 5.11. Speculative code motion can also be helpful to eliminate common subexpression across a conditional block by moving the common expression upward, when there exist a common expression inside one of the branch blocks and of the join block.

```
if(a<b)
  x = c + d;
else
  x = e + f;
y = x - h;
```

(a)



(b)



(c)

Figure 5.11: Speculative Code Motion: Moving addition outside the condition block helps to reduce the critical path, costing an extra adder: (a) Original Code, (b) Original RTL: resource sharing is enabled, (c) Speculation: an extra adder is being used.

5.3 Loop Transformations

Since most of the applications spend a considerable amount of execution time executing loops, loop-level transformations are likely to generate abundant opportunities for parallelism. In the context of HLS from a C program, a loop transformation can be specified with the help of `pragmas`, e.g. loop unrolling, and most of them can be generated automatically. However, in the absence of a key transformation or if the HLS generates an imperfect implementation, the designer may have to expose the parallelism by himself to be able to generate an efficient RTL design from the HLS tool. A better understanding of these transformations is therefore essential to be able to use HLS tools in an efficient manner.

5.3.1 Unrolling

Loop unrolling is the most common transformation to exhibit parallelism in the architecture. Loop unrolling replicates the statements inside the loop body and in this manner it provides the possibility to execute several loop iterations concurrently. This kind of execution is also known as `doacross` concurrent loop scheduling, where processor P_1 executes the first iteration and P_2 executes the second iteration and so on [Wol90]. Loops with constant bounds can theoretically be fully unrolled to achieve parallelism. The replication of the instructions increases the required computational and storage resources, but it also reduces the control overhead for the loop iterations. The concurrent execution of several iterations requires higher data bandwidth, first to read the input data for all iterations being executed in parallel and finally to write back the computed results. Thus, loop unrolling might not be a profitable transformation in situations where data dependencies restrict the concurrent execution or when the available data bandwidth is

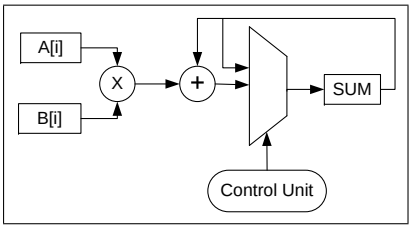
unable to sustain the increased pressure. Loop unrolling may also help the compiler to detect expressions subject to THR, when loop-carried dependencies are related to computations holding associative and commutative properties. After unrolling, such kind of computation may be detected and optimized through THR.

```
for(i=0;i<4;i++){
  sum+=A[i]*B[i];
}
```

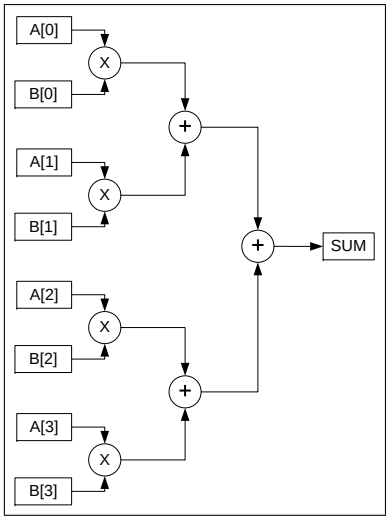
(a)

```
sum+=A[i]*B[i];
sum+=A[i+1]*B[i+1];
sum+=A[i+2]*B[i+2];
sum+=A[i+3]*B[i+3];
```

(b)



(c)



(d)

Figure 5.12: Loop unrolling impact on resulting architecture: (a) Intial loop, (b) Fully unrolled loop, (c) Architecture of the original rolled Loop, (d) Architecture of the fully unrolled loop after THR. [Example borrowed from [CD08]]

5.3.2 Loop Interchange

Loop interchange is the process of switching inner and outer loops. It can be used to expose parallelism, to improve the data locality and to reduce the memory traffic. Memory traffic can be reduced by fetching the operands from memory at the beginning of the loop and by reusing the data throughout the execution of the loop [Wol90]. A similar technique has been used as a power saving technique by minimizing the number of operand changes in a functional unit input [MC95]. Figure 5.13 and 5.14 show how interchange can help to expose common subexpressions and can also be used to move a loop without intra-dependency inward for unrolling.

5.3.3 Loop Shifting

Loop shifting is a type of circuit retiming, a well known hardware technique based on relocating registers to reduce combinational rippling [LS91]. The transformation shifts the loop instructions in order to overcome the data dependencies within a loop iteration.

```
for(j=0;j<N;j++){
  for(i=0;i<N;i++){
    A[i][j]=B[i]*C[i][j];
  }
}
```

(a)

```
for(i=0;i<N;i++){
  tmp=B[i];
  for(j=0;j<N;j++){
    A[i][j]=tmp*C[i][j];
  }
}
```

(b)

Figure 5.13: Loop interchange example: Reducing the memory traffic with the help of loop interchange and a CSE: (a) Original Code: Array `B` will be accessed `NxN` times, (b) Reducing Memory Traffic: The `B` operand will be invariant in the inner loop and can be kept in a register for the duration of the inner loop.

```
for(j=0;j<N;j++){
 for(i=0;i<N;i++){
   A[i][j]=B[i-1][j]*C[i-1][j];
  }
}
```

(a)

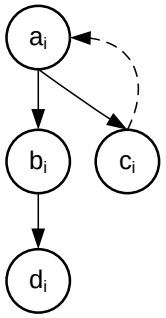
```
for(i=0;i<N;i++){
 for(j=0;j<N;j++){
  A[i][j]=B[i-1][j]*C[i-1][j];
 }
}
```

(b)

Figure 5.14: Loop interchange example: Exposing parallelism by loop interchanging, so that the inner loop can be parallel. (a) Original Code: The inner loop unrolling will not be beneficial due to dependence `i` $\rightarrow$ `i-1`, (b) Parallelism: loop interchange moves the dependencies from inner to outer loop, making the innermost loop fully parallel.

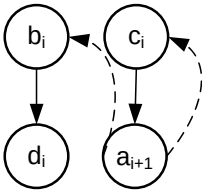
Figure 5.15 shows a code fragment where instructions inside the loop body can not be executed in parallel, due to the data dependency from array `a` to `d` through `b`, hence a loop unroll will not be beneficial. However, a simple retiming enables the latency to be reduced to 2 cycles.

```
for(i=1;i<100;i++){
  a[i] = f1(c[i-1]);
  b[i] = f2(a[i]);
  c[i] = f3(a[i]);
  d[i] = f4(b[i]);
}
```



(a)

```
a[1] = f1(c[0]);
for(i=1;i<100;i++){
  b[i] = f2(a[i]);
  c[i] = f3(a[i]);
  d[i] = f4(b[i]);
  if(i<99)
    a[i+1] = f1(c[i]);
}
```



(b)

Figure 5.15: Loop shifting example: The dependency from $a[i] \rightarrow b[i] \rightarrow d[i]$ is shifted inside the loop body. The dashed arrows show dependency for next loop iteration. (a) Original Code: the loop body requires at least 3 cycles to complete, (b) Loop Shifting: `d[i]` and `a[i+1]` can be computed in parallel.

5.3.4 Loop Peeling

Loop peeling transformation helps to eliminate dependencies established by early loop iterations, restricting parallelization, by moving these early iterations out of the loop. The original code in Figure 5.16 contains a self dependency for the first iteration `a[1]`. By moving this iteration out of the loop, the loop body has no auto dependency and can be executed completely in parallel.

```
for(i=1;i<100;i++){
  a[i] = a[1] + b[i];
}
```

(a)

```
a[1] = a[1] + b[1]
for(i=2;i<100;i++){
  a[i] = a[1] + b[i];
}
```

(b)

Figure 5.16: Loop Peeling removes the self-dependency of the loop body: (a) Orignal Code, (b) Transformed Code.

5.3.5 Loop Skewing

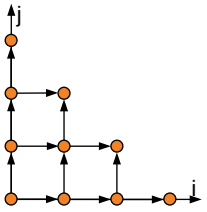
In a situation where the inner-most loop contains an intra-loop dependency, loop unrolling may not be beneficial. Loop skewing transformation helps to expose the parallelism by overlapping the outer loop iterations in such a loop nest, as shown in Fig. 5.17. In this example, loop skewing is able to expose the anti-diagonal parallelism in the loop nest and now the inner loop can be completely unrolled to be executed in parallel. The loop skewing transformation has been extensively used to parallelize bioinformatic algorithms like Smith-Waterman [SW81], Needleman-Wunsch [NW70] and simplified HMMER [OSJM06]. In presence of only diagonal dependencies, loop skewing can be helpful to construct rectangular tiling (discussed later in section 5.3.8).

5.3.6 Loop Fusion

Loop fusion or *loop merging* merges two loops into a single loop by concatenating the bodies of the original loops. Since the transformed loop will execute the original loop bodies in an interleaved order, the transformation should be applied carefully to avoid any violation of data dependencies. Loop fusion is often an adequate transformation for efficient memory contraction (discussed later in section 5.3.11). Fig. 5.18 shows an example where loop fusion allows a `N` element array `T[i]` to be reduced to a scalar variable `t`. Loop fusion has been extensively used, in combination with other transformations, for improving parallelism and data locality [SXWL04]. Loop shifting alone and in combination with loop peeling has been also used to maximize the loop fusion opportunities [Dar99, MA97, SXWL04].

```
for(i=0; i<N; i++){
  for(j=0; j<N-i; j++){
    if(i==0 && j==0)
      A[i][j]= 2*(A[i-1][j]+A[i][j-1]);
  }
}
```

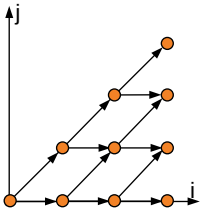
(a)



(b)

```
for(i=0; i<N; i++){
  for(j=0; j<=i; j++){
  t = i-j;
  A[t][j] = 2*(A[t-1][j]+A[t][j-1]);
  }
}
```

(c)



(d)

Figure 5.17: Loop Skewing Transformation: (a) Original Code, (b) Data dependency for original code, (c) Transformed Code: All operations in a vertical column can be executed in parallel now, (d) Data dependency after skewing.

```
for(i=1;i<N;i++){
  T[i] = A[i] + B[i];
}
for(i=1;i<N;i++){
  D[i] = T[i] + C[i];
}
```

(a)

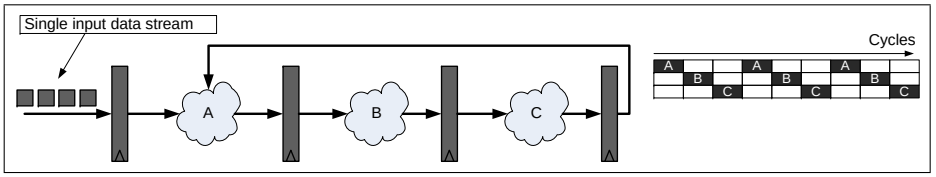
```
for(i=1;i<N;i++){
  t = A[i] + B[i];
  D[i] = t + C[i];
}
```

(b)

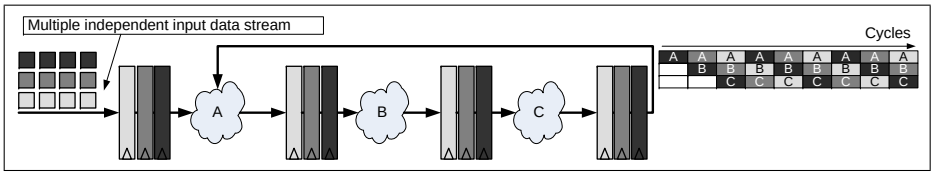
Figure 5.18: Loop Fusion: Loop Fusion + Array Contraction: (a) Original Code, (b) After Loop Fusion.

5.3.7 C-Slowing

Loop pipelining becomes less effective or ineffective, when an inter-iteration dependency exist in the loop, forming a feedback path in the loop body. Pipelining becomes less effective as the initiation interval will always be greater than 1 or totally ineffective when the feedback is precisely from the end to the start of the datapath, as shown in Fig. 5.19. In such a situation, a C-slow pipeline might be implemented, if the design allows multiple independent input data instances to be processed. A C-Slow pipelined loop nest accepts an interleaved data of independent data inputs. Every pipeline stage is active on each cycle, operating on disassociated data. Another way to view this transformation is to consider that we add an additional outer loop iterating over independent instances of the algorithm, and then performing a loop interchange in order to move this outer parallel loop as innermost loop. Finally, the pipelining transformation is applied to implement the multiple independent instances on pipelined hardware.



(a) Pipelining without C-slow: The pipeline with a single input data stream and a feedback loop from the end to the start is not be completely filled at any time instance due to dependencies on previous results. At each cycle only one logic block is active.



(b) Pipelining with C-slow: In presence of a feedback loop in a pipeline, several (precisely equal to the number of pipeline stages inside the feedback loop) independent interleaved input data streams make efficient use of pipeline hardware. Now all logic block are active at each cycle and they are operating on independent data streams.

Figure 5.19: Impact of pipelining with C-slow in presence of a feedback loop.

5.3.8 Loop Tiling & Strip-mining

Loop tiling transforms the iteration space of the loop nest into smaller blocks (also called chunks) of iterations. Loop tiling transforms n nested loops into $2 \times n$ nested loops. The outer loops are the *control loops*, controlling which iteration block to be executed by the inner loops.

Loop strip-mining is a special case of loop tiling, where tiling is applied only to the inner-most loop, instead of the complete loop nest. Loop tiling and strip-mining increase the data locality and increase coarse-grain parallelization opportunities. These transformations may also expose memory mapping opportunities, as each block of iterations may access disjoint memory banks, thus enabling the execution of several blocks in parallel.

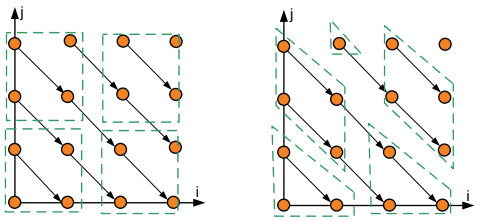
The tile shape and size should be chosen carefully to reduce the communication with external memories and to increase data reuse. For instance, in presence of diagonal data dependencies in Figure 5.20, rectangular tiles are not appropriate for data reuse, but if parallelogram tiles are shaped in the direction of the dependencies, one can benefit from data reuse being produced inside the tile.

5.3.9 Memory Splitting & Interleaving

As loop unrolling replicates the loop instructions, the access to memories in such loop instruction are also replicates. purely a parallel execution of the unrolled loop, the memory banks need to be accessed several times per cycle. But a memory bank can only be accessed in accordance with its number of memory input/output ports. This often hinders a parallel execution of multiple instructions, and a solution to this problem is to split the array memory into disjoint data subsets, where each subset is mapped to a separate

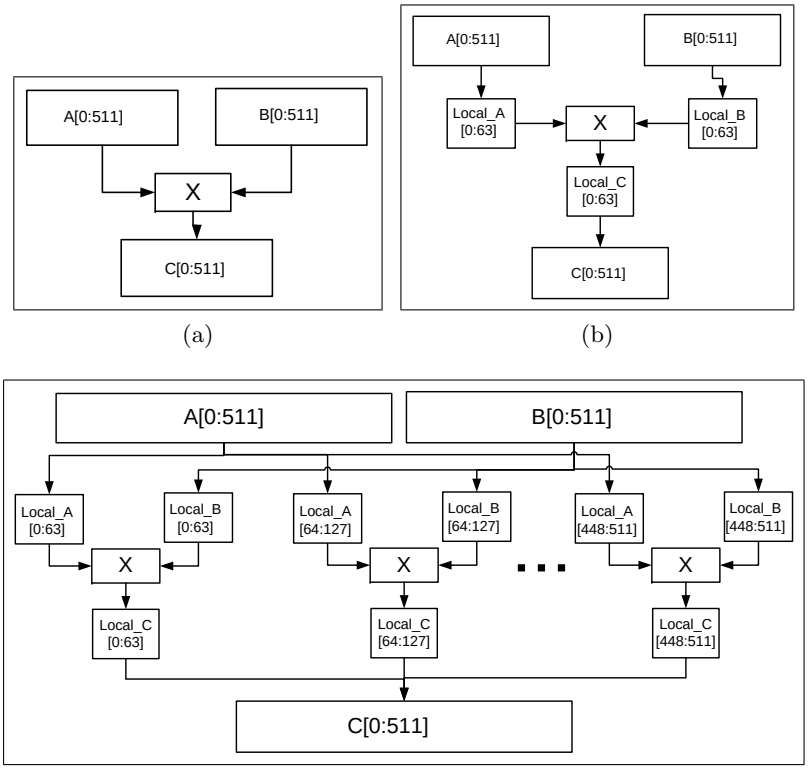
```
for(i=0;i<N;i++){
  for(j=0;j<N;j++){
    c[i+j] += a[i]*b[j];
  }
}
```

(a)



(b) (c)

Figure 5.20: Loop Tiling: Impact of tile shape on data reuse opportunity [Ple10]



(a) (b)

(c)

Figure 5.21: Impact of tiling and strip-mining for a vector multiplication example $C[i] = A[i] * B[i]$: (a) Original Code: No data locality is implemented for input-output memories, (b) Strip-mining: Data being accessed in the form of strips of size 64, (c) Coarse-grained parallelization of strip-mined code. [Example borrowed from [CD08]]

memory bank. Fig. 5.22 shows two types of memory splitting using blocks or interleaving. Both techniques can be used to express fine and coarse-grain parallelism.

5.3.10 Data Replication, Reuse and Scalar Replacement

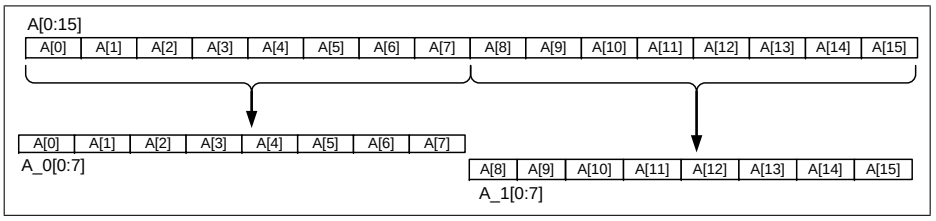
In many computations, particularly in dynamic programing algorithms, data values are reused. A compiler sometimes identify the data reuse in a computation, and save this data in scalar variables to avoid multiple memory accesses. Figure 5.23 shows example of scalar

```
for(i=0;i<15;i++){
  A[i]
}
```

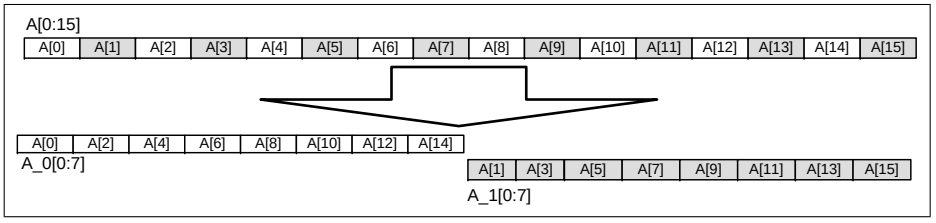
(a)

```
for(i=0;i<7;i++){
  A_0[i]
  A_1[i]
}
```

(b)



(c)



(d)

Figure 5.22: Memory splitting of an array memory into disjoint memory modules: (a) Original Code, (b) Transformed code, (c) Block splitting, (d) Interleave splitting.

replacement. The original loop requires three accesses to the external memory bank for each loop iteration. However, by scalar replacement, the memory access is reduced to one write operation. The use of local registers substantially decreases the data access latency and the number of external memory accesses.

```
for(i=2;i<N;i++){
  A[i] = A[i-1] + A[i-2];
}
```

(a)

```
tmp2=A[0];
tmp1=A[1];
for(i=2;i<N;i++){
  tmp3 = tmp1 + tmp2;
  tmp2 = tmp1;
  tmp1 = tmp3;
  A[i] = tmp3;
}
```

(b)

Figure 5.23: Scalar Replacement: (a) Original Code, (b) Scalar replacement exploits the data reuse and reduces memory accesses to a single write operation.

5.3.11 Array Contraction

Array contraction is a transformation that reduces the array size while preserving the correct output of the program. Typically, array contraction helps to contract, or converts to a scalar, a temporary array introduced by designer in order to store intermediate computations being used in several successive loops, as shown in Figure 5.18. Loop fusion and loop shifting transformations have been used to enable array contraction [DH02, SXWL04, GOST92, KM94]. An early work by Sarkar and Gao focused on finding the most suitable loop reversal transformation to enable array contraction [SG91]. For programs with affine array index functions and loop bounds, Alias et al. provide an array contraction algorithm for intra-array memory reuse [ABD07]. Intra-array memory reuse reduces the size of temporary arrays by reusing the memory locations when they contain a data that is not used later.

```
#define N 200
int t[N][N];
for(i=1;i<N;i++){
 for(j=1;j<N;j++){
   t[i][j]= ...
   out = t[i][j-1]-t[i][j];
  }
}
```

(a)

```
#define N 200
int t[2];
for(i=0;i<N;i++){
 for(j=1;j<N;j++){
   t[i%2]= ...
   out = t[(i-1)%2]-t[i%2];
 }
}
```

(b)

```
#define N 200
int t[N][N];
for(i=1;i<N;i++){
 for(j=1;j<N;j++){
   t[i][j]=t[i-1][j-1]
           -t[i-1][j];
  }
}
```

(c)

```
#define N 200
int t[2][N];
for(i=0;i<N;i++){
 for(j=1;j<N;j++){
   t[i%2][j]=t[(i-1)%2][j-1]
             -t[(i-1)%2][j];
 }
}
```

(d)

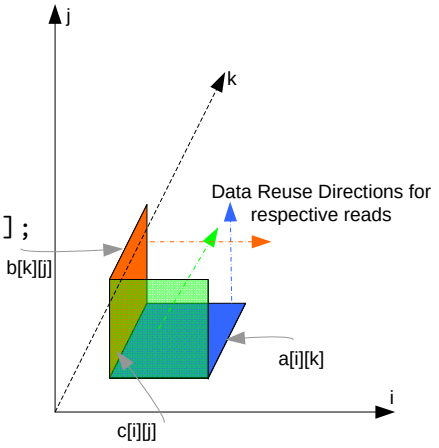
Figure 5.24: Array Contraction Example: (a) Original Code, (b) Memory contracted to 2 cells, (c) Original Code, (d) Memory contracted to 2 columns.

5.3.12 Data Prefetching

A hardware accelerator accessing external memories, such as SDRAM, may suffer from serious performance degradation due to non-consecutive accesses to DDR. For example, in Figure 5.25a, array `a,b` and `c` are accessed in an interleaved manner and slow down the system performance. Plesco [Ple10] derived a data fetch mechanism for such architecture for C2H HLS tool. Loop tiling is used to improve the data locality and it also tries to increase the data reuse, with the help of local memories, in order to to reduce the memory

```
for(i=0;i<=N;i++){
  for(j=0;j<=N;j++){
    for(k=0;k<=N;k++){
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

(a)



(b)

Figure 5.25: Data Prefetching for a matrix multiplication example. [Example borrowed from [Ple10]]

traffic. Here, `c[i][j]` should be read once at the start of loop `k` and should be written once at the end of loop `k`. At the start of a tile, only the input data which are not produced by a previous tile, or not already loaded for a previous tile are to be loaded from the DDR. Similarly, only that data which is no more needed by any future tile are stored to DDR . Later on, the local memories are optimized using array contraction and data lifetime analysis. Figure 5.25b shows a single tile data inputs with colored tiles. The arrows shows the direction of reuse of this input data, i.e. the subsequent tiles in this direction, use the same data already loaded, as input to the current tile, from DDR.

5.3.13 Memory Duplication

In order to improve parallelism, memory duplication can be beneficial in many cases by enabling parallel accesses and by removing read-write dependencies for the same memory. An important application is a scenario where tiling or strip-mining is done in presence of a diagonal or vertical dependency, and a write operation is to be performed outside the current tile. For instance, array contraction reduces the memory `t` to 2 columns in Figure 5.24c and 5.24d. However, in case of tiling, we can reduce it to a single column by using memory duplication only for the tile boundaries. The example in Figure 5.26a shows a loop nest, where inner-most loop is strip-mined and each strip is unrolled to execute in parallel. Due to the presence of diagonal dependency (`A[j]`→`A[j-1]`), the input to the first computation of the strip will be corrupted, as it has been updated to a new value in last cycle. The code in Figure 5.26b uses an extra cell `tempL` to store the temporary result of the boundary computation, and update the original memory cell in next cycle.

Figure 5.27 shows the graphical illustration of such read/write operations for a single column memory(self read operations are not shown here). Dashed arrows show the diagonal read operations and solid arrows show the write operations. The horizontal dash-dot lines show the boundary of a strip. All operations inside a strip are executed in

```
for(i=1;i<N;i++){
  A[0] = -infty;
  for(j=0;j<9;j+=3){
    // Scalar Replacement
    in0=A[j];
    in1=A[j+1];
    in2=A[j+2];
    in3=A[j+3];
    // Computation
    tmp_0 = fun(in1, in0);
    tmp_1 = fun(in2, in1);
    tmp_2 = fun(in3, in2);
    // Write Back
    A[j+1]=tmp_0;
    A[j+2]=tmp_1;
    A[j+3]=tmp_2;
  }
}
```

(a)

```
for(i=1;i<N;i++){
  tmpL = -infty;
  for(j=0;j<9;j+=3){
    // Scalar Replacement
    in0=tmpL;
    in1=A[j+1];
    in2=A[j+2];
    in3=A[j+3];
    // Computation
    tmp_0 = fun(in1, in0);
    tmp_1 = fun(in2, in1);
    tmp_2 = fun(in3, in2);
    // Write Back
    A[j] = tmpL;
    A[j+1] = tmp_0;
    A[j+2] = tmp_1;
    tmpL = tmp_2;
  }
}
```

(b)

Figure 5.26: Memory Duplication: The code in (a) is corrupt, since old value of `A[j]` is overwritten in last iteration as `A[j+3]`, and a naive solution will be to duplicate entire memory `A`. However, the code (b) uses an additional register `tempL` to store only the newly computed value of the corner of the tile and update the original memory cell in the next cycle.

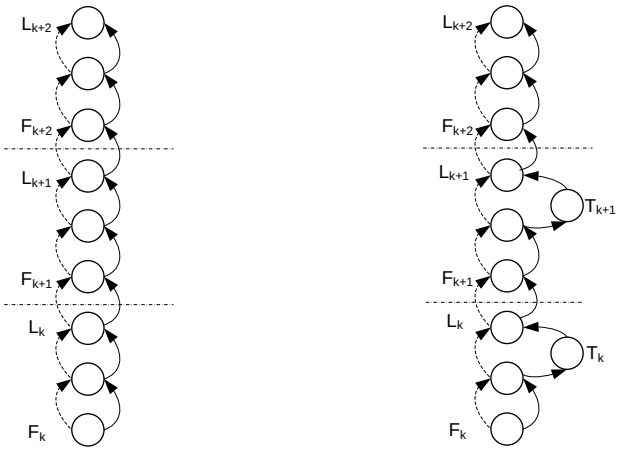


Figure 5.27: Memory Duplication Example: Memory duplication helps to reduce the memory layout from $2 \times N$ cells to $N+1$ cells. (a) Original Layout: Functionally incorrect, (b) Duplicated Memory.

parallel. We can see that the first operation of a strip k, F_k needs to read the previous value of L_{k-1}. However since, L_{k-1} is updated in last cycle , the input value read for F_k in next cycle will be corrupted, as shown in Figure 5.27a. Figure 5.27b shows a solution to this problem by using a temporary memory T_k. Now, for the execution of any strip k, the memories from L_{k-1} till L_k will be read and F_k till $L_k - 1$ will be written along with a temporary memory T_k, which will update L_k at next cycle.

5.4 Conclusion

It has been extensively observed that the quality of the generated RTL design from an HLS tool largely depends on the quality of its input specifications. This chapter covers various code transformations that may help to improve the quality of the input code. Since most of the HLS tools use a C-dialect, this chapter shows how different versions of a C code input may lead to different speed and area results at the hardware level. We have discussed such transformations at various levels, i.e. bit, instruction and loop-level transformations.

Bit-level transformations allow one to express the custom bit-width storage and operation in hardware. Bit-width narrowing techniques help to find the data types with the exact required precision. The reduced bit-width representation may result in a huge resource conservation and may improve clock speed. Since bioinformatics algorithms mostly operate on `char` data types, the reduced bit-width implementation on FPGAs is helpful for accelerating the kernel computation. It also reduces the resource requirement, which allow more independent kernels to be embedded, amplifying the coarse grain acceleration.

Instruction level transformations simplify the mathematical computations, thus they can reduce the required resources and the number of execution cycles for the operations. The mathematical properties of operations, such as associativity, commutativity and distributivity can be exploited to rearrange the computations in such a way that it may result in a reduction of the number of execution cycles or in the reduction in resource requirement by resource sharing. Similarly, expensive operations e.g. multiplication and division, involving constants, can be transformed into inexpensive shift and add operations.

Loop transformations play a vital role in the design, because the execution time of a computation kernels is mostly spend inside loops. Hence, improving the parallelism inside loops can greatly accelerate the kernel. In HLS, loop parallelism can be either expressed through *unrolling*, i.e. several loop iterations executed in parallel, or through *pipelining*, i.e. several loop iterations executed in an overlapped manner. The majority of the other transformations, such as *interchange*, *shift*, *peel*, *skew*, enables the code to be unrolled and/or pipelined in the most beneficial way. Similarly transformations like *Memory Splitting and Interleaving*, *Data Replication*, *Prefetching* ensure the data availability, avoiding memory access delay.

HLS based design is rapid and error-free in comparison with RTL based design.

However, there is still lack of adoption from professional community, as the design generated from these tools is often poor in performance in comparison with a finely tuned manual design. However, this difference in performance can be reduced by expressing design more sensibly. A careful application of these code transformation techniques can lead to an as efficient RTL design, as a manual coded design can be. In coming chapters, we apply, the techniques we have just learned, on a well-known compute-intensive application of bioninformatics (HMMER), and show how efficient HLS based design can be. This will rest our case that HLS based FPGA development is fast, efficient and generic.

6

Extracting Parallelism in HMMER

6.1 Introduction

Sequence database homology searching is one of the most important applications in computational molecular biology. In this application, protein sequences of unknown characteristics are searched against a database of known sequences in order to predict their functions and to classify them in families. Performing this operation at a large scale is however becoming time prohibitive due to the exponentially growing size of sequence databases, which double every eighteen months [NCB11]. Over the last few years, reconfigurable computing has proved to be an attractive solution for accelerating compute-intensive bioinformatic applications, such as Smith Waterman [SW81] or BLAST [AGM+90]. The possibility of massive parallel processing, power efficiency and comparable price/performance solutions makes FPGA-based accelerations a practicable alternative to other supercomputing infrastructures such as vector computers or PC clusters.

Sequence alignment techniques based on Hidden Markov Models (HMMs)[RJ86] has generated very good results [Edd98]. Profile HMMs, introduced to computational biology by Krogh et al. [KBM+94] for representing profiles of multiple sequence alignments, has been successfully applied in homology detection and protein classification ([Edd98, HKMS93, JH98]). A profile HMM, built from multiple sequence alignments of the sequence family, concentrates on the features or key residues conserved by the family of sequences, so it can find even a remote sequence homolog which can not be detected by pairwise alignment techniques (e.g. BLAST [AGM+90] or Smith-Waterman [SW81]).

One of the most commonly used program for HMM analysis is the open source software suite HMMER, developed at Washington University, St. Louis by Sean Eddy [Edd]. HMMER involves very computationally demanding algorithms and accounts for a large amount of time spent in biological sequence analysis. Many authors

have also investigated dedicated parallel hardware implementations, notably on FPGAs and GPUs [MBC+06, OSJM06, BVK08, JLBC07, OYS07, DQ07, TM09, SLG+09, EGMJdM10, HHH05, WBKC09a, GCBT10]. Recently, a new software implementation (HMMER3) has been released and a great deal of effort has been put to improve its software execution through both fine grain (SIMD extension such as ALTIVEC and SSE extensions) and coarse grain parallelization (using MPI or multi-threads) [Edd11b]. It has been shown that DualCore SSE implementation of HMMER3 is faster than most of previous FPGA and GPU versions of HMMER2.

As currently defined and programmed, HMMER3 spends most of its time in two kernel functions called MSV and P7Viterbi. These two kernels contain so-called loop-carried dependencies (caused by the feedback path from the end to the beginning of the model) which restricts any kind of parallelism.

We propose a technique to rewrite the computations in such a way that both kernels become very amenable to parallel implementation, while keeping all the original dependencies into account. In this chapter we describe how the original dynamic programming equations of MSV and P7Viterbi can be rewritten so as to develop a new algorithm that admits a scalable parallelization scheme, at the price of a moderate, constant factor increase in the algorithm computational volume.

6.2 Background

6.2.1 Profile HMMs

HMMs are stochastic models that capture the statistical properties of observed data. A Hidden Markov Model is a finite set of states, each of which is associated with a probability distribution, and the transitions among the states are governed by a set of transition probabilities. A profile HMM of a family of biological sequences is built from the multiple sequence alignments. For each column in the Profile HMM, a *match* state models the allowed residue, while an *insert* and *delete* state models the insertion of one or more residues, or the deletion of a residue. The multiple alignment of a sequence family shows the pattern of conservation of the sequence, i.e. some regions are more conserved by the family and some regions seem to tolerate insertions and deletions. The position specific information shows the degree of conservation in some positions and the degree of variation to which insertions and deletions are permitted. Profile HMMs use this information for position specific scoring, e.g. there will be more penalty for insertion/deletion in a conserved region than in a region of tolerance. Traditional pairwise alignments, like BLAST [AGM+90] or Smith-Waterman [SW81], use position independent scoring (i.e. gap penalties are globally fixed) and the pattern of conservation in a sequence family is not considered. Several available software packages implement profile HMMs or HMM-like models. The HMMER toolsuite uses the 'Plan7' HMM architecture shown in Fig. 6.1. The Plan7 HMM incorporates multiple features in a single model [Edd98]: local alignment with respect to the model (through $B \rightarrow M \rightarrow E$ paths), local alignment with respect

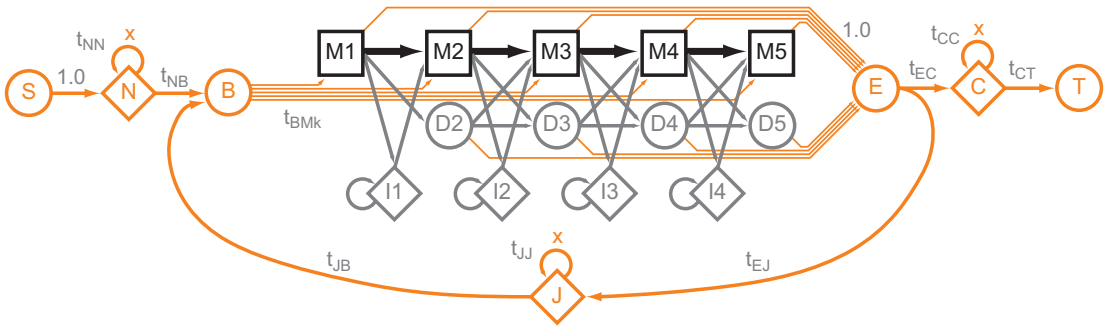


Figure 6.1: Structure of a Plan7 HMM [Edd11b]

to the sequence (through flanking insert states) and more than one hit to the HMM per sequence (through feedback loop $E \to J \to B$). HMMER implements the Plan7 HMM in the P7Viterbi kernel that we describe now.

6.2.2 P7Viterbi Algorithm Description

P7Viterbi is the most time consuming kernel inside the *hmmsearch* tool. This kernel solves Plan7 HMMs through the well-known Viterbi dynamic programming algorithm. The architecture of Plan7 HMM model is shown in Fig.6.1. The M (Matching), I (Insertion) and D (Deletion) states constitute the core section of the model, whose equations are:

$$M_i[k] = \max \begin{cases} e_M(\text{Seq}_i, k) + \max \begin{cases} M_{i-1}[k-1] + \text{TMM}[k] \\ I_{i-1}[k-1] + \text{TIM}[k] \\ D_{i-1}[k-1] + \text{TDM}[k] \\ B_{i-1} + \text{TBM}[k] \end{cases} \\ -\infty \end{cases} \tag{6.1}$$

$$I_i[k] = \max \begin{cases} e_I(\text{Seq}_i, k) + \max \begin{cases} M_{i-1}[k] + \text{TMI}[k] \\ I_{i-1}[k] + \text{TII}[k] \end{cases} \\ -\infty \end{cases} \tag{6.2}$$

$$D_i[k] = \max \begin{cases} M_i[k-1] + \text{TMD}[k] \\ D_i[k-1] + \text{TDD}[k] \\ -\infty \end{cases} \tag{6.3}$$

The Seq_i in Eq.(6.1) and Eq.(6.2) is the current sequence character being aligned. States N, B, E, C and J are called "control states". State B and E are dummy non-emitting states, representing the beginning and the end of the model:

$$E_i = \max_k \left(M_i[k] + \text{TME}[k], -\infty \right) \tag{6.4}$$

$$B_i = \max\left(N_i + t_{\text{NB}}, J_i + t_{\text{JB}}, -\infty\right) \tag{6.5}$$

States N, J and C are used to control algorithm-dependent features like local and multi-hit alignments:

$$N_i = \max\left(N_{i-1} + t_{\text{NN}}, -\infty\right) \tag{6.6}$$

$$J_i = \max\left(E_i + t_{\text{EJ}}, J_{i-1} + t_{\text{JJ}}, -\infty\right) \tag{6.7}$$

$$C_i = \max\left(C_{i-1} + t_{\text{CC}}, E_i + t_{\text{EC}}, -\infty\right) \tag{6.8}$$

The e_M, e_I, TMM, TIM, etc., are transition memories while t_{NB}, t_{JB}, etc., are a set of constants.

States E, J and B form a feedback loop in the model (i.e., a cycle in the graph), which rarely changes the value of M. Many hardware accelerators exploit this fact and ignore this feedback path, as will be shown in section 6.3.1.

On the other hand, this feedback loop gives HMMER the ability to perform multiple hit alignments, i.e. more than one segment per sequence can be aligned to the core section of the model. The self-loop over J provides the separating sequence length between two aligned segments. Thus, from an algorithmic point of view, it is incorrect to ignore this edge in the model.

The computations in (6.3) and (6.4) are the most crucial as far as extracting parallelism is concerned. Early implementations used to ignore the B_{i-1} in (6.1), and which removes the inter-column cyclic dependency (i.e. $M_i \rightarrow E_i \rightarrow J_i \rightarrow B_i \rightarrow M_{i+1}$). In our work, we will rewrite the (6.3) in such a way that the dependency $D_i[k] \longrightarrow D_i[k-1]$ is transformed into a look-ahead computation, thus allowing computations in column k to be done in parallel.

6.2.3 Look ahead Computations

The feedback in a recursive algorithm often destroys the opportunity to pipeline or parallelize the execution. Consider a first order recursion:

$$T_k = a_{k-1} \otimes T_{k-1} \oplus u_{k-1} \qquad \forall k : 1 \leq k \leq N \tag{6.9}$$

The dependence $T_k \rightarrow T_{k-1}$ enforces the sequential execution of the computations. However, in order to obtain the parallelism, look-ahead computations can be defined as an algorithmic transformation to express T_{k+m} in term of T_k, without directly depending on the values of $T_{k+m-1} \ldots T_{k+1}$. This algorithmic transformation is based on the properties (commutativity and distributivity) of the algebraic functions involved, i.e.

$(\oplus, \otimes)$. Fettweis et al. [FTM90] define the algebraic structures amenable to look-ahead computations. Fettweis and Meyr [FM89],[FM91] also apply a similar concept to a simplified Viterbi decoder and show the possibility of look-ahead computations in the presence of add/compare/select recursions in the decoder.

In this chapter, we propose a similar look-ahead computation architecture for a more complex Viterbi decoder. The computation $D_i[k]$ in (6.3) holds a similar auto-dependency, as in the recursion in (6.9), and the algebraic functions involved in the computation, *sum* and *max*, also hold the above mentioned algebraic properties. This makes (6.3) suitable for a look-ahead scheme. Our parallelization strategy is based on transforming the intra-column dependency of (6.3) into look-ahead computations, as described in Section 6.5.

6.3 Related work

6.3.1 Early Implementations

HMMER has received a lot of attention from the high performance computing community, with several implementations either for standard parallel machines or for more heterogeneous architectures [WBKC09a, WBC05, LPAF08]. In the following we will focus on hardware implementations targeting ASIC or FPGA technologies.

Simplified Viterbi Implementations: Early proposals [MBC+06, OSJM06, BVK08, JLBC07] of hardware accelerators for profile-based similarity search considered an over-simplified version of the algorithm in which the feedback loop was ignored, as shown in Figure 6.2. The simplification was based on the idea that the feedback loop has a relatively limited impact on the actual quality of the algorithm. The removal of the feedback loop, removes the inter-column dependency in the Viterbi algorithm and allows the column-wise computations to be overlapped through *loop skewing*, as discussed in Section 5.3.5. The feedback-free recurrence can be computed in $N + L$ steps, compared to $N \times L$ steps for the original recurrence, where N is the size of model and L is the length of query sequence. However, as discussed in section 6.2.2, feedback loop detects multiple hits of a single motif, the feedback-free algorithm generates false-negative results and reduces the sensitivity of the original algorithm.

Exact Viterbi Implementations: Since the presence of a feedback loop in the original Viterbi algorithm does not allow computation of several cells to be done in parallel, the exact implementation of Viterbi will be considerably slow. However, researchers have taken advantage of the fact that during each call to HMMER, a full database of independent input sequences need to be processed. Thus, acceleration can be done by computing several instances of Viterbi in parallel.

The first hardware implementation of the exact algorithm was proposed by Oliver et al [OYS07]. By aligning several input sequences independently on separate PEs. They were able to fit 10 independent PEs on a Xilinx Spartan-3 board. However, one issue

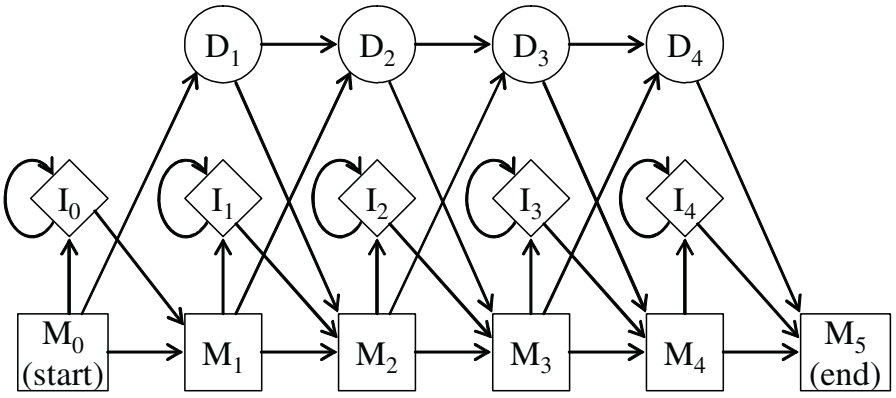


Figure 6.2: A simplified HMM model, without the feedback loop from end to the start of the model [OSJM06].

with their parallelization scheme is that all PEs need to access the same emission cost look-up tables for $e_M(Seq_i, k)$ and $e_I(Seq_i, k)$, see Equation (6.1) and (6.2). On the other hand, for each cycle, all PEs need to access only a subset of the tables, where the subset is defined by the unit increment in k and the random value of the sequence character, Seq_i. Since the number of amino-acid alphabets is 24, the size of the subset is $e_M(0 \ldots 23, k)$ and $e_I(0 \ldots 23, k)$. Oliver et al. addressed this problem by implementing a 24-element wide data bus, and each PE selects its cost value using a 24-to-1 multiplexer. This solution suffers from severe scalability issues, and makes it impractical for massively parallel implementation.

Another approach was proposed by Derrien and Quinton [DQ07]. It also uses the fact that many instances of the Viterbi algorithm can be processed in parallel. However, the parallelization scheme (based on polyhedral space-time transformations) is more sophisticated than that of Oliver et al. [OYS07]. In their approach, each PE operates on all input sequences, instead of each PE operating on independent sequence. The proposed architecture does not need to access a shared memory for calculating the transition costs, since the emission tables are partitioned among the PEs and each PE requires to access only a subset of the emission table.

Figure 6.3 shows a space-time mapping for an exemplary architecture, with $M = N = 4$ and $L = 5$. The computational dependencies of the Viterbi algorithm are on i-k plane and there is no dependency on the j axis (due to independent sequences). The space-time map shows an implementation of 2 PEs on the k-axis, where the black dots correspond to the iteration sub-space allocated to first PE. The first PE processes by starting 2 computations of the HMM model for every input sequence, and thus only needs to access a sub-set of the emission tables, i.e. $e_M(0 \ldots 23, 0 \ldots 1)$ and $e_I(0 \ldots 23, 0 \ldots 1)$, which is implemented as a separate RAM being accessed by only one single PE. The solid arrows show the iteration execution order of a single PE. In this approach, each PE operates on interleaved input sequences, compared to Oliver et al. where each PE operates on independent sequence. It can be seen that sequence interleaving allows a delay of one cycle between the dependent

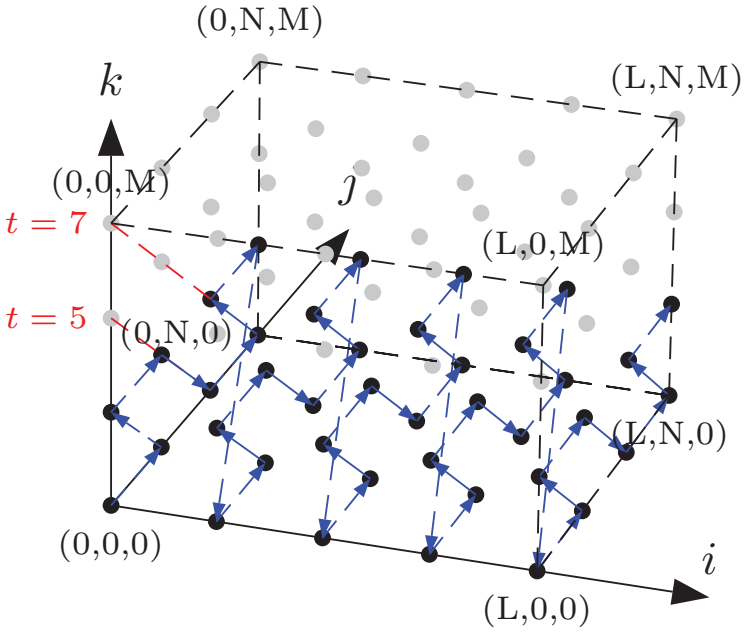


Figure 6.3: Pipelined space-time mapping proposed by Derrien and Quinton [DQ07]: Here L denotes the length of query sequence, M the size of HMM model, and N the number of independent sequences being processed in parallel. The blue arrows show the execution order of operations on a single PE. [Courtesy [DQ07]]

computations and hence helps to pipeline the datapath. By changing the execution order, the delay can be adjusted flexibly.

The proposed approach easily handles resource constraints by controlling the number of PEs in the architecture, and allows the datapath pipeline to be precisely tuned. However, the scalability of this approach is still somewhat limited, as the local storage requirements of the hardware implementation can be prohibitive. For example a 64-element processing array with a 6 stage pipelined datapath would need more than 500 embedded memory blocks on a FPGA.

6.3.2 Speculative Execution of the Viterbi Algorithm

More recently, an approach for hardware acceleration based on speculative execution was proposed by Takagi et al. [TM09] and Sun et al. [SLG+09]. Their idea is to take advantage of properties of the *max* operation, so as speculatively ignore the dependency over variable B_i in (6.1), since it very seldom contributes to the actual computation of $M_{i+1}[k]$, $D_{i+1}[k]$ and $I_{i+1}[k]$. Ignoring this dependency results in a feedback-loop free algorithm, which is very easy to parallelize.

Whenever it is observed that the actual value of B_i would have contributed to the actual value of $M_{i+1}[0]$, all computations related to columns i' such as $i' > i$ are discarded (flushed) and the computation must be re-executed so as to enforce the original algorithm dependencies, as shown in Figure 6.4. To do so, Takagi et al. propose a misspeculation detection mechanism which stores in a buffer the values of M, D and I computed at the

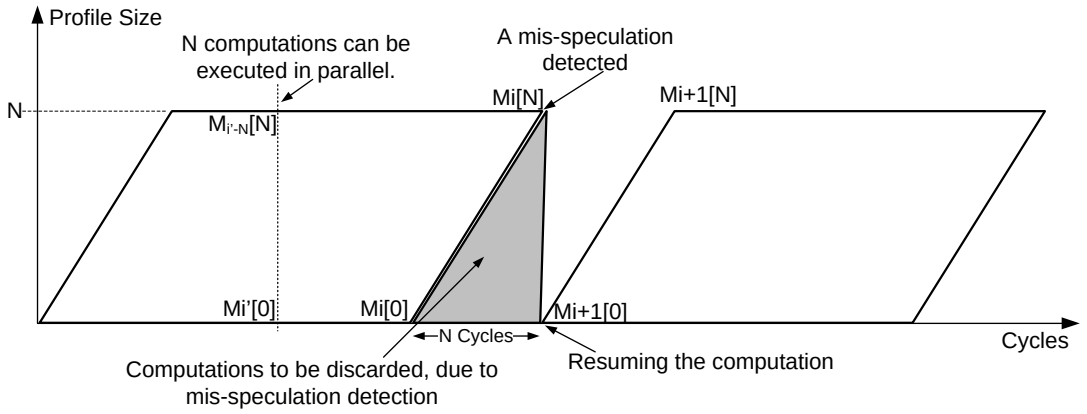


Figure 6.4: Speculative execution of the Viterbi algorithm

beginning of the new column (together with their inputs) until the actual value of B is available (that is N_{prof} cycles later). The true values of M, D and I are then recomputed, and if they differ from the previous one, it means that the speculation was wrong, and that the previous results must be discarded.

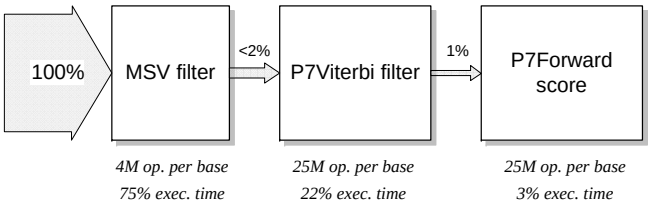
The main issue of such an approach is to estimate the probability and the cost of a misprediction. In this solution, whenever a misprediction occurs, the architecture has been running useless calculations during last N cycles. Assuming a misspeculation probability p, the execution overhead for a sequence of S amino-acid bases can then be written as :

$$e = \frac{S + N}{S + N + pSN}$$

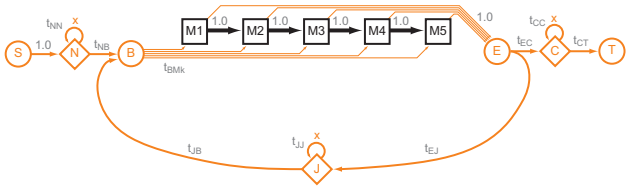
As noticed by Takagi et al, the average observed value for p is 0.0001, which leads to an efficiency that vary between 94% and 99% depending on the depth of the speculation. It can also be observed that the overhead is more important for an architecture exhibiting a large level of parallelism (the *depth* of the speculation being deeper), and for long sequences matched against small profiles, for which the probability of observing a repetition is cumulative with the sequence size. As an example Takagi et al. report cases where HMM profile characteristics lead to a poor efficiency (performance degradation by 85 %).

Very recently, Eusse Giraldo et al. [EGMJdM10] proposed another approach. They used a simplified (without J state) Viterbi kernel as a filter and pass only sequences with significant scores to original Viterbi kernel along with a divergence algorithm [BBdM08] data. The divergence algorithm data reduces the number of cells that must be calculated with the original Viterbi kernel by providing limits of the alignment region. The alignment region defines where the alignment starts and ends. This approach yields an acceleration of 5.8 GCUPS (Giga Cell Updates Per Second).

However, the use of a simplified Viterbi algorithm as a filter may not detect multiple hit alignments. As the filter also specifies the alignment region to the original Viterbi kernel, the original kernel will not try to align sequence segments lying outside the alignment region, and this may produce false negatives. The paper [EGMJdM10] does not discuss



(a) HMMER3 execution pipeline, with profiling data



(b) MSV Filter [Edd11b]

Figure 6.5: HMMER3 execution pipeline and MSV filter

issues with the multiple hit alignments, and how this is handled inside the simplified Viterbi filter.

6.3.3 GPU Implementations of HMMER

The HMMER search was implemented on graphics processing units by Horn et al. [HHH05] as Claw-HMMER and later by Walters et al. [WBKC09a]. The overall speed up reported by Walters is 15 to 35 × on a single NVIDIA 8800 GTX Ultra GPU in comparison with a software implementation. The GPU implementation lacks a very high speedup due to extensive global memory access by P7Viterbi algorithm.

A very recent implementation on GPU by Ganesan et al. [GCBT10] accelerates the HMMER by breaking the chain of dependencies inside P7Viterbi. The reported speed-up is 100+ times on 4 Tesla C1060 GPUs in comparison with a software implementation of HMMER2. They converted the vertical cell dependency $D_i[k] \longrightarrow D_i[k-1]$ into dependencies between equal sized chunks of a column. During the first stage, the headers of the chunks are updated serially, then the intermediate values of each chunk are computed in parallel. Ganesan et al. follow an almost similar strategy as we do for breaking this dependency in computation of D, but we convert this dependency into well-known parallel prefix networks. The parallel prefix network topologies provide the freedom to compromise between delay and area costs according to the architecture requirement, as shown in later Sections.

6.3.4 HMMER3 and the Multi Ungapped Segment Heuristic

The new version of HMMER, which is available for use now, is a radical rewrite of the original algorithm, with a clear emphasis on speed-up. The most noticeable difference in this new version lies in a new filtering heuristic (called *Multi Ungapped Segment Viterbi*) which serves as a prefiltering step, and MSV is executed before the standard P7Viterbi in

the HMMER pipeline as illustrated in Fig. 6.5. This algorithmic modification alone helps improving the speed-up by a factor of 10.

It is important to note that the MSV still holds the feedback loop ($E \rightarrow J \rightarrow B$), which restricts to start computing $M_{i+1}[k]$ until $M_i[k]$ is finished. But as compared to Fig. 6.1, the computations of $D_i[k]$ and $I_i[k]$ are removed and hence $M_i[k]$ no longer depends on these computations, which gives an opportunity to accelerate the computation.

Table 6.1: Performance of HMMER in GCUPS on a Quad-core Intel Xeon machine

HMMER	HMM Profile Size N			
	61	84	119	255
V2	≈ 0.03	≈ 0.03	≈ 0.03	≈ 0.03
V3-noSSE	0.3	0.26	0.3	0.37
V3-SSE	3.4	4.3	6.7	10.3

In addition to this filtering step, both the P7Viterbi and the MSV algorithms have also been redesigned to operate on short wordlengths (8 bits for MSV and 16 bits for P7Viterbi), so as to fully benefit from the SIMD extensions (SSE, Altivec) available on all Intel/AMD CPUs. The SSE allows up to 16 simultaneous operations for MSV and 8 operations for P7Viterbi, to be computed using 128-bit vectors. Similarly, row-based-shift and horizontal-*max* operations reduce expensive data shuffling by aligning the data for the diagonal dependencies and performing the *max* operations for the multiple values stored within a single register respectively. These optimizations enable the new HMMER3 to run about as fast as BLAST, slightly faster than WU-BLAST and somewhat slower than NCBI BLAST [Edd09].

Table 6.1 shows the performance in GCUPS for the PfamB.fasta database on a Quad-core Intel Xeon machine. These results show that the combination of the MSV pre-filtering stage with SIMD has a huge impact on the overall software performance, which is improved by a factor of more than 100, and makes most previous FPGA based accelerations slower than any recent Quad-core CPU machine, as shown by Table 6.2.

Table 6.2: Reported average performance for previous FPGA implementations of HMMER2

	Min GCUPS	Max GCUPS
Simplified Viterbi Implementation		
T. Oliver [OSJM06]		5.3
Benkrid [BVK08]		5.2
Exact Viterbi Implementation		
T. Oliver [OYS07]		0.7
Derrien [DQ07]	0.64	1.8
Speculative Viterbi Implementation		
Takagi [TM09]	0.78	7.38
Y Sun [SLG+09]	0.28	3.2

6.3.5 Accelerating the Complete HMMER3 Pipeline

As shown in Fig. 6.5, because the MSV algorithm is used as a prefiltering step, the P7Viterbi algorithm still contributes in a non-negligible way to the execution time. In other words, significantly improving the global execution time cannot be done by only accelerating the MSV kernel alone, and there is still a need for efficiently accelerating the P7Viterbi algorithm.

In the following section we propose to rewrite both MSV and P7Viterbi algorithms to make them amenable to hardware acceleration. We do so by using a simple reformulation of the MSV equations to expose reduction operations, and by using an adaptation of the technique proposed by Gautam and Rajopadhye [GR06] to detect scans and prefix computations in P7Viterbi. This exposes a new previously unknown level of parallelism in both algorithms. We use the V3-SSE results in Table 6.1 as a baseline for performance comparison with our implementation.

6.4 Rewriting the MSV Kernel

As mentioned earlier, the main computation in the MSV kernel is a dynamic programming algorithm that follows the standard algorithmic technique of filling up one data table (called $M_i[k]$ in this chapter with i as the row index, and k as the column index) together with some other auxiliary variables. The values of the table entries are determined from previously computed entries (with appropriate initializations) using the following formulas:

$$M_i[k] = \mathrm{MSC}[k] + \max \begin{cases} M_{i-1}[k-1] \\ B_{i-1} + t_{\mathrm{BMK}} \end{cases} \tag{6.10}$$

$$E_i = \max_k (M_i[k]) \tag{6.11}$$

$$J_i = \max(J_{i-1} + t_{\mathrm{loop}}, E_i + t_{\mathrm{EJ}}) \tag{6.12}$$

$$C_i = \max(C_{i-1} + t_{\mathrm{loop}}, E_i + t_{\mathrm{EC}}) \tag{6.13}$$

$$N_i = \max(N_{i-1} + t_{\mathrm{loop}}, -\infty) \tag{6.14}$$

$$B_i = \max(N_i + t_{\mathrm{move}}, J_i + t_{\mathrm{move}}) \tag{6.15}$$

It can be observed that the computation of M_i has a diagonal dependency for column M_{i-1} and B_i, where B_i depends on all values of M_{i-1}. In other words, no computation for column M_i can start, before all computations for the column M_{i-1} are computed, which gives a column-wise sequential execution to the algorithm.

On the other hand, all values of a given column M_i can be computed in parallel. Since the computation of E_i consists of a max reduction operation, this can be realized using a max tree computation, as shown in Fig.6.6, thus reducing the latency of the MSV architecture from $\mathrm{O}(N)$ to $\mathrm{O}(\log_2 N)$.

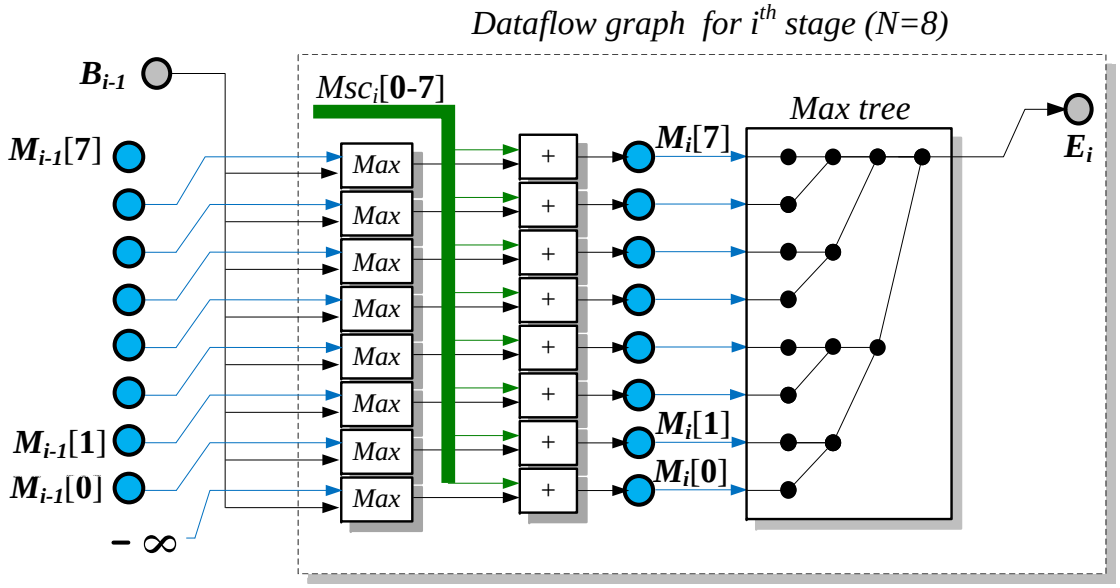


Figure 6.6: Dataflow dependencies for one stage of the MSV filter ($N = 8$) algorithm after rewriting

6.5 Rewriting the P7Viterbi Kernel

As shown in the previous Section, it is easy to rewrite the MSV algorithm recurrence equations so as to expose parallelism in the form of a simple max-reduction operation.

In this Section, we show how it is also possible to use a similar (but more complex) transformation on the P7Viterbi kernel. Here again, the goal is to get rid of the current inherent sequential behavior caused by the so-called *feedback loop*. To do so, we replace the accumulation along the k index for one of the variables by a prefix-scan operation and replace the feedback loop by a simple max-reduction operation. This transformation leads to a modified dependence graph which is much better suited to a parallel hardware implementation. In the rest of the chapter, we express control states collectively as X, to emphasize the main part of the model:

$$M_i[k] = f_M(M_{i-1}[k-1], I_{i-1}[k-1], D_{i-1}[k-1], X_{i-1}) \tag{6.16}$$

$$I_i[k] = f_I(M_{i-1}[k], I_{i-1}[k]) \tag{6.17}$$

$$D_i[k] = f_D(M_i[k-1], D_i[k-1]) \tag{6.18}$$

$$X_i = f_X(\max_k(M_i[k] + E[k])) \tag{6.19}$$

The above equations are a simplified form of equation (6.1)-(6.4), highlighting dependencies on dynamic computations. The key observations concerning P7Viterbi formulas (6.16-6.19) are that

- there is a chain of dependences in the increasing order of k in computing the values of D in any column;
- to compute the X for any column, we need *all* the values of M of that column, each

of which needs a D from the previous column;

– Finally, the value of X of a column is needed to compute *any* M in the next column.

Because of above the observations, there seems to be an inherent sequentiality in the algorithm, as noted by all previous work on this problem.

6.5.1 Finding Reductions

We now develop an alternate formulation of equations (6.16)-(6.19) so that there is no such chain of dependences, thus enabling scalable parallelization of the computations on a hardware accelerator.

More specifically, we show that equation (6.18) computing D can be replaced by a different equation in which such dependences either do not exist, or can be broken through well-known techniques. For our purposes, we shall focus on the function f_D of equation (6.18), which is defined more precisely as follows:

$$D_i[k] = \begin{cases} k = 1 & : \quad M_i[0] + TMD[0] \\ k > 1 & : \quad \max(D_i[k-1] + TDD[k], M_i[k-1] + TMD[k-1]) \end{cases} \tag{6.20}$$

In order to emphasis the self-dependency of D_i, we can represent other variables as inputs, and thus equation (6.20) can be abstracted as follows:

$$D_i[k] = \begin{cases} k = 1 & : \quad a_0 \\ k > 1 & : \quad \max(D_i[k-1] + b_{k-1},\ a_{k-1}) \end{cases} \tag{6.21}$$

Now, if B is zero, the equation is a simple *scan computation* (also called *prefix computations*) $D_i[k] = \max_{i=1}^{k} a_i$.

How to efficiently and scalably parallelize such scan computations is well-known [LF80]. However, if $B \neq 0$, the solution is not at all obvious. We show below how to obtain a scan-like structure for this case. If we expand out the individual terms, we see that:

$$\begin{aligned} D[1] &= a_0 \\ D[2] &= \max(a_0 + b_1, a_1) \\ D[3] &= \max(\max(a_0 + b_1, a_1) + b_2, a_2)) \\ &= \max(a_0 + b_1 + b_2, a_1 + b_2, a_2) \\ D[4] &= \max(a_0 + b_1 + b_2 + b_3, a_1 + b_2 + b_3, a_2 + b_3, a_3) \\ &\vdots \\ D[k] &= \max(a_0 + b_1 + b_2 + b_3 \ldots b_{k-1}, a_1 + b_2 + b_3 \ldots b_{k-1}, \\ &\qquad a_2 + b_3 \ldots b_{k-1}, \ldots a_{k-2} + b_{k-1}, a_{k-1}) \end{aligned}$$

The last term can be written more visually as

$$D[k] = \max\left(a_{k-1}, \max\left(\left(\begin{array}{r} b_1+b_2+b_3+\ldots+b_{k-1} \\ b_2+b_3+\ldots+b_{k-1} \\ b_3+\ldots+b_{k-1} \\ \vdots \\ b_{k-1} \end{array}\right) + \left(\begin{array}{c} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{k-2} \end{array}\right)\right)\right)$$

or more compactly:

$$D[k] = \max\left(a_{k-1}, \max_{j=1}^{k-1}\left(a_{j-1} + \sum_{i=j}^{k-1} b_i\right)\right) \tag{6.22}$$

In (6.22), one can easily identify a reduction operation over vector b and a max-prefix over this reduction operation. But the reduction operation still depends on the inner loop index and we would like to get rid of it. To do so, we add and subtract a same term which does not effect the computation and yields following expression:

$$D[k] = \max\left(a_{k-1}, \max_{j=1}^{k-1}\left(\right.\right.$$
$$\left.\left. a_{j-1} + \sum_{i=j}^{k-1} b_i + \sum_{i=1}^{j-1} b_i - \sum_{i=1}^{j-1} b_i\right)\right)$$

The term $\sum_{i=1}^{k-1} b_i = \sum_{i=j}^{k-1} b_i + \sum_{i=1}^{j-1} b_i$ is independent of j, so it can be moved out of the max:

$$D[k] = \max\left(a_{k-1}, \left(\sum_{i=1}^{k-1} b_i + \max_{j=1}^{k-1}\left(a_{j-1} - \sum_{i=1}^{j-1} b_i\right)\right)\right)$$

Let $b'_{j-1} = \sum_{i=1}^{j-1} b_i$. We note that b'_{j-1} is a scan of the b input, so

$$\begin{aligned} D[k] &= \max\left(a_{k-1}, b'_{k-1} + \max_{j=1}^{k-1}(a_{j-1} - b'_{j-1})\right) \\ &= \max\left(a_{k-1}, b'_{k-1} + \max_{j=1}^{k-1} a'_{j-1}\right) \end{aligned} \tag{6.23}$$

where $a'_j = a_j - b'_j$ is the element-wise difference of a and b'.

Now, the inner term is a max-scan of the a' vector. Hence the $D[k]$, as specified in (6.23), can be computed in parallel using the following steps.

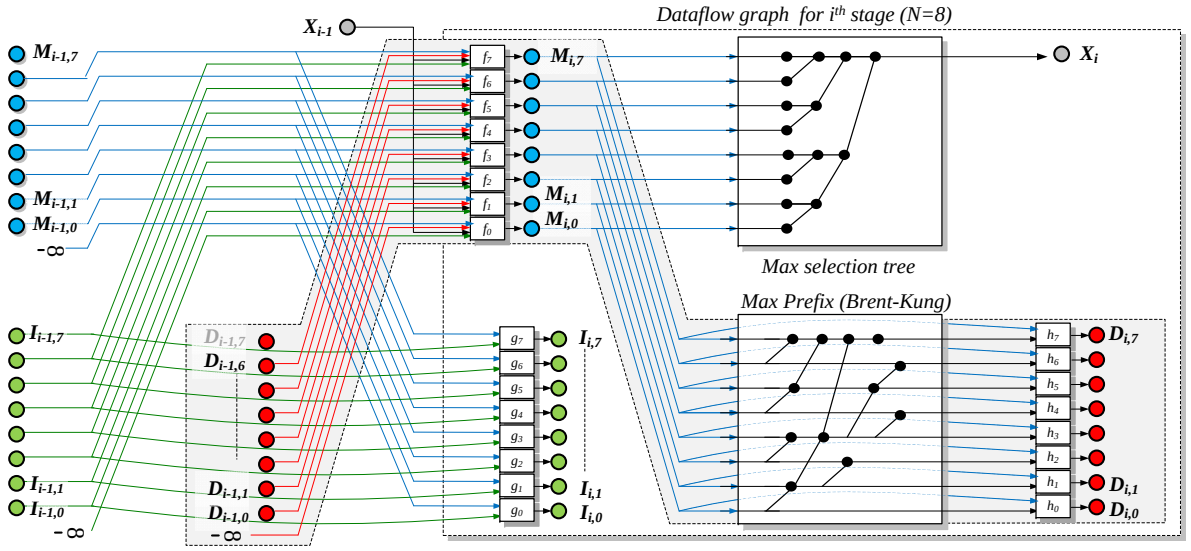


Figure 6.7: Dataflow dependencies for one stage of the P7Viterbi ($N = 8$) algorithm after rewriting. The dependency $D_{i,k} \longrightarrow D_{i,k-1}$ in equation (6.3) is converted to a *max-prefix* block, reducing the critical path from $O(N)$ to $O(\log_2 N)$ operations.

Step 1. Compute, b' the sum-prefix of the array b i.e. $\sum_{i=1}^{j-1} \text{TDD}[i]$, Eq. 6.3. Note that in the Viterbi algorithm, b' needs to be computed only once since TDD is an input.

Step 2. Compute a', the element wise subtraction of b' from a, where $a_{j-1} = M_i[j-1] + \text{TMD}[j-1]$, Eq. 6.20.

Step 3. Compute a'', the max-prefix on a'. The max-prefix computation can be parallelized perfectly and scalably.

Step 4. Add b' element wise to a'' and compare (again element wise) the result with the a input, retaining the larger one. This yields D, the desired answer.

We have rewritten the dependency $D_i[k] \longrightarrow D_i[k-1]$, and now the vector D can be computed in parallel by the above steps, where the computation path is converted into a max-prefix network rather than a strict intra-column dependency.

6.5.2 Impact of the Data-Dependence Graph

To help the reader understanding the benefits of this rewriting transformation, we provide in Fig. 6.7 an illustration of the data dependence flow in the rewritten algorithm for a small problem size (profile size $N = 8$). In this dataflow graph, functions f_k,g_k and h_k are defined as follows :

$$f_M(w,x,y,z) = \max_4(w + bsc_k, x + TMM_k, y + TDM_k, z + TIM_k) + msc_k[dsq_i])$$
$$g_I(x,y,z) = \max_2(x + TII_k, y + TMI_k) + isc_k[dsq_i]$$
$$h_D(x,y) = \max_2(x + TMD_k, y + y)$$

In these expressions *max* and *sum* correspond to saturated (w.r.t to $-\infty$) *max* and *sum* operations. It can be observed that there is no longer a chain of dependencies along the vertical axis in the data-flow graph, and that the longest critical path is now set by the depths of the parallel *max-tree* and of the parallel *max-prefix* blocks, which is $O(log_2(N))$. Another consequence is that the update operations for $M_{i,k}$, $I_{i,k}$ and $D_{i,k}$ can be executed in parallel for all values of k in the domain $0 \leq k \leq N$. In the next Section, we briefly introduce a wide class of prefix computation and discuss various existing prefix topologies, that can be adopted after rewriting.

6.6 Parallel Prefix Networks

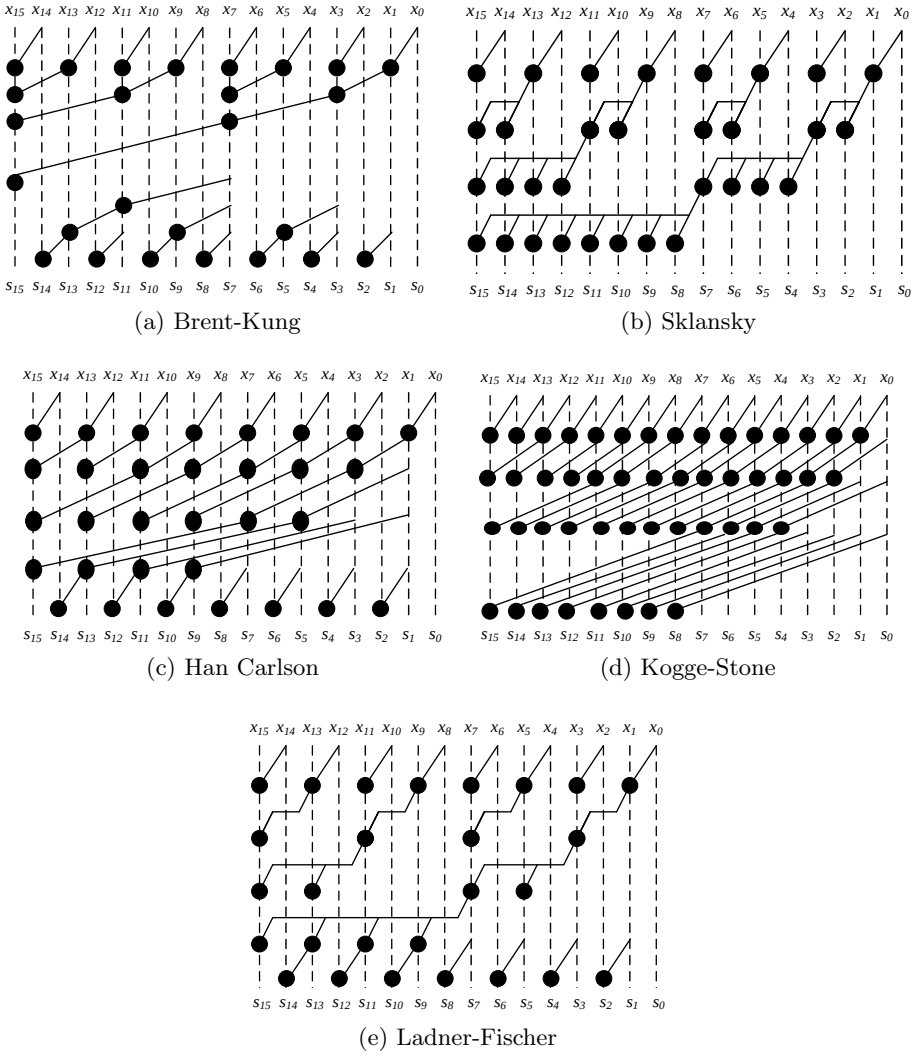
As mentioned in Section 6.5.1, step 3, the rewritten version of the P7Viterbi algorithm exhibits a max-prefix pattern. Prefix computation is a very general class of computations which can be formally defined as follows : given an input vector x_i with $0 \leq i < N$ we define its $\oplus$-prefix vector y_i as :

$$y_i = \bigoplus_{k=0}^{i} x_k = x_0 \oplus x_1 \oplus \ldots \oplus x_i$$

where $\oplus$ is a binary associative operator (and possibly commutative, see [Kno99] for a more detailed definition). Because binary adders fall into this category and since adders form one of the most important building blocks in digital circuits, there is a wealth of research going back almost 50 years dealing with fast (i.e parallel) implementations of prefix adders [LF80, BK82, KS73, HC87, Skl60] using various interconnection networks topologies. Blelloch [Ble90] presents a detailed list of parallel prefix applications in various domains, such as string comparison, polynomial evaluation, different sorting algorithms, solving tri-diagonal linear system and many others.

Figure 6.8 shows some popular prefix network topologies. One of the most important aspects of these network topologies is that they allow the designer to explore the trade-off between speed (i.e. critical path of the resulting circuit), area (number of operators used to implement the prefix operation), and other metrics such as fan-out or wiring length. A comprehensive classification, describing the trade-offs in existing network topologies, has been done in [Har03].

For example, a Brent-Kung [BK82] network computes the prefix in $2\log_2 N - 1$ stages with $2(N-1) - \log_2 N$ operators, while a Sklansky network implements a faster circuit



(a) Brent-Kung (b) Sklansky

(c) Han Carlson (d) Kogge-Stone

(e) Ladner-Fischer

Figure 6.8: Examples of parallel prefix implementation for $N = 16$

($\log_2 N$ stages) at a price of an increase in area ($\frac{N}{2} \log_2 N$ operators). Similarly, Han-Carlson network provides a trade-off between Brent-Kung and Kogge-Stone for the delay and wiring.

By rewriting the algorithmic dependencies of P7Viterbi, we are able to express a naively sequential computation in the form of a max-prefix computation. This rewriting task not only enables us to compute it in parallel, but it also provides us with the ability to utilize the characteristics of existing various prefix network topologies and to explore Speed/Area trade-offs. From Table 6.3, we can see that, D_i can now be computed in ($\log_2 N$) to ($2 \log_2 N + 1$) cycles, instead of N cycles, where the cost range from ($2N - 2 - \log_2 N$) to ($N \log_2 N - N + 1$) operators.

In our case, since the max-prefix computation lies on the critical path, as shown in Figure 6.7, a faster prefix network is desirable. To cause a minimum delay, but a minimal area consumption is also crucial, due to the limited resources of the target platform.

Table 6.3: Characteristics of various parallel-prefix networks

Method	Delay	Cost
Sklansky[Skl60]	$\log_2 N$	$\frac{N}{2}\log_2 N$
Ladner-Fischer[LF80]	$\log_2 N + 1$	$\frac{N}{4}(\log_2 N - 1) + N$
Kogge-Stone [KS73]	$\log_2 N$	$N\log_2 N - N + 1$
Brent-Kung [BK82]	$2\log_2 N - 1$	$2N - 2 - \log_2 N$
Han-Carlson [HC87]	$\log_2 N + 1$	$\frac{N}{2}\log_2 N + \frac{N}{4}$

Hence, it is be interesting to investigate FPGA-based implementations of different prefix topologies and to integrate the most suitable one to our design. In the next chapter, we present several implementations of these network topologies for different size of N, we compare their performance on FPGA, and select one of them, after hardware mapping optimizations, for our final implementation.

6.7 Conclusion

HMMER is a widely used tool in bioinformatics for sequence homology searches. The data dependencies of HMMER kernels, namely MSV and P7Viterbi, lead to a pure sequential execution. All previous attempts to parallelize HMMER either exploit other sources of parallelism such as independent calls to the kernel functions, simplify the algorithm to approximate the computation, or use other techniques such as speculation.

In this chapter, we have proposed an original parallelization scheme for the new HMMER3 profile based search software, which leverages on a rewriting of the compute-intensive kernels inside HMMER, in order to transform the intra-column dependencies into reduction and prefix scan computation patterns without modifying the semantic of the original algorithm. The modified algorithm allows us to exercise the plenty of research efforts already done in the domain of parallel prefix networks, to explore speed and area trade-off, and to implement a faster HMMER application.

7

Hardware Mapping of HMMER

The rewritten version of MSV and P7Viterbi in Chapter 6 exposes a significant amount of parallelism. However, the feed-back loop dependency still exists and we cannot start the computation of column $i+1$, before finishing all computations of current column i. The critical path is shortened to $O(\log_2 N)$ from N, but for larger value of N, the delay can slow-down the circuit.

In this chapter, we discuss the hardware implementation of the rewritten algorithms and we present various implementation schemes. Specifically, this chapter shows the following contributions.

- First, we propose several fine-grain parallelization strategies for efficiently implementing this improved algorithm on an FPGA-based high performance computing platform and we discuss the performance that we obtain.
- Besides exploring fine-grain parallelism opportunities inside each computational kernel independently, we propose a system-level design approach, where the computational kernels are connected in an execution pipeline. We implemented these designs and present speed-up results.

We propose two system-level implementations strategies.

- A straightforward pipeline strategy that connects the computation kernels (MSV and P7Viterbi) through a filter. The coarse-grain parallelism is employed through multiple independent pipelines.
- The second pipeline strategy utilizes more efficiently the filtering characteristics. We implement a single a aggregated pipeline, instead of several disconnected pipelines, and enables load balancing throughout the execution path, among several parallel pipelines.

The profile size for an HMMER database may vary between 50 to 650, and such differences in profile sizes cause a performance degradation due to the fixed size of the hardware design.

We handle this issue by creating a library of preexisting configurations and by loading the optimal configuration for a given profile size.

This chapter is organized as follows. Section 7.1 presents various hardware mapping opportunities. Section 7.2 discusses how various hardware mapping techniques are implemented through High-Level Synthesis tool. In Section 7.3 we present the speed/area performance results for each individual blocks and also for our system-level implementations. Conclusion and future work directions are drawn in Section 7.4.

7.1 Hardware Mapping

Even though the rewritten versions of both the MSV and P7Viterbi algorithms exhibit a significant amount of *hidden* parallelism, deriving an efficient architecture from the modified dataflow graph is not straightforward. In this section we address the different challenges involved in this architectural mapping. We first start by discussing efficient hardware implementations of parallel prefix operations as needed by P7Viterbi, and we present two transformations (namely C-Slow and tiling) that we use to improve the architecture efficiency.

7.1.1 Architecture with a Single Combinational Datapath

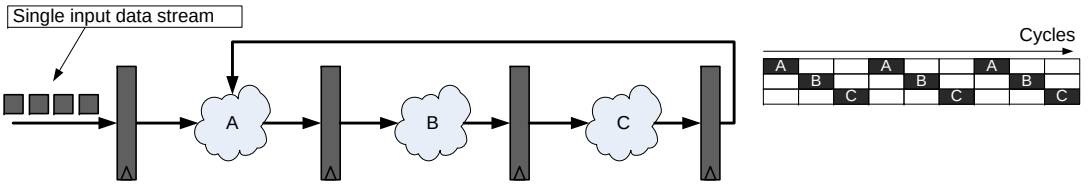
It can be easily seen from Figure 6.7 and Figure 6.6, that in both MSV and P7Viterbi, it is not possible to pipeline the execution of consecutive stages —all the results of the i^{th} stage are needed before any value in the (i+1)th stage can be computed.

As a consequence, and in spite of the fact that we replaced in both cases the initial chain of dependence of $O(N)$ operations by a chain of $O(\log_2(N))$, large values of N may induce a long critical path, which could lead to a poor clock frequency.

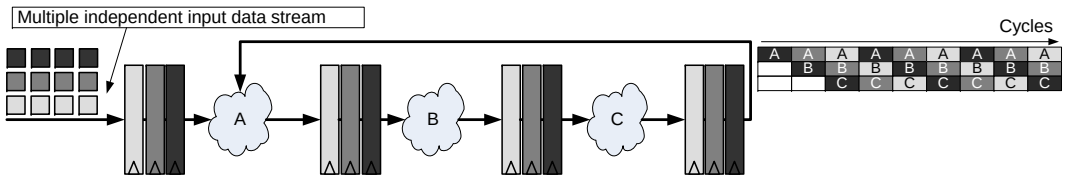
7.1.2 A C-slowed Pipelined Datapath

To obtain a datapath with better clock speed, pipelining is always an adequate choice. Pipelining a datapath without feedback loop results in fast and efficient designs. However it becomes ineffective when there is a feedback loop in the pipelined datapath. Figure 7.1a illustrates this situation. One can observe that the pipeline is never completely filled, due to the dependence of the logic block 'A' over results of the logic block 'C'. A new sample waits until the results of all previous sample are calculated, hence only one logic block is active at any clock cycle.

As the HMMER hardware will be always executed for a set of independent sequences, a wise choice is to input interleaved sequences to the pipeline, after slowing down the pipeline-rate by a factor of C. The resulting architecture is shown in Fig.7.1b. This method costs extra registers (i.e. $(\mathrm{C}-1)\times$stages). But in return, our architecture becomes as efficient as a normal pipelined architecture without a feedback loop. The same solution



(a) Pipelining without C-slow: The pipeline with a single input data stream and a feedback loop from end to the start will not be completely filled at any time instance due to dependency on previous results and at each cycle only one logic block will be active.



(b) Pipelining with 3-slow C-slow: In presence of a feedback loop in a pipeline, several (precisely equal to number of pipeline stages inside the feedback loop) independent interleaved input data streams make efficient use of the pipeline hardware. Now all logic blocks are active at each cycle.

Figure 7.1: Impact of pipelining with C-slow in presence of a feedback loop

to improve the throughput of the hardware implementation has been used by Derrien and Quinton [DQ07], and also by Oliver et al. [OYS07].

On a loop representation of such a calculation, this transformation amounts to add an additional outer loop iterating over independent instances of the algorithm, and then perform a loop interchange so as to move this parallel outer loop to the innermost level and to implement the multiple independent instances on a pipelined hardware in parallel.

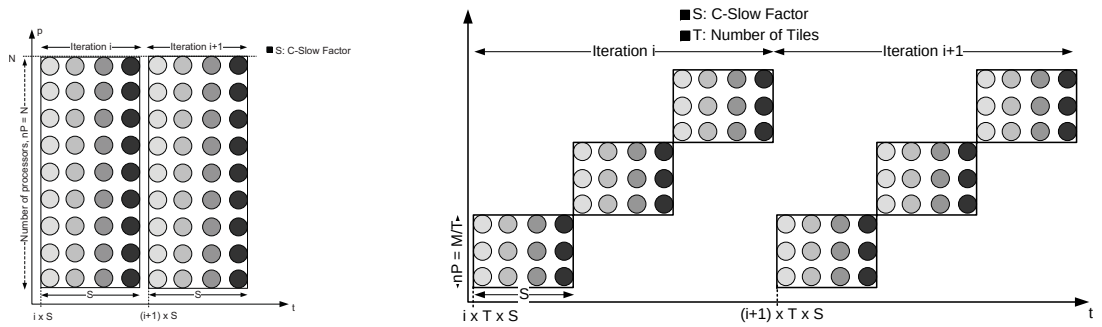
Using this idea, and assuming that S independent instances are to be interleaved, the i^{th} stage only depends on the computation that was executed $i - S$ stages ago. This extra delay can then be used to pipeline the stage execution, as depicted in Fig. 7.2a.

This of course includes the use of additional memories, as we must replicate all registers/memories in the architecture; but because the critical path remains $O(\log_2 N)$, we only need a reasonably small slowing factor, S, to achieve the maximum throughput (as compared to $S \approx O(N)$ in the approach of Derrien and Quinton).

7.1.3 Implementing the Max-Prefix Operator

We have shown in the previous chapter that the max-prefix computation is part of the critical path. Although we need a fast network to reduce this critical path, resource minimization is also crucial in our case, since a smaller kernel block allows one to accommodate more coarse grain parallelism). It would be very interesting to see how much a faster network, such as Sklansky or Kogge-Stone, can speed up our system, and in what manner a slow network, such as Brent-Kung, helps to reduce the resource consumption. Since the presence of a C-Slow pipelined architecture allows also the prefix network to be pipelined, we can hide the delay caused by the extra cycles required by a slow prefix network and can still benefit from the resource saving offered by such network.

Another important aspect is that most of the algorithmic explorations in the domain



(a) Viterbi with C-slow: A Viterbi kernel with S pipeline stages is receiving S independent interleaved data streams, to overcome the feedback loop effect. The required number of processors is N, the profile size.

(b) Viterbi with C-slow and tiling: A Viterbi kernel implemented with S pipeline stages and S interleaved independent data inputs on N/T processors through tiling.

Figure 7.2: Viterbi kernel implementation with simple C-Slowed pipeline and Tiled C-slowed pipeline

of prefix networks were in a context where the operator was extremely fine grain—just a few Boolean gates, as in a half- or full-adder. Despite the fact that our computation scheme is based on the same prefix patterns as binary adders, our situation differs in two ways :

- The basic operation is not a bit-level but a more complex word-level operation (namely max).
- The size of the prefix can be very large (up to 256 input elements) which poses scalability issues in terms of routing.

To the best of our knowledge there has been no systematic study of FPGA implementations of prefix computations. One reason is that the typical use of such circuits would be in adders, where high-speed carry circuits are already provided by FPGA vendors, and there are few applications that need coarse-grain, word-level operators. For the HMMER application, we implemented a number of the max-prefix as well as max-reduce architectures. The performance comparisons are reported later in Section 7.3.2.

7.1.4 Managing Resource Constraints through Tiling

Both MSV and P7Viterbi dataflow graph sizes scale linearly[1] with the target HMM profile size N. For large values of N e.g., $N > 100$, the straightforward mapping of the complete dataflow graph to a hardware datapath quickly becomes out of reach of most FPGA platforms.

However, since the computational pattern of both algorithms exhibits a lot of regularity, it is possible to apply a simple *tiling* transformation, which separates each dataflow of size

[1]The scaling is linear for the Brent-Kung architecture that we implemented. For the Ladner-Fischer architecture, the resource usage grows as $n \log n$

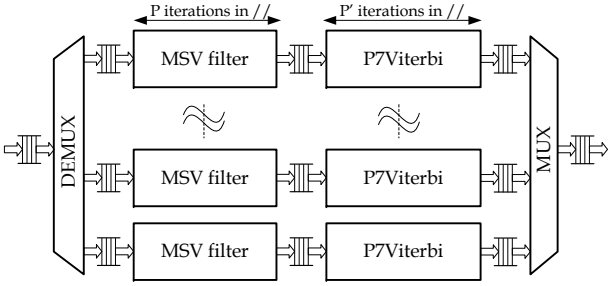


Figure 7.3: System Level Implementation: First Approach

N into P partitions (tiles), each of them calculating N/P consecutive values of the current column. This transformation, and its impact on the scheduling of the computations is depicted in Fig. 7.2b. In the case of the MSV, the partitioned datapath should implement a N/P reduction max operator, whereas in the case of P7Viterbi, we need a N/P max prefix operation.

As a summary, the characteristics of various designs that we explored are listed in Table 7.1. It can be concluded that in our case, optimal throughput of $O(N/P)$ can be obtained by combining tiling and c-slowing techniques.

Table 7.1: A summary of the different architectural solutions, along with their space-time characteristics

Method	Area	T_{clk}	Through-put
Combinational	$O(N)$	$O(\log_2 N)$	$O(\frac{N}{\log_2 N})$
Tiled	$O(N/P)$	$O(\log_2 \frac{N}{P})$	$O(\frac{N/P}{\log_2 \frac{N}{P}})$
C-slow	$O(N)$	$O(1)$	$O(N)$
Tiled + C-slow	$O(N/P)$	$O(1)$	$O(N/P)$

7.1.5 Accelerating the Full HMMER Pipeline

As mentioned in section 6.3.5, improving the global performance requires that both MSV and P7Viterbi are accelerated in hardware. This can be done by streaming the output of the MSV to the input of the P7Viterbi, so as to map the complete HMMER3 pipeline to hardware. Special care must be given to the C-Slow factor of both accelerators, which must be the same to avoid a complex data reorganization engine between the two accelerators.

In addition, depending on available resources, it is even possible to instantiate several HMMER3 pipelines in parallel, as illustrated in Fig. 7.3. However, in order to optimize the hardware resource usage, we must also ensure that the pipeline workload is well distributed among the hardware accelerators. Let us quantify the total algorithm execution time, T_{total} when the two task executions are pipelined, we have :

$$T_{\text{total}} = \max\left(T_{\text{msv}}, \alpha T_{\text{Viterbi}}\right) \tag{7.1}$$

where T_{MSV} and $T_{Viterbi}$ correspond to the average algorithm execution times, and where α is the filtering selectivity. Optimizing the performance therefore means ensuring that the

raw performance (in GCUPS) of the P7Viterbi accelerator is able to sustain the filtered output of the MSV accelerator, that is, its performance should be at least $1/50^{th}$ that of MSV, i.e. the filtering percentage of MSV in Fig. 6.5. Using this constraint, we can then define a set of pipeline configurations, by choosing distinct tiling parameters (i.e. partition sizes) for P7Viterbi and MSV such that the level of parallelism exposed in MSV is at least 50 times that of P7Viterbi.

7.2 Implementation through High-Level Synthesis

Our design flow leverages high-level synthesis through a commercial C to Hardware compiler (Impulse CoDeveloper C-to-FPGA) combined with the GeCos [rg] framework, a semi-automated source-to-source compiler targeted at program transformations for high level synthesis. The combined use of these two tools allowed us to explore a very large architectural design space in a very reasonable amount of time. In this section we explain how various hardware mapping techniques are implemented through Impulse C, we discuss the challenges raised by the HLS tools and the solutions that we adopted to generate an efficient hardware design.

We have seen in section 7.1.2 that due to the C-Slow pipelining transformations, we now have a triply nested loop $[j, i, k]$, instead of the previous doubly nested $[i, k]$ loops, for MSV and P7Viterbi described in section 6.2.2 and section 6.4. Since index j corresponds to independent input sequences, this loop can be executed in parallel on independent PEs. Similarly, as shown in Figure 6.7, calculations along index k can also be executed in parallel. However for large profile size, the hardware resources may not be sufficient.

7.2.1 Loop Transformations

In our final implementation of MSV and P7Viterbi, we perform a loop interchange to interleave independent sequences to be processed on the same PE, i.e. $[j, i, k] \rightarrow [i, j, k]$. For MSV, we implement a fully parallel loop k. However in the case of P7Viterbi, a complete parallel implementation of loop k cannot be implemented on a single chip. So we perform a strip-mining on k axis to transform our loop nest to $[j, i, k', k'']$, where k'' is the parallel loop. After having an inner-most parallel loop, we would like to pipeline the rest of the loop nest. However, since Impulse C allows only the inner-most loop to be pipelined, the architecture will experience a repeated behaviour of pipeline start and end, and since the inner loop count is smaller than the outer one, the pipeline stays most of the time in the *epilog* and the *prolog* state. In order to avoid this situation, we need to coalesce the rest of the loop nest, $[i, j]$ for MSV and $[i, j, k']$ for P7Viterbi, to have a regular and unbroken pipeline (i.e. the *prolog* of the next outer loop iteration overlaps the *epilog* of the current loop iteration). Since Impulse C does not support any of these loop transformations, we have to apply these transformations through GeCos or by a manual rewriting of the code.

7.2.2 Loop Unroll & Memory Partitioning

One big challenge for designing with Impulse C is the limitation of the loop unrolling feature. Impulse C can only perform full unrolling of the inner-most loop and does not support a partial unrolling, which is required for P7Viterbi. Although for MSV we can use this automated full unrolling of loop k, another constraint limits the design efficiency, i.e. the limitation of Impulse C on memory partitioning. Impulse C can only scalarize an array completely to variables, and does not support automatic memory partitioning into splitted blocks or interleaved blocks, as discussed in section 5.3.9. Since HMMER kernels need to access a lot of profile data, it is not a wise choice to map the profile database on logic cells, as it also makes the design more complex with muxes to access the right register. Thus, the limitation on memory partitioning does not allow us to use the automated loop unrolling and we have to implement both transformations (i.e. loop unroling and memory partitioning) manually.

7.2.3 Ping-Pong Memories

In our implementation, read/write accesses to memory locations in consecutive cycles belong to independent input sequences, and hence a data being written in cycle t will be read in cycle $t + S$, where S is the slowing factor. These memories are being accessed in circular manner, and a data read and write do not have any dependency. However, Impulse C compiler conservatively imposes dependency on such accesses and does not allow parallel read and write accesses, which will result in an increase in the pipeline rate. In order to cope with this constraint, we use ping-pong memories where we read and write to these memories on alternate cycles (i.e. every cycle read from one memory and write to another and vice versa). The use of ping-pong memories duplicates the required memory resources, but it also helps to improve the design throughput. On the other hand, since the C-Slow factor is very small (the size of a memory), the duplication of memories does not effect heavily the resource management.

7.2.4 Scalar Replication

In a single cycle, if a same memory location is being accessed several times (without any intermediate write back to the same location), it would be better to store the first read in a local register and reuse this memory access to avoid any increase memory latency. The Impulse C compiler does not support automatic scalar replacement of such multiple memory accesses. We implemented scalar replacement manually, by reading the memory cell to a register, at first usage, and use this register for later accesses. Similarly multiple memory write accesses are combined to a single access during the last write operation, and intermediate operations write to a register.

7.2.5 Memory Duplication

In presence of a tiled execution of size T, a read/write diagonal dependency results in reading data sets from location $[i, \ldots, i+T]$, computing and writing back to locations $[i+1, \ldots, i+T+1]$, where the location $[i+T+1]$ is accessed during the next cycle to read the previous data. Hence, a straightforward write operation will corrupt the data. One solution to this problem can be to duplicate the complete memory and access them in a ping-pong manner. A better choice is to duplicate only the tile corners, and to perform the write operation to the duplicated cell, which can update the original corner location with a delay of one cycle.

7.3 Experimental results

In this section we provide an in-depth quantitative analysis of our proposed architectures, and we compare their performance with that of a state of the art software implementation of HMMER3 on a CPU using the SSE SIMD implementation.

Our target execution platform consists in a high-end FPGA accelerator from Xtreme-Data (XD2000i-FSBFPGA) which has already been successfully used for implementing bioinformatics algorithms [ACL+09]. This platform contains two Stratix-III 260 FPGAs, a high-bandwidth local memory (8.5 GBytes/s) and a tight coupling to the host front side bus through Intel Quick Assist technology, providing sustained a 2 GBytes/s bandwidth between the FPGA and the host main system memory.

The rest of this section is organized as follows: we first make a quantitative analysis of speed/area results for the MSV accelerator, then we address the mapping of the max prefix network implemented on FPGA along with P7Viterbi implementation results. Finally, we discuss the system-level performance and compare the performance of our approach with that of an hypothetical, state-of-the-art GPU implementation.

7.3.1 Area/Speed Results for the MSV Filter

Table 7.2 summarizes the area and speed of MSV hardware accelerators for different values of N and S (the MSV accelerator does not need tiling as for all profile sizes, it fits in the FPGA). It can be observed, that even though we use a C-to-hardware high-level synthesis tool, we are able to achieve remarkably high operating frequencies (up to 215 MHz). When compared[1] with Table 6.1, these results indicate a speedup for a single accelerator varying between 3× to 6× depending on the profile size, N.

7.3.2 Area/Speed Results for Max-Prefix Networks

As mentioned in Section 7.1, the P7Viterbi implementation uses a parallel max prefix scheme, for which many implementations exist. As this computational pattern is at

[1] This is an rough approximation, as we should also account for the time spent by the software in P7Viterbi (50% of the total execution time)

Table 7.2: Speed and resource usage of a single MSV kernel hardware implementation

N	C-Slow (S)	Logic Util.	M9K	MHz	GCUPS
64	7	10k / 5%	66 / 8%	215	14
128	8	19k / 9%	130 / 15%	201	26
256	9	37k / 19%	258 / 30%	175	45
512	10	69k / 34%	513/60%	160	82

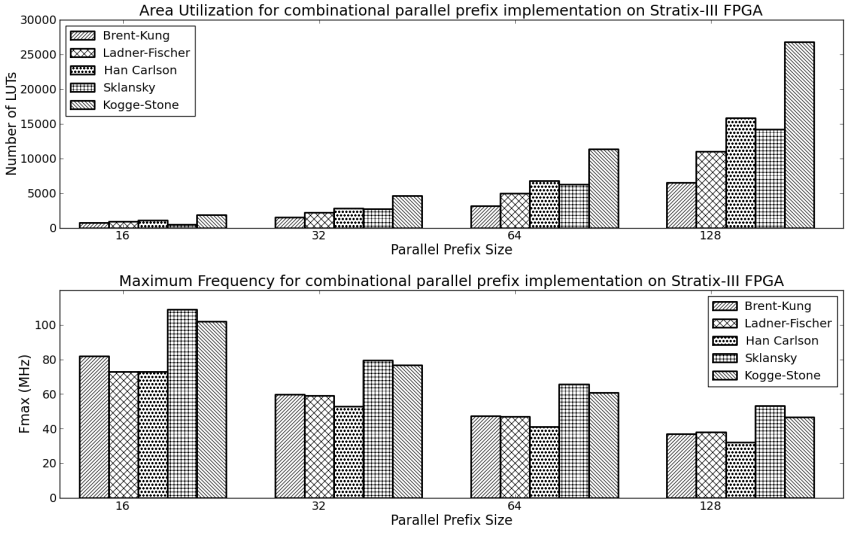


Figure 7.4: Speed/Area results for combinational parallel max prefix implementations on Stratix-III FPGA

the core of the modified algorithm, we explored several alternative implementations to experimentally quantify their respective merits with respect to an FPGA implementation. We used an in-house Java based RTL generator, to generate these network topologies.

The results provided in Fig. 7.4 show that for large values of N, so called *fast* implementations of parallel prefix such as Kogge-Stone or Ladner-Fischer provide only marginal speed improvements with respect to the Brent-Kung architecture. This can easily be explained by the long wires used in the first two approaches, which make the routing much more challenging on an FPGA. For our implementation, we decided to use the Brent-Kung architecture due to its minimal resource utilization and we increased the speed by pipelining all stages in the network.

7.3.3 Area/Speed Results for the P7Viterbi Filter

Table 7.3 summarizes the area and speed of P7Viterbi hardware accelerators for different values of N, S and P. It can be observed, that the rewritten P7Viterbi kernel can deliver quite promising speed with a $\log_2(N)$ C-Slow factor. By fitting multiple instances of P7Viterbi on a chip, it can alone (i.e. not using MSV filter) perform better than earlier implementation of HMMER2, reported in Table 6.2.

Table 7.3: Performance and area for a Single P7Viterbi implementation

P	N	Logic Util.	M9K	MHz	GCUPS
8	64	5.8K / 2.8%	69 / 8%	126	1
8	128	6.8K / 3.3%	128 / 14.8%	124	0.99
16	64	10.1K / 4.9%	112 / 13%	119	1.9
16	256	14K / 6.9%	170 / 19.7%	117	1.87
32	256	28.7k / 14%	332 / 38%	112	3.6

Table 7.4: Speed/Area results for our System-Level implementation

N	P	Logic Util.	M9K	MLAB	MHz	GCUPS
64	8	21K / 10%	54 / 6%	27Kb	99	7
64	16	26K / 13%	99 / 11%	29Kb	97	7.6
128	8	31K / 15%	57 / 7%	51Kb	94	12.7
128	16	38k / 19%	105 / 12%	55Kb	93	13
512	8	79K / 39%	675 / 78%	66Kb	89	45.7

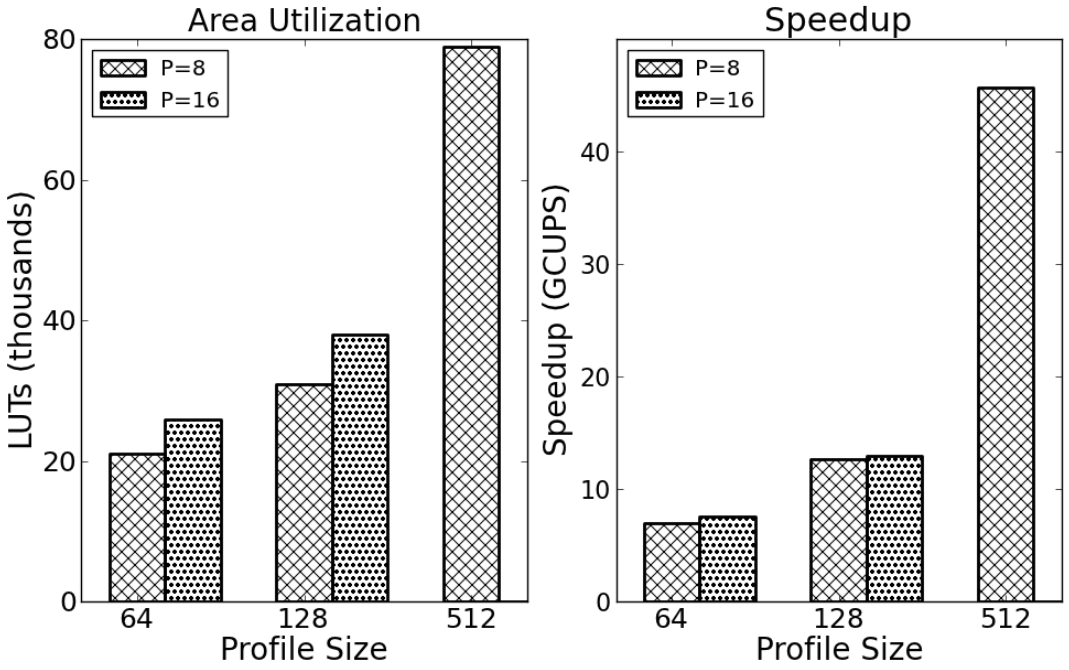


Figure 7.5: Speed/Area results for a single HMMER3 pipeline implementation

7.3.4 System level performance

So far, we have provided area/performance results only for standalone accelerator modules, which should be integrated together in one or several complete HMMER3 computation pipelines.

Following the constraints on pipeline workload balancing, and given the resources available for implementing the accelerator on a chip, we derived a set of pipeline configurations depending on the target profile size N.

These configurations have parameters (C-Slow factor S, Tiling parameter P) chosen to maximize the overall performance. The set of parameters for a given value of N in chosen as follows:

- First, we choose the C-slow factor S to enable fine grain pipelining (i.e. at the operator level) of the MSV accelerator. The same value is then used for the pipelining of the P7Viterbi accelerator.
- Second, we choose the tiling size P so that the P7Viterbi accelerator can sustain the MSV input throughput.

Table 7.4 describes some Quartus place and route data for the pipeline configurations that we derived through this approach. The results presented correspond to a single pipeline implementation. These results show that speedups of up to 4.5×, compared to HMMER V3-SSE in Table 6.1, can be achieved for a single execution pipeline implemented on one out of the two FPGAs of the platform.

By implementing multiple HMMER3 pipelines we should be able to also improve the overall speed for smaller profile sizes (e.g. 64 and 128). Indeed, more than 9× speedups could be achieved compared to our baseline QuadCore SSE implementation for smaller profile sizes. By using both FPGAs on board, we could double the number of HMMER3 pipelines running on the XD2000i platform. However due to firmware and device driver limitations, only one of the two FPGAs can be used at a time.

7.3.4.1 Discussion

For now, we followed a typical approach for system-level implementation, i.e. keep all pipelines independent of each other and connect each MSV block to its own P7Viterbi block, as shown in Fig.7.3. At the end, results are collected from all P7Viterbi blocks.

However, this approach lacks load balancing after the MSV filters. The initial demultiplexer distributes sequences evenly between the available MSV blocks, but there is no guarantee that the following P7Viterbi blocks will also receive the same amount of inputs to process and this might result in an inefficient use of the P7Viterbi units.

Similarly, while increasing coarse-grain parallelism, a complete HMMER3 pipeline needs to be instantiated, which might not be possible to fit inside the remaining resources and will reduce the obtainable speedup, due to under utilization of the resources. Similarly, for S input sequences and N existing parallel pipelines, each additional MSV unit reduce

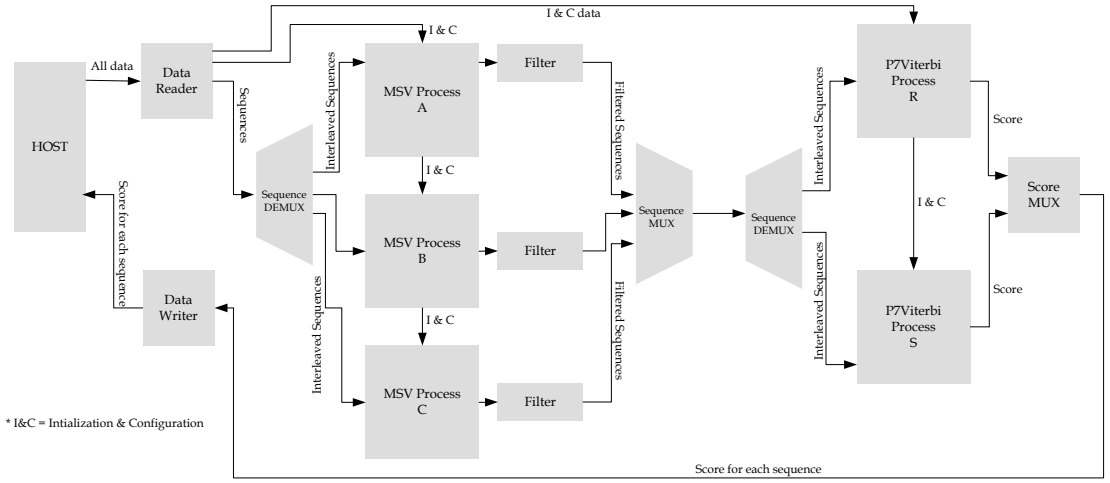


Figure 7.6: System level view of a complete HMMER3 hardware accelerator: The figure shows an HMMER3 pipeline with 5 computation blocks (i.e., 3 `MSV` blocks communicating to 2 `P7Viterbi` blocks). The `Sequence DEMUX` distributes the interleaved sequences between computation blocks. The `Filter` filters out the sequences with scores lower than the threshold. The `Sequnce MUX` receives interleaved sequences and outputs un-interleaved sequences to `Sequence DEUMX`.

S by $S \times 1/N(N+1)$ load for each MSV, while an additional P7Viterbi may only reduce $.02 \times S \times 1/N(N+1)$ load for each P7Viterbi. Hence, an additional P7Viterbi block is not as beneficial as an additional MSV block.

In order to solve these issues, we propose a complete redesign at system-level in the next Section.

7.3.5 A Complete System-Level Redesign

A better way to utilize both available hardware resources and HMMER filtering characteristics is to couple a larger number of MSV blocks to a smaller number of P7Viterbi blocks, as shown in Figure 7.6. However, this design requires a complex filtering step, collecting results and sequences from all MSV blocks, filtering out sequences with results less than the specified threshold, interleaving the rest of the sequences again and distributing them to P7Viterbi blocks. We organize this step into three modules: *Filter*, *Sequence MUX* and *Sequence DEMUX*. A dedicated *Filter* for each MSV block performs the collection of results and filtering step and sends filtered out sequences to the *Sequence MUX*. The *Sequence MUX* receives inputs from all *Filter* blocks, un-interleaves all sequences and sends serialized sequences to the *Sequence DEMUX*, which interleaves the sequences once again and distributes them evenly among the P7Viterbi blocks.

By this approach the *Sequence DEMUX* block works as a load balancing component by collecting the sequences and distributing them evenly between computation components. However, while selecting the number of MSV and P7Viterbi components, Eq. (7.1) should be in consideration to keep the T_{MSV} and the $T_{Viterbi}$ balanced.

Table 7.5 shows the area and speed of the system-level implementation described in

Fig. 7.6. It can be seen that by fitting multiple HMMER3 pipelines, a speedup of 13 × can be achieved compared to QuadCore SSE implementation for profile sizes of 64, and for higher profile sizes we reach a speedup of 5 over the software implementation. Since our FPGA platform is not a recent one, we limit our comparison to a QuadCore machine, instead of an 8-core machine. A fair comparison for an 8-core machine implementation should be against the recent stratix V FPGAs, which contains almost 4 times more resources than a Stratix III.

As the profile size N varies between 50 to 650 [DQ10], the design should be able to handle arbitrary value of N and sustain its speedup. This can be managed by taking advantage of the reconfigurable nature of FPGAs. For a given profile size, we can choose and load the configuration, which best suits that specific profile size from a set of predetermined bitstream configurations. As the *hmmsearch* compares an HMM profile against a database of sequences, the profile reconfiguration is only required in the beginning of a new profile-sequence comparison. To obtain real-life performance measurements, we benchmarked our implementation on representative profile and sequence data sets. The experimentally measured speed-ups were in average within 80% of the place and route estimated results. This discrepancy can be explained by the communication and initialization overhead. The amount of initialization overhead depends on the size of HMM profile to be loaded. However, a large sequence database to be processed following the initialization, makes this overhead negligible.

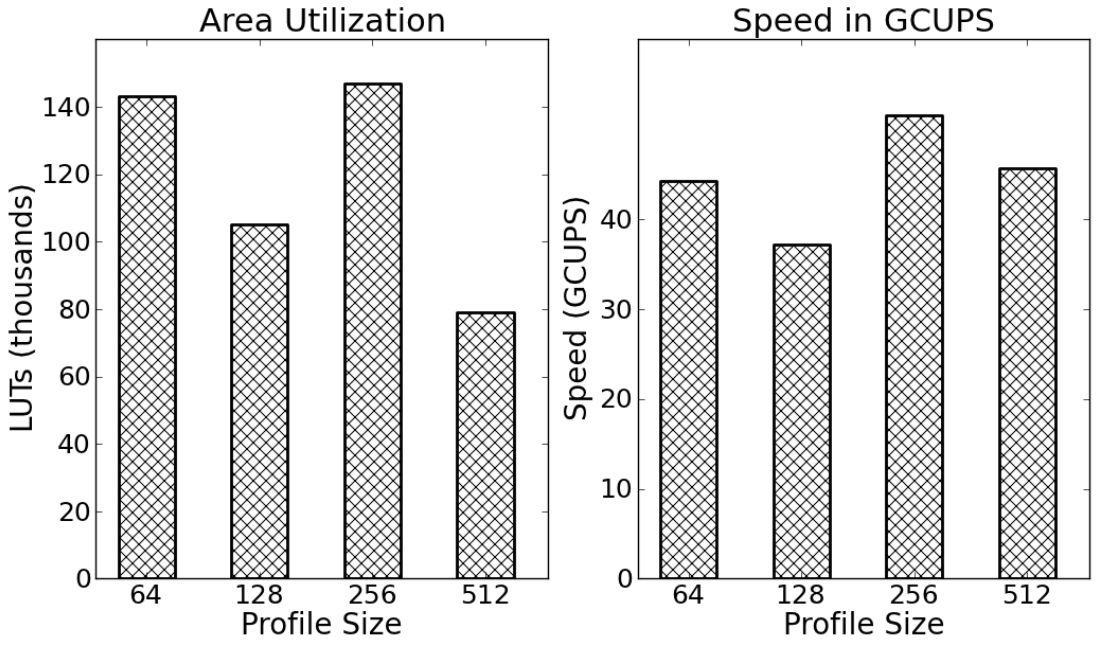


Figure 7.7: Speed/Area results for multiple HMMER3 pipelines implementation

Table 7.5: Performance and area for our System-Level implementation

N	P	MSV	P7Vit	Logic Util.	M9K	MLAB	MHz	GCUPS
64	8	6	1	128K / 63%	864 / 100%	215Kb	103	40.0
64	8	7	1	143K / 70%	864 / 100%	269Kb	97	44.2
128	8	3	1	105K / 52%	864 / 100%	181Kb	95	37.2
256	8	2	1	147K / 72%	864 / 100%	203Kb	99	51.5
512	8	1	1	79K / 39%	675 / 78%	66Kb	89	45.7

7.3.6 Discussion

One question raised by our results is whether an implementation of a complex system-level architecture, such as that of Fig. 7.6, is beneficial or not. The area cost for auxiliary components (e.g. MUX, DEMUX, Sequence Interleaver) becomes much higher than the area cost for such components in independent pipelines. And the area cost for these auxiliary components restricts the overall speedup improvement in comparison with independent pipelines of our first approach. It can be concluded that while implementing complex architecture on hardware, a high implementation effort may result in a marginal speedup compared to a simple architecture involving fewer auxiliary components.

Another important point of discussion is whether our FPGA would actually perform faster than an equivalent GPU implementation. This is an important point as GPU offers more flexibility at a much lower cost than a typical HPC FPGA platform, albeit at a higher cost in power/energy consumption. Unfortunately, there is currently no GPU version of HMMER3 available for comparing the two targets.

We however believe that, contrary to HMMER2, a GPU version of HMMER3 would only offer marginal speedup w.r.t the optimized SSE version. The GPU speedup for HMMER2 were reported in the order of $15 - 35\times$ [WBKC09a] and $20 - 70\times$ [GCBT10] for a single GPU over the software implementation. While the software implementation of HMMER3 gives $340\times$ performance improvement compared to that baseline.

It is a well known fact that very well optimized multi-core software implementations reduce the performance gap to only 2.5× on average against a GPU implementation [LKC+10]. When looking at HMMER3 speedup results (given in Table 6.1), it turns out that the use of an optimized SSE software implementation alone brings up to $27\times$ speedup improvement over the non SSE version, a speed-up somewhat comparable to the GPUs implementation for HMMER2, and which is mostly due to the systematic use of sub-word parallelism, as discussed in section 6.3.4. As GPUs do not have support for short integer sub-word parallelism, it is therefore very unlikely that they will do much better than SSE implementation.

7.4 Conclusion

This rewritten version of HMMER has permitted us to obtain a full and efficient parallelization of the two kernels on hardware. In this chapter, we have combined the new parallelization scheme with an architectural design space exploration stage and we have

presented various fine-grain parallelization and resource management strategies. Finally, we have shown implementation of each kernel individually as well as in combination in the HMMER execution pipeline. After determining the best performing architecture for a given HMMER profile size, by taking into account the amount of available hardware resource and the pipeline workload balance, we have presented two system-level implementation schemes.

An improvement would be to take advantage of data reusability. Because HMMER is often used in a context where a profile database is matched against a sequence database, reducing the bandwidth pressure by using the on-board memory of the accelerator to store part of the input data-set seems an attractive option. The sequence database can be received in the beginning and stored in the on-board memory and it can be utilized for each profile in HMM profile database. Indeed, it can be observed that, for small HMM profiles (say $N < 128$), the combined throughput of a complete system level accelerator (with several pipelines) gets very close to the maximum throughput supported by the board.

8

Conclusion & Future Perspectives

8.1 Conclusion

In recent years, the growing dependence of the biological research and of the pharmaceutical industry on advancements in bioinformatics, and the exponential increase in available biological data to process have urged the development of powerful computational platforms that can sustain the accumulating computational requirements. The resulting platforms can leverage on the advancements in parallel computing.

Among several alternatives, such as multi-core, GPUs, Grid and ASICs, FPGAs appear to be the viable target platform due to their reconfigurability along with speed and power optimality. It is worth to be noted that a power optimized framework can considerably reduce the operating cost, considering a long life cycle.

However the major impediment to the wide spread use of FPGAs is the design cost. Custom circuits on FPGAs are usually described at the Register Transfer Level that provides very little abstraction. Such low level design description results in an error-prone design path. A large part of the design time is usually spend in design verification efforts. Although the resulting design is very efficient, it is architecture specific and shows low code reusability opportunity. Thus, an FPGA-based accelerator, with a long development cycle and resulting into a rigid design, can hardly be helpful for bioinformatics community, which requires computing platforms to be powerful as well as scalable and flexible to the constantly growing requirements.

Fortunately, High-Level Synthesis tools can overcome the problems associated with FPGA design flow. HLS tools transform the long, error-prone design path to an automated, error-free design step, where the designer's job is reduced to provide an abstract functionality of the application and the tool generates a RTL design automatically. Since the abstract specification (usually a `C` program) is truly generic, the design can be

altered easily and can be retargeted to an up-to-date platform. HLS based accelerators thus appear to satisfy the requirements of bioinformatics.

In this research work, we investigated HLS based accelerators for bioinformatics, specifically HMMER, a bioinformatics application well-known for its compute-intensiveness and for its hard to parallelize data dependencies. In order to obtain an efficient design, we first explored the behavior of various HLS tools with different design inputs. It has been widely observed that the generated hardware from an HLS tools, largely depends on the input `C` code. The higher abstraction level of input design results in a huge design space that needs to be explored. Hence, an HLS based designer needs to understand 'what' kind of `C` code will be translated to 'what' kind of hardware circuit. Chapter 5 encompassed the techniques that can lead to an efficient hardware design.

We demonstrated these design practices for an HLS based design on acceleration of HMMER. The computation kernels of HMMER, namely MSV and P7Viterbi are very compute-intensive, and their data dependencies, if interpreted naively, lead to a purely sequential execution. We proposed an original parallelization scheme for HMMER based on rewriting its mathematical formulation, in order to expose hidden potential parallelization opportunities. We discussed the different challenges involved in the architectural mapping and exercised various fine-grain parallelization techniques. In order to take full advantage of available hardware resources, we employed and compared coarse-grain parallelization schemes through different system-level implementations of the complete execution pipeline, i.e. based on either several independent pipelines or on a large aggregated pipeline. Our parallelization scheme targeted FPGA technology, and our architecture can achieve up to 13 times speedup compared with the latest HMMER3 SSE version on a Quad-core Intel Xeon machine, without compromising on the sensitivity of the original algorithm.

This research shows that a decent amount of design efforts (still at abstract level) can turn `C` based hardware development as efficient as a manual RTL design. HLS based acceleration, makes FPGAs very affordable target platforms. The reduced design time can now sustain the fast-paced time-to-market requirements. Similarly, the reduced design efforts allow complex algorithms to be implemented at abstract level, and makes FPGA design flow competitor with that of multi-core and GPUs. Furthermore, the low power consumption of FPGAs along with the faster design time and high design performance can outperform other acceleration platforms.

This research focus on acceleration of bioinformatics application on FPGA using high-level synthesis, and our approach shows promising speedup results, and makes a strong case for using C-to-Hardware tools for FPGA based acceleration. Future challenges and some interesting aspects to be explored, as short-term goals, are as follows:

FPGAs demonstrate considerable performance achievements with low power budget, however there are still many challenges to be addressed for a wide adoption in the high performance computing community. High-Level Synthesis tools answer only partly one major concern, i.e. the programming model is simplified and the HPC designers do not need to program in HDL. However, a hardware design still requires a different way of

thinking and even with a most advanced HLS tool, software developers must be "system-aware" to produce good performance results. Since each HLS tool is equipped with a set of particular design translation techniques, which might not be the same for any other tool, a designer working on several HLS tools will not have a standard level of efficiency on each tool. Hence, the designer should understand the capabilities of each tool, and provide the design specifications accordingly. Since FPGAs are normally used as coprocessors in a hardware-software co-design paradigm, the I/O bandwidth of such design may appear as a performance bottleneck. Similarly the manual mapping to a platform, e.g. HW-SW partitioning, designing interface between hardware and software parts, still obstruct a software designer to perform FPGA based design.

In order to make FPGAs easier for designers to adopt, the HLS tools need to make an extra effort to incorporate techniques related to application analysis and parallelization extraction. In other words, the techniques for an efficient hardware design, discussed in Chapter 5, need to be embedded inside the HLS compilation framework. This will simplify the requirement of specialized skills, as "thinking in hardware", and may attract more software designers to perform hardware-based acceleration. The research efforts for a front-end source-to-source compiler, such as [MKFD11, Ple10, RP08], seems to be an interesting solution to extract parallelization automatically through these tools and feed the output source code to HLS tools.

Similarly, HLS tools need to provide support for more and more FPGA platforms through Platform Support Packages (PSPs), so that the tool can generate architecture specific designs and hardware/software interfaces automatically.

In our research efforts, we have been limited to use one out of two FPGAs on XD2000i. It would be interesting to use both FPGAs, after release of firmware update, and observe how it can sustain with the doubled data band-width requirement. Similarly, our final system-level implementation of HMMER, is an aggregated pipeline. It would be interesting to see the possibility of a single aggregated pipeline, implemented on both FPGAs, so that a single load-balancing unit balance inputs to all P7Viterbi blocks. Besides exploration on FPGA, it would be also interesting to implement a SIMD optimized version of P7Viterbi block on an 8-core host machine and implement only MSV blocks on FPGA. The only concern is to keep the execution time of both kernels balanced, as discussed earlier. Since the SIMD based optimizations can accelerate P7Viterbi significantly, it would be interesting to see how the performance of P7Viterbi on an 8-core machine, can sustain with the performance of MSV blocks on FPGA, while the P7Viterbi needs to deal with only 2% of original data input to the MSV.

The algorithmic rewriting of HMMER, has exposed many parallelization opportunities, which were not visible before. It would be interesting to implement the rewritten HMMER on GPUs to establish a fair comparison with FPGA's performance.

8.2 Future Perspectives

In this research work, we studied the feasibility of using High-Level Synthesis for the acceleration of bioinformatics applications and experimented the optimization techniques on dynamic programming kernels inside HMMER.

Bioinformatics comprises a large class of algorithms based on dynamic programming techniques. Many of these algorithms hold similar or a little varying data dependencies as compared to the P7Viterbi and the MSV kernels. It would be interesting to apply similar techniques, we have learnt, on these algorithms for acceleration. For example, Smith-Waterman and Needleman-Wunsch algorithms hold much simpler data dependencies than P7Viterbi, since there is no feed-back loop causing an inter-column sequential execution. Similarly, the RNA-folding algorithms hold similar, but more complex data dependencies due to non-local memory access. One future perspective would be to accelerate these potential algorithms by experimenting similar techniques we used for HMMER.

The rewriting of HMMER kernels, based on look-ahead computations, helped to expose the hidden parallelism. It would be interesting to use similar rewriting techniques for other dynamic programming algorithm to expose the parallelism. For example, it is well-known that computations, inside a Smith-Waterman algorithm, lying on same diagonal can be computed in parallel. However, we can rewrite the computations, and can express the dependencies in terms of alternate diagonals. This kind of rewriting will enable us to compute every alternate diagonal, on the same time re-using the partial computations of the middle diagonal. Similarly, look-ahead computations may help to break the intra-diagonal dependencies in Nussinov algorithm.

Another interesting perspective is to automate the identification of parallel prefix networks, for parallelization extraction. The reduction computation and various types of prefix networks provide the designer freedom to compromise between speed and area of the resulting design. It would be interesting to automatically select the most feasible architecture for the given application. Such analysis can be based on the time slack between the production of input to such architectures and the first consumption of the output. Thus, a very sophisticated architecture can be implemented by utilizing the available cycles to execute the computation.

Bibliography

[AAHCH03] Mirela Andronescu, Rosalía Aguirre-Hernández, Anne Condon, and Holger H. Hoos, *RNAsoft: a Suite of RNA Secondary Structure Prediction and Design Software tools*, Nucleic Acids Research **31** (2003), no. 13, 3416–3422.

[ABD07] Christophe Alias, Fabrice Baray, and Alain Darte, *Bee+Cl@k: an Implementation of Lattice-based Array Contraction in the Source-to-Source Translator Rose*, Languages, Compilers, and Tools for Embedded Systems, 2007, pp. 73–82.

[ACL+09] Jeffrey Allred, Jack Coyne, William Lynch, Vincent Natoli, Joseph Grecco, and Joel Morrissette, *Smith-Waterman implementation on a FSB-FPGA module using the Intel Accelerator Abstraction Layer*, IPDPS, 2009, pp. 1–4.

[Aga08] S.K. Agarwal, *Bioinformatics*, APH Publishing Corporation, 2008.

[AGM+90] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman, *Basic Local Alignment Search Tool*, Journal of Molecular Biology **215** (1990), no. 3, 403 – 410.

[AT01] G. Acosta and M. Tosini, *A Firmware Digital Neural Network for Climate Prediction Applications*, Intelligent Control, 2001. (ISIC '01). Proceedings of the 2001 IEEE International Symposium on, 2001, pp. 127 –131.

[BBdM08] Rodolfo Bezerra Batista, Azzedine Boukerche, and Alba Cristina Magalhaes Alves de Melo, *A parallel strategy for biological sequence alignment in restricted memory space*, J. Parallel Distrib. Comput. **68** (2008), 548–561.

[BCHM94] P Baldi, Y Chauvin, T Hunkapiller, and M A McClure, *Hidden Markov models of biological primary sequence information*, Proceedings of the National Academy of Sciences **91** (1994), no. 3, 1059–1063.

[BK82] R.P. Brent and H.T. Kung, *A Regular Layout for Parallel Adders*, IEEE Transactions on Computers **C-31** (1982), no. 3, 260 –264.

[BKMH96] Philipp Bucher, Kevin Karplus, Nicolas Moeri, and Kay Hofmann, *A flexible motif search technique based on generalized profiles*, Computers & Chemistry **20** (1996), no. 1, 3 – 23.

[Ble90] Guy E. Blelloch, *Prefix Sums and Their Applications*, Tech. Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[BMP+03] Michael Brudno, Sanket Malde, Alexander Poliakov, Chuong B. Do, Olivier Couronne, Inna Dubchak, and Serafim Batzoglou, *Glocal alignment: finding rearrangements during alignment*, Eleventh International Conference on Intelligent Systems for Molecular Biology, Brisbane, Austrailia (2003).

[BMZ11] R. Beidas, Wai Sum Mong, and Jianwen Zhu, *Register pressure Aware Scheduling for High Level Synthesis*, Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific, jan. 2011, pp. 461 –466.

[BPN01] P. Baptiste, C.L. Pape, and W. Nuijten, *Constraint-based Scheduling: Applying Constraint Programming to Scheduling Problems*, International series in operations research & management science, Kluwer Academic, 2001.

[BSWG00] Mihai Budiu, Majd Sakr, Kip Walker, and Seth Copen Goldstein, *Bit-Value Inference: Detecting and Exploiting Narrow Bitwidth Computations*, Proceedings from the 6th International Euro-Par Conference on Parallel Processing (London, UK), Euro-Par '00, Springer-Verlag, 2000, pp. 969–979.

[BVK08] K. Benkrid, P. Velentzas, and S. Kasap, *A High Performance Reconfigurable Core for Motif Searching Using Profile HMM*, NASA/ESA Conference on Adaptive Hardware and Systems, June 2008, pp. 285 –292.

[Cas] Norman Casagrande, *Basic-Algorithms-of-Bioinformatics Applet*, http://baba.sourceforge.net.

[CCSV04] Mario Cannataro, Carmela Comito, Filippo Lo Schiavo, and Pierangelo Veltri, *Proteus, a Grid based Problem Solving Environment for Bioinformatics: Architecture and Experiments*, IEEE Computational Intelligence Bulletin **3** (2004).

[CD08] Joo M.P. Cardoso and Pedro C. Diniz, *Compilation Techniques for Reconfigurable Architectures*, 1 ed., Springer Publishing Company, Incorporated, 2008.

[CFH+05] J. Cong, Yiping Fan, Guoling Han, Yizhou Lin, Junjuan Xu, Zhiru Zhang, and Xu Cheng, *Bitwidth-aware Scheduling and Binding in High-Level Synthesis*, Design Automation Conference, 2005. Proceedings of the

ASP-DAC 2005. Asia and South Pacific, vol. 2, jan. 2005, pp. 856 – 861 Vol. 2.

[CGMT09] P. Coussy, D.D. Gajski, M. Meredith, and A. Takach, *An Introduction to High-Level Synthesis*, Design Test of Computers, IEEE **26** (2009), no. 4, 8 –17.

[Dar99] Alain Darte, *On the Complexity of Loop Fusion*, Parallel Computing **26** (1999), 149–157.

[DH02] Alain Darte and Guillaume Huard, *New Results on Array Contraction*, Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors (Washington, DC, USA), ASAP '02, IEEE Computer Society, 2002, pp. 359–.

[DQ07] Steven Derrien and Patrice Quinton, *Parallelizing HMMER for hardware acceleration on FPGAs*, ASAP 2007, 18th IEEE International Conference on Application-specific Systems, Architectures and Processors, July 2007.

[DQ10] Steven Derrien and Patrice Quinton, *Hardware Acceleration of HMMER on FPGAs*, Journal of Signal Processing Systems (2010), 53–67.

[DR00] Steven Derrien and Tanguy Risset, *Interfacing compiled FPGA programs: the MMAlpha approach*, International Workshop on Engineering of Reconfigurable Hardware/Software Objects, 2000.

[DRQR08] Steven Derrien, Sanjay Rajopadhye, Patrice Quinton, and Tanguy Risset, *High-Level Synthesis of Loops Using the Polyhedral Model*, High-Level Synthesis (Philippe Coussy and Adam Morawiec, eds.), Springer Netherlands, 2008, pp. 215–230.

[DSLS10] U. Dhawan, S. Sinha, Siew-Kei Lam, and T. Srikanthan, *Extended Compatibility Path based Hardware Binding Algorithm for Area-Time Efficient Designs*, Quality Electronic Design (ASQED), 2010 2nd Asia Symposium on, aug. 2010, pp. 151 –156.

[DTOG+10] Paolo Di Tommaso, Miquel Orobitg, Fernando Guirado, Fernado Cores, Toni Espinosa, and Cedric Notredame, *Cloud-Coffee: Implementation of a Parallel Consistency-based Multiple Alignment Algorithm in the T-Coffee package and its Benchmarking on the Amazon Elastic-Cloud*, Bioinformatics **26** (2010), no. 15, 1903–1904.

[Edd] Sean Eddy, *HMMER3: a new generation of sequence homology search software*, `http://hmmer.janelia.org/`.

[Edd98] Sean R. Eddy, *Profile Hidden Markov Models*, Bioinformatics **14** (1998), no. 9, 755–763.

[Edd09] ———, *A new generation of homology search tools based on probabilistic inference*, International Conference on Genome Informatics **23** (2009), 205–211.

[Edd10] ———, *HMMER User's Guide : Biological sequence analysis using profile hidden Markov models*, Janelia Farm Research Campus, 2010.

[Edd11a] Sean R. Eddy, *Accelerated Profile HMM Searches*, PLoS Computational Biology **7** (2011), no. 10, e1002195.

[Edd11b] Sean R. Eddy, *Accelerated profile HMM searches (preprint)*, 2011.

[EGMJdM10] Juan Fernando Eusse Giraldo, Nahri Moreano, Ricardo Pezzuol Jacobi, and Alba Cristina Magalhaes Alves de Melo, *A HMMER Hardware Accelerator using Divergences*, Proceedings of the Conference on Design, Automation and Test in Europe, 2010, pp. 405–410.

[Fel93] J. Felsenstein, *PHYLIP: phylogenetic inference package, version 3.5c.*, Seattle, 1993.

[Fin10] Michael Fingeroff, *High-Level Synthesis Blue Book*, Xlibris Corporation, 2010.

[FM89] G. Fettweis and H. Meyr, *Parallel Viterbi algorithm implementation: breaking the ACS-bottleneck*, Communications, IEEE Transactions on **37** (1989), no. 8, 785 –790.

[FM91] ———, *High-speed parallel Viterbi decoding: algorithm and VLSI-architecture*, Communications Magazine, IEEE **29** (1991), no. 5, 46 –55.

[FTM90] G. Fettweis, L. Thiele, and G. Meyr, *Algorithm Transformations for Unlimited Parallelism*, Circuits and Systems, 1990., IEEE International Symposium on, may 1990, pp. 1756 –1759 vol.3.

[Gaj92] Daniel D. Gajski, *High-level Synthesis: Introduction to Chip and System Design*, Kluwer Academic, 1992.

[Gal] NIGMS Image Gallery, *Central Dogma of Life*, `http://images.nigms.nih.gov/`.

[GCBT10] Narayan Ganesan, Roger D. Chamberlain, Jeremy Buhler, and Michela Taufer, *Accelerating HMMER on GPUs by implementing hybrid data and task parallelism*, Proceedings of the First ACM International Conference on Bioinformatics and Computational Biology (New York, NY, USA), BCB '10, ACM, 2010, pp. 418–421.

[GDGN03] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, *SPARK: A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations*, VLSI Design, 2003. Proceedings. 16th International Conference on, jan. 2003, pp. 461 – 466.

[GGDN04] Sumit Gupta, Rajesh Gupta, Nikil D. Dutt, and Alexandru Nicolau, *SPARK:: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*, Springer, 2004.

[GJL97] Pascale Guerdoux-Jamet and Dominique Lavenier, *Samba: Hardware accelerator for biological sequence comparison*, Computer Application in the Biosciences **13** (1997), no. 6, 609–615.

[GLB+08] Andreas R. Gruber, Ronny Lorenz, Stephan H. Bernhart, Richard Neuböck, and Ivo L. Hofacker, *The Vienna RNA Websuite*, Nucleic Acids Research **36** (2008), no. suppl 2, W70–W74.

[GOST92] Guang R. Gao, R. Olsen, Vivek Sarkar, and Radhika Thekkath, *Collective Loop Fusion for Array Contraction*, Languages and Compilers for Parallel Computing, 1992, pp. 281–295.

[GP92] M. Girkar and C.D. Polychronopoulos, *Automatic Extraction of Functional Parallelism from Ordinary Programs*, Parallel and Distributed Systems, IEEE Transactions on **3** (1992), no. 2, 166 –178.

[GQRM00] A.-C. Guillou, P. Quinton, T. Risset, and D. Massicotte, *Automatic design of VLSI pipelined LMS architectures*, Parallel Computing in Electrical Engineering, 2000. PARELEC 2000. Proceedings. International Conference on, 2000, pp. 144 –149.

[GR06] Gautam and S. Rajopadhye, *Simplifying reductions*, POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), ACM, 2006, pp. 30–41.

[HAA11] Laiq Hasan and Zaid Al-Ars, *An Overview of Hardware-Based Acceleration of Biological Sequence Alignment*, Computational Biology and Applied Bioinformatics, InTech, 2011.

[Har03] D. Harris, *A taxonomy of Parallel Prefix Networks*, Thirty-Seventh Asilomar Conference on Signals, Systems and Computers, vol. 2, nov. 2003, pp. 2213 – 2217 Vol.2.

[HC87] T. Han and D. A. Carlson, *Fast area-efficient VLSI adders*, Proceedings of the 8th Symposium on Computer Arithmetic, IEEE, 1987, pp. 49–55.

[HCLH90] Chu-Yi Huang, Yen-Shen Chen, Yan-Long Lin, and Yu-Chin Hsu, *Data Path Allocation based on Bipartite Weighted Matching*, Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE, jun 1990, pp. 499 –504.

[HHH05] D. R. Horn, M. Houston, and P. Hanrahan, *ClawHMMER: A Streaming HMMer-Search Implementation*, SC'05 : Proceedings of the 2005 ACM/IEEE conference on Supercomputing, 2005.

[HKMS93] D. Haussler, A. Krogh, I.S. Mian, and K. Sjolander, *Protein Modeling using Hidden Markov Models: Analysis of Globins*, Proceeding of the Twenty-Sixth Hawaii International Conference on System Sciences, vol. i, jan 1993, pp. 792 – 802 vol.1.

[HLH91] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu, *A Formal Approach to the Scheduling Problem in High Level Synthesis* , Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **10** (1991), no. 4, 464 –475.

[HLSa] *Impulse Accelerated Technologies:, Impulse C*, http://www.impulsec.com.

[HLSb] *Mentor Graphics, Catapult C*, http://www.mentor.com/catapult.

[HLS09a] *Gary Smith EDA, Market Trends*, 2009.

[HLS09b] *NIOS II C2H Compiler User Guide*, November 2009, http://www.altera.com.

[HMS+07] Martin C. Herbordt, Josh Model, Bharat Sukhwani, Yongfeng Gu, and Tom VanCourt, *Single pass streaming BLAST on FPGAs*, Parallel Computing **33** (2007), 741 – 756, ¡ce:title¿High-Performance Computing Using Accelerators¡/ce:title¿.

[Hof03] Ivo L. Hofacker, *Vienna RNA secondary structure server*, Nucleic Acids Research **31** (2003), no. 13, 3429–3431.

[HPH98] Steven Henikoff, Shmuel Pietrokovski, and Jorja G. Henikoff, *Superior Performance in Protein Homology Detection with the Blocks Database servers*, Nucleic Acids Research **26** (1998), no. 1, 309–312.

[Hug96] Richard Hughey, *Parallel Hardware for Sequence Comparison and Alignment*, Computer applications in the biosciences : CABIOS **12** (1996), no. 6, 473–479.

[INT] *UniProtKB/Swiss-Prot Protein Knowledgebase Release 2012-01 Statistics*, http://web.expasy.org/docs/relnotes/relstat.html.

[Ive62] Kenneth A. Iverson, *A Programming Language*, Jonh Wiley & Sons, 1962.

[JBC08] A. Jacob, J. Buhler, and R.D. Chamberlain, *Accelerating Nussinov RNA secondary structure prediction with systolic arrays on FPGAs*, Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on, july 2008, pp. 191 –196.

[JBC10] A.C. Jacob, J.D. Buhler, and R.D. Chamberlain, *Rapid RNA Folding: Analysis and Acceleration of the Zuker Recurrence*, Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on, may 2010, pp. 87 –94.

[JH98] Tommi Jaakkola and David Haussler, *Exploiting Generative Models in Discriminative Classifiers*, Advances in Neural Information Processing Systems 11, MIT Press, 1998, pp. 487–493.

[JLBC07] Arpith C. Jacob, Joseph M. Lancaster, Jeremy D. Buhler, and Roger D. Chamberlain, *Preliminary results in accelerating profile HMM search on FPGAs*, Workshop on High Performance Computational Biology (HiCOMB), 2007.

[JP04] Neil C. Jones and Pavel A. Pevzner, *An Introduction to Bioinformatics Algorithms (Computational Molecular Biology)*, The MIT Press, August 2004.

[KAH97] P. Kollig and B.M. Al-Hashimi, *Simultaneous Scheduling, Allocation and Binding in High Level Synthesis*, Electronics Letters **33** (1997), no. 18, 1516 –1518.

[KBM+94] A. Krogh, M. Brown, I. S. Mian, K. Sjölander, and D. Haussler, *Hidden Markov Models in Computational Biology: Applications to Protein Modeling*, Journal Molecular Biology **235** (1994), 1501–1531.

[KH03] Bjarne Knudsen and Jotun Hein, *Pfold: RNA secondary structure prediction using stochastic context-free grammars*, Nucleic Acids Research **31** (2003), no. 13, 3423–3428.

[KL07] Taemin Kim and Xun Liu, *Compatibility Path Based Binding Algorithm for Interconnect Reduction in High Level Synthesis*, Computer-Aided Design, 2007. ICCAD 2007. IEEE/ACM International Conference on, nov. 2007, pp. 435 –441.

[KM94] Ken Kennedy and Kathryn S. McKinley, *Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution*, IN LANGUAGES AND COMPILERS FOR PARALLEL COMPUTING, Springer-Verlag, 1994, pp. 301–320.

[Kno99] S. Knowles, *A Family of Adders*, ARITH '99: Proceedings of the 14th IEEE Symposium on Computer Arithmetic (Washington, DC, USA), IEEE Computer Society, 1999, p. 30.

[KR07] I. Kuon and J. Rose, *Measuring the Gap Between FPGAs and ASICs*, Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **26** (2007), no. 2, 203 –215.

[KS73] Peter M. Kogge and Harold S. Stone, *A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations*, IEEE Transcation on Computers **22** (1973), no. 8, 786–793.

[Kuc03] Krzysztof Kuchcinski, *Constraints-driven Scheduling and Resource Assignment*, ACM Trans. Des. Autom. Electron. Syst. **8** (2003), 355–383.

[Lam88] M. Lam, *Software Pipelining: An Effective Scheduling Technique for VLIW Machines*, Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation (New York, NY, USA), PLDI '88, ACM, 1988, pp. 318–328.

[LBP+08] Heshan Lin, Pavan Balaji, Ruth Poole, Carlos Sosa, Xiaosong Ma, and Wu-chun Feng, *Massively Parallel Genomic Sequence Search on the Blue Gene/P Architecture*, Proceedings of the 2008 ACM/IEEE conference on Supercomputing (Piscataway, NJ, USA), SC '08, IEEE Press, 2008, pp. 33:1–33:11.

[LF80] Richard E. Ladner and Michael J. Fischer, *Parallel Prefix Computation*, Journal of ACM **27** (1980), no. 4, 831–838.

[LHL89] Jiahn-Hung Lee, Yu-Chin Hsu, and Youn-Long Lin, *A New Integer Linear Programming Formulation for the Scheduling Problem in Data Path Synthesis*, Computer-Aided Design, 1989. ICCAD-89. Digest of Technical Papers., 1989 IEEE International Conference on, nov 1989, pp. 20 –23.

[LKC+10] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey, *Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU*, Proceedings of the 37th annual international symposium on Computer architecture, ISCA '10, 2010, pp. 451–460.

[LL85] R.J. Lipton and D. Lopresti, *A systolic array for rapid string comparison*, Chapel Hill Conference on VLSI, 1985, pp. 363–376.

[LMD94] Birger Landwehr, Peter Marwedel, and Rainer Dömer, *OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer*

Programming, Proceedings of the conference on European design automation (Los Alamitos, CA, USA), EURO-DAC '94, IEEE Computer Society Press, 1994, pp. 90–95.

[LPAF08] Jizhu Lu, M. Perrone, K. Albayraktaroglu, and M. Franklin, *HMMer-Cell: High Performance Protein Profile Searching on the Cell/B.E. Processor*, IEEE International Symposium on Performance Analysis of Systems and Software, april 2008, pp. 223 –232.

[LS91] Charles Leiserson and James Saxe, *Retiming synchronous circuitry*, Algorithmica **6** (1991), 5–35, 10.1007/BF01759032.

[LSK+05] O. Lehtoranta, E. Salminen, A. Kulmala, M. Hannikainen, and T.D. Hamalainen, *A Parallel MPEG-4 Encoder for FPGA Based Multiprocessor SoC*, Field Programmable Logic and Applications, 2005. International Conference on, aug. 2005, pp. 380 – 385.

[LST] Isaac TS Li, Warren Shum, and Kevin Truong, *160-fold Acceleration of the Smith-Waterman Algorithm using a Field Programmable Gate Array (FPGA)*.

[LVMQ91] Hervé Le Verge, Christophe Mauras, and Patrice Quinton, *The ALPHA language and its use for the design of systolic arrays*, J. VLSI Signal Process. Syst. **3** (1991), 173–182.

[MA97] Naraig Manjikian and Tarek S. Abdelrahman, *Fusion of Loops for Parallelism and Locality*, IEEE Trans. Parallel Distrib. Syst. **8** (1997), 193–209.

[MBC+06] R. P. Maddimsetty, J. Buhler, R. D. Chamberlain, M. A. Franklin, and Brandon Harris, *Accelerator Design for Protein Sequence HMM Search*, Proceedings of the ACM International Conference on Supercomputing (Cairns, Australia), ACM, 2006.

[MC95] E. Musoll and J. Cortadella, *High-Level Synthesis Techniques for Reducing the Activity of Functional Units*, Proceedings of the 1995 international symposium on Low power design (New York, NY, USA), ISLPED '95, ACM, 1995, pp. 99–104.

[MDQ11] A. Morvan, S. Derrien, and P. Quinton, *Efficient Nested Loop Pipelining in High Level Synthesis using Polyhedral Bubble Insertion*, Field-Programmable Technology (FPT), 2011 International Conference on, dec. 2011, pp. 1 –10.

[MFDW98] B Morgenstern, K Frech, A Dress, and T Werner, *DIALIGN: finding local similarities by multiple sequence alignment.*, Bioinformatics **14** (1998), no. 3, 290–294.

[MKFD11] Antoine Morvan, Amit Kumar, Antoine Floch, and Steven Derrien, *GeCoS : a source to source optimizing compiler for the automatic synthesis of parallel hardware accelerators*, Poster in Designing for Embedded Parallel Computing Platforms at DATE, March 2011.

[MoH05] John Morrison, Padraig o'Dowd, and Philip Healy, *An Investigation of the Applicability of Distributed FPGAs to High-Performance Computing*, High-performance computing: paradigm and infrastructure, Wiley, 2005.

[Moo66] R.E. Moore, *Interval Analysis*, Prentice-Hall, Englewood Cliffs N. J., 1966.

[Mou04] David W. Mount, *Bioinformatics: Sequence and Genome Analysis*, Cold Spring Harbor Laboratory Press, 2004.

[MPPZ87] Daniel J. Magenheimer, Liz Peters, Karl Pettis, and Dan Zuras, *Integer Multiplication and Division on the HP Precision Architecture*, Proceedings of the second international conference on Architectual support for programming languages and operating systems (Los Alamitos, CA, USA), ASPLOS-II, IEEE Computer Society Press, 1987, pp. 90–99.

[MS09] G. Martin and G. Smith, *High-Level Synthesis: Past, Present, and Future*, Design Test of Computers, IEEE **26** (2009), no. 4, 18 –25.

[MTF08] A. Matsunaga, M. Tsugawa, and J. Fortes, *CloudBLAST: Combining MapReduce and Virtualization on Distributed Resources for Bioinformatics Applications*, eScience, 2008. eScience '08. IEEE Fourth International Conference on, dec. 2008, pp. 222 –229.

[MV08] Svetlin Manavski and Giorgio Valle, *CUDA Compatible GPU cards as Efficient Hardware Accelerators for Smith-Waterman Sequence Alignment*, BMC Bioinformatics **9** (2008), no. Suppl 2, S10.

[NCB11] NCBI, *GenBank Realse Notes*, August 2011.

[NLLL97] Andrew F. Neuwald, Jun S. Liu, David J. Lipman, and Charles E. Lawrence, *Extracting Protein Alignment Models from the Sequence Database*, Nucleic Acids Research **25** (1997), no. 9, 1665–1677.

[NPGK78] Ruth Nussinov, George Pieczenik, Jerrold R. Griggs, and Daniel J. Kleitman, *Algorithms for Loop Matchings*, SIAM Journal on Applied Mathematics **35** (1978), no. 1, 68–82.

[NW70] Saul B. Needleman and Christian D. Wunsch, *A General Method Applicable to The Search for Similarities in The Amino Acid Sequence of Two Proteins*, Journal of Molecular Biology **48** (1970), no. 3, 443 – 453.

[oEGP08] U.S. Department of Energy Genome Programs, *The Science Behind the Human Genome Project* , http://genomics.energy.gov, 2008.

[OSJM06] T. Oliver, B. Schmidt, Y. Jakop, and D. L. Maskell, *Accelerating the Viterbi Algorithm for Profile Hidden Markov Models Using Reconfigurable Hardware.*, International Conference on Computational Science, 2006.

[OYS07] T. Oliver, L. Y. Yeow, and B. Schmidt, *High Performance Database Searching with HMMer on FPGAs*, HiCOMB 2007, Sixth IEEE International Workshop on High Performance Computational Biology, march 2007.

[PK87] P. G. Paulin and J. P. Knight, *Force-Directed Scheduling in Automatic Data Path Synthesis*, Proceedings of the 24th ACM/IEEE Design Automation Conference (New York, NY, USA), DAC '87, ACM, 1987, pp. 195–202.

[PK89] P.G. Paulin and J.P. Knight, *Force-Directed Scheduling for The Behavioral Synthesis of ASICs*, Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **8** (1989), no. 6, 661 –679.

[Ple10] Alexandre Plesco, *Program Transformation and Memory Architecture Optimization for High-Level Synthesis of Hardware Accelerators*, Ph.D. thesis, École Normale Supérieure de Lyon, 2010.

[PT05] David Pellerin and Scott Thibault, *Practical FPGA Programming in C*, Prentice Hall, 2005.

[QEB+09] Xiaohong Qiu, Jaliya Ekanayake, Scott Beason, Thilina Gunarathne, Geoffrey Fox, Roger Barga, and Dennis Gannon, *Cloud Technologies for Bioinformatics Applications*, Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers (New York, NY, USA), MTAGS '09, ACM, 2009, pp. 6:1–6:10.

[QR91] Patrice Quinton and Yves Robert, *Systolic Algorithms and Architectures*, Prentice Hall, 1991.

[RFJ95] Minjoong Rim, Yaw Fann, and Rajiv Jain, *Global Scheduling with Code-Motions for High-Level Synthesis Applications*, Very Large Scale Integration (VLSI) Systems, IEEE Transactions on **3** (1995), no. 3, 379 –392.

[rg] CAIRN-INRIA research group, *The GeCoS: The Generic Compiler Suite*, `http://gecos.gforge.inria.fr/`.

[Ric] Allen B. Richon, *A Short History of Bioinformatics*, Network Science Corporation, http://www.netsci.org/Science/Bioinform/feature06.html.

[RJ86] L. Rabiner and B. Juang, *An Introduction to Hidden Markov Models*, IEEE ASSP Magazine **3** (1986), no. 1, 4 – 16.

[RJDL92] M. Rim, R. Jain, and R. De Leone, *Optimal Allocation and Binding in High-Level Synthesis*, Proceedings of the 29th ACM/IEEE Design Automation Conference (Los Alamitos, CA, USA), DAC '92, IEEE Computer Society Press, 1992, pp. 120–123.

[RP08] Tanguy Risset and Alexandru Plesco, *Coupling Loop Transformations and High-Level Synthesis*, Symposium en Architecture de machines (Sympa 2008) (Fribourg, Suisse), 2008.

[SBA00] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe, *Bitwidth Analysis with Application to Silicon Compilation*, In Proceedings of the SIGPLAN conference on Programming Language Design and Implementation, 2000, pp. 108–120.

[Sch08] Sophie Schneiderbauer, *RNA Secondary Structure Prediction*, Bachelor's Thesis, Cognitive Science, University Of Osnabrück, 2008.

[SD02] A.M. Sllame and V. Drabek, *An efficient List-Based Scheduling Algorithm for High-level Synthesis*, Digital System Design, 2002. Proceedings. Euromicro Symposium on, 2002, pp. 316 – 323.

[SDLS11] S. Sinha, U. Dhawan, Siew Kei Lam, and T. Srikanthan, *A Novel Binding Algorithm to Reduce Critical Path Delay During High Level Synthesis*, VLSI (ISVLSI), 2011 IEEE Computer Society Annual Symposium on, july 2011, pp. 278 –283.

[SEQa] *BIOWIC: Bioinformatics Workflow for Intensive Computation*, http://biowic.inria.fr/.

[SEQb] *Longman Science Biology 9*, Pearson Education India.

[SEQc] *Review of the Universe, Unicellular Organisms*, http://universe-review.ca.

[SG91] Vivek Sarkar and Guang R. Gao, *Optimization of Array Accesses by Collective Loop Transformations.*, ICS'91, 1991, pp. 194–205.

[SKD06] E. Sotiriades, C. Kozanitis, and A. Dollas, *FPGA based Architecture for DNA Sequence Comparison and Database Search*, Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, april 2006, p. 8 pp.

[Skl60] J. Sklansky, *Conditional-Sum Addition Logic*, IRE Transactions on Electronic Computers **EC-9** (1960), no. 2, 226 –231.

[SLG+09] Yanteng Sun, Peng Li, Guochang Gu, Yuan Wen, Yuan Liu, , and Dong Liu, *HMMER Acceleration Using Systolic Array Based Reconfigurable Architecture*, IEEE International Workshop on High Performance Computational Biology, 2009.

[SRG03] Robert D. Stevens, Alan J. Robinson, and Carole A. Goble, *myGrid: Personalised Bioinformatics on the Information Grid*, Bioinformatics **19** (2003), no. suppl 1, i302–i304.

[SRV+07] Lieven Sterck, Stephane Rombauts, Klaas Vandepoele, Pierre Rouze´, and Yves Van de Peer, *How many genes are there in plants (... and why are they there)?*, Current Opinion in Plant Biology (2007), 199–203.

[SW81] T.F. Smith and M.S. Waterman, *Identification of Common Molecular Subsequences*, Journal of Molecular Biology **147** (1981), no. 1, 195 – 197.

[SXWL04] Yonghong Song, Rong Xu, Cheng Wang, and Zhiyuan Li, *Improving Data Locality by Array Contraction*, IEEE Trans. Comput. **53** (2004), 1073–1084.

[Tha09] Sabu M. Thampi, *Bioinformatics*, 2009, http://arxiv.org/ftp/arxiv/papers/0911/0911.4230.pdf.

[THG94] Julie D. Thompson, Desmond G. Higgins, and Toby J. Gibson, *CLUSTAL W: Improving the Sensitivity of Progressive Multiple Sequence Alignment Through Sequence Weighting, Position-Specific Gap Penalties and Weight Matrix Choice*, Nucleic Acids Research **22** (1994), no. 22, 4673–4680.

[TM99] Tatiana A Tatusova and Thomas L Madden, *Blast 2 sequences, a new tool for comparing protein and nucleotide sequences*, FEMS Microbiology Letters **174** (1999), no. 2, 247–250.

[TM09] Toyokazu Takagi and Tsutomu Maruyama, *Accelerating HMMER Search Using FPGA*, International Conference on Field Programmable Logic and Applications, September 2009.

[TS86] Chia-Jeng Tseng and D.P. Siewiorek, *Automated Synthesis of Data Paths in Digital Systems*, Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **5** (1986), no. 3, 379 – 395.

[VAM+01] J. Craig Venter, Mark D. Adams, Eugene W. Myers, Peter W. Li, Richard J. Mural, Granger G. Sutton, Hamilton O. Smith, Mark Yandell, Cheryl A. Evans, Robert A. Holt, Jeannine D. Gocayne, Peter Amanatides, Richard M. Ballew, Daniel H. Huson, Jennifer Russo Wortman, Qing Zhang, Chinnappa D. Kodira, Xiangqun H. Zheng, Lin Chen, Marian Skupski, Gangadharan Subramanian, Paul D. Thomas, Jinghui Zhang, George L. Gabor Miklos, Catherine Nelson, Samuel Broder, Andrew G. Clark, Joe Nadeau, Victor A. McKusick, Norton Zinder, Arnold J. Levine, Richard J. Roberts, Mel Simon, Carolyn Slayman, Michael Hunkapiller, Randall Bolanos, Arthur Delcher, Ian Dew, Daniel Fasulo, Michael Flanigan, Liliana Florea, Aaron Halpern, Sridhar Hannenhalli, Saul Kravitz, Samuel Levy,

Clark Mobarry, Knut Reinert, Karin Remington, Jane Abu-Threideh, Ellen Beasley, Kendra Biddick, Vivien Bonazzi, Rhonda Brandon, Michele Cargill, Ishwar Chandramouliswaran, Rosane Charlab, Kabir Chaturvedi, Zuoming Deng, Valentina Di Francesco, Patrick Dunn, Karen Eilbeck, Carlos Evangelista, Andrei E. Gabrielian, Weiniu Gan, Wangmao Ge, Fangcheng Gong, Zhiping Gu, Ping Guan, Thomas J. Heiman, Maureen E. Higgins, Rui-Ru Ji, Zhaoxi Ke, Karen A. Ketchum, Zhongwu Lai, Yiding Lei, Zhenya Li, Jiayin Li, Yong Liang, Xiaoying Lin, Fu Lu, Gennady V. Merkulov, Natalia Milshina, Helen M. Moore, Ashwinikumar K Naik, Vaibhav A. Narayan, Beena Neelam, Deborah Nusskern, Douglas B. Rusch, Steven Salzberg, Wei Shao, Bixiong Shue, Jingtao Sun, Zhen Yuan Wang, Aihui Wang, Xin Wang, Jian Wang, Ming-Hui Wei, Ron Wides, Chunlin Xiao, Chunhua Yan, Alison Yao, Jane Ye, Ming Zhan, Weiqing Zhang, Hongyu Zhang, Qi Zhao, Liansheng Zheng, Fei Zhong, Wenyan Zhong, Shiaoping C. Zhu, Shaying Zhao, Dennis Gilbert, Suzanna Baumhueter, Gene Spier, Christine Carter, Anibal Cravchik, Trevor Woodage, Feroze Ali, Huijin An, Aderonke Awe, Danita Baldwin, Holly Baden, Mary Barnstead, Ian Barrow, Karen Beeson, Dana Busam, Amy Carver, Angela Center, Ming Lai Cheng, Liz Curry, Steve Danaher, Lionel Davenport, Raymond Desilets, Susanne Dietz, Kristina Dodson, Lisa Doup, Steven Ferriera, Neha Garg, Andres Gluecksmann, Brit Hart, Jason Haynes, Charles Haynes, Cheryl Heiner, Suzanne Hladun, Damon Hostin, Jarrett Houck, Timothy Howland, Chinyere Ibegwam, Jeffery Johnson, Francis Kalush, Lesley Kline, Shashi Koduru, Amy Love, Felecia Mann, David May, Steven McCawley, Tina McIntosh, Ivy McMullen, Mee Moy, Linda Moy, Brian Murphy, Keith Nelson, Cynthia Pfannkoch, Eric Pratts, Vinita Puri, Hina Qureshi, Matthew Reardon, Robert Rodriguez, Yu-Hui Rogers, Deanna Romblad, Bob Ruhfel, Richard Scott, Cynthia Sitter, Michelle Smallwood, Erin Stewart, Renee Strong, Ellen Suh, Reginald Thomas, Ni Ni Tint, Sukyee Tse, Claire Vech, Gary Wang, Jeremy Wetter, Sherita Williams, Monica Williams, Sandra Windsor, Emily Winn-Deen, Keriellen Wolfe, Jayshree Zaveri, Karena Zaveri, Josep F. Abril, Roderic Guigó, Michael J. Campbell, Kimmen V. Sjolander, Brian Karlak, Anish Kejariwal, Huaiyu Mi, Betty Lazareva, Thomas Hatton, Apurva Narechania, Karen Diemer, Anushya Muruganujan, Nan Guo, Shinji Sato, Vineet Bafna, Sorin Istrail, Ross Lippert, Russell Schwartz, Brian Walenz, Shibu Yooseph, David Allen, Anand Basu, James Baxendale, Louis Blick, Marcelo Caminha, John Carnes-Stine, Parris Caulk, Yen-Hui Chiang, My Coyne, Carl Dahlke, Anne Deslattes Mays, Maria Dombroski, Michael Donnelly, Dale Ely, Shiva Esparham, Carl Fosler, Harold Gire, Stephen Glanowski, Kenneth Glasser, Anna Glodek, Mark Gorokhov, Ken Graham, Barry Gropman, Michael

Harris, Jeremy Heil, Scott Henderson, Jeffrey Hoover, Donald Jennings, Catherine Jordan, James Jordan, John Kasha, Leonid Kagan, Cheryl Kraft, Alexander Levitsky, Mark Lewis, Xiangjun Liu, John Lopez, Daniel Ma, William Majoros, Joe McDaniel, Sean Murphy, Matthew Newman, Trung Nguyen, Ngoc Nguyen, Marc Nodell, Sue Pan, Jim Peck, Marshall Peterson, William Rowe, Robert Sanders, John Scott, Michael Simpson, Thomas Smith, Arlan Sprague, Timothy Stockwell, Russell Turner, Eli Venter, Mei Wang, Meiyuan Wen, David Wu, Mitchell Wu, Ashley Xia, Ali Zandieh, and Xiaohong Zhu, *The Sequence of the Human Genome*, Science **291** (2001), no. 5507, 1304–1351.

[VS11] Panagiotis D. Vouzis and Nikolaos V. Sahinidis, *GPU-BLAST: Using Graphics Processors to Accelerate Protein Sequence Alignment*, Bioinformatics **27** (2011), no. 2, 182–188.

[Wak04] Kazutoshi Wakabayashi, *C-based Behavioral Synthesis and Verification Analysis on Industrial Design Examples*, Proceedings of the 2004 Asia and South Pacific Design Automation Conference (Piscataway, NJ, USA), ASP-DAC '04, IEEE Press, 2004, pp. 344–348.

[WBC05] Ben Wun, Jeremy Buhler, and Patrick Crowley, *Exploiting Coarse-Grained Parallelism to Accelerate Protein Motif Finding with a Network Processor*, PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, 2005.

[WBKC09a] John Paul Walters, Vidyananth Balu, Suryaprakash Kompalli, and Vipin Chaudhary, *Evaluating the use of GPUs in Liver Image Segmentation and HMMER Database Searches*, IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing (Washington, DC, USA), IEEE Computer Society, 2009, pp. 1–12.

[WBKC09b] J.P. Walters, V. Balu, S. Kompalli, and V. Chaudhary, *Evaluating the use of GPUs in Liver Image Segmentation and HMMER Database Searches*, IEEE International Symposium on Parallel and Distributed Processing (IPDPS09), 2009.

[WGK08] Gang Wang, Wenrui Gong, and Ryan Kastner, *Operation Scheduling: Algorithms and Applications*, High-Level Synthesis (Philippe Coussy and Adam Morawiec, eds.), Springer Netherlands, 2008, pp. 231–255.

[Wil03] Steve Wiley, *DNA to RNA to Protein*, Tutorials - Biology for the Novice, 2003.

[Wol90] Michael Joseph Wolfe, *Optimizing Supercompilers for Supercomputers*, MIT Press, Cambridge, MA, USA, 1990.

[Wol96] M.J. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley, 1996.

[WP84] Michael Waterman and Marcela Perlwitz, *Line Geometries for Sequence Comparisons*, Bulletin of Mathematical Biology **46** (1984), 567–577, 10.1007/BF02459504.

[WV08] N.A. Woods and T. VanCourt, *FPGA Acceleration of Quasi-Monte Carlo in Finance*, Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on, sept. 2008, pp. 335 –340.

[YHK09] Chao-Tung Yang, Tsu-Fen Han, and Heng-Chuan Kan, *G-BLAST: a Grid-based Solution for mpiBLAST on Computational Grids*, Concurrency and Computation: Practice and Experience **21** (2009), no. 2, 225–255.

[ZLH+05] G.L. Zhang, P.H.W. Leong, C.H. Ho, K.H. Tsoi, C.C.C. Cheung, D.-U. Lee, R.C.C. Cheung, and W. Luk, *Reconfigurable Acceleration for Monte Carlo based Financial Simulation*, Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on, dec. 2005, pp. 215 –222.

[ZS81] Michael Zuker and P. Stiegler, *Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information*, Nucleic Acids Research **9** (1981), no. 1, 133–148.

[ZS84] Michael Zuker and David Sankoff, *RNA Secondary Structures and Their Prediction*, Bulletin of Mathematical Biology **46** (1984), 591–621, 10.1007/BF02459506.

[Zuk03] Michael Zuker, *Mfold web server for Nucleic Acid Folding and Hybridization Prediction*, Nucleic Acids Research **31** (2003), no. 13, 3406–3415.

[ZZZ+10] Jiyu Zhang, Zhiru Zhang, Sheng Zhou, Mingxing Tan, Xianhua Liu, Xu Cheng, and Jason Cong, *Bit-level Optimization for High-level Synthesis and FPGA-based Acceleration*, Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays (New York, NY, USA), FPGA '10, ACM, 2010, pp. 59–68.