Comparability Graph Coloring for Optimizing Utilization of Stream Register Files in Stream Processors

Xuejun Yang* Li Wang * † ¶ Jingling Xue† Yu Deng* Ying Zhang*

National Laboratory for Parallel and Distributed Processing, School of Computer, NUDT, China*

Programming Languages and Compilers Group, School of Computer Science and Engineering, UNSW, Australia†

{lwang, jingling}@cse.unsw.edu.au {xjyang, yudeng, zhangying}@nudt.edu.cn

Abstract

A stream processor executes an application that has been decomposed into a sequence of kernels that operate on streams of data elements. During the execution of a kernel, all streams accessed must be communicated through the SRF (Stream Register File), a non-bypassing software-managed on-chip memory. Therefore, optimizing utilization of the SRF is crucial for good performance. The key insight is that the interference graphs formed by the streams in stream applications tend to be comparability graphs or decomposable into a set of multiple comparability graphs. We present a compiler algorithm that can find optimal or near-optimal colorings in stream IGs, thereby improving SRF utilization than the First-Fit bin-packing algorithm, the best in the literature.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—compilers and optimization; B.3.2 [*Memory Structures*]: Design Styles—Primary memory

General Terms Algorithms, Languages, Performance

Keywords Stream processor, stream programming, comparability graph coloring, software-managed cache

1. Introduction

Media applications, such as image processing, signal processing, video and graphics, are becoming an increasingly dominant portion of computing workloads today. In contrast with other applications, media applications exhibit producer-consumer locality with little global data reuse, have abundant parallelism and require high computation rates (with 10-100 billion operations per second and a few to thousands of operations per input data). These characteristics are poorly matched to conventional general-purpose programmable architectures that depend on data reuse (captured by hardware-managed caches), cannot exploit the available parallelism and cannot support high computation rates. On the other hand, special-purpose media-processing processors tailored to one specific application require significant design effort and are thus difficult to change as applications and/or algorithms evolve.

¶ The work was carried out during Li Wang's visit to Jingling Xue's Research Group at UNSW during February 2008 – February 2009.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPoPP'09, February 14–18, 2009, Raleigh, North Carolina, USA.

Copyright © 2009 ACM 978-1-60558-397-6/09/02 . . . $5.00

The (programmable) stream processors, such as Imagine [18], Raw [20], Cell [23], AMD FireStream and Merrimac [4], represent a promising alternative in achieving high performance in media applications [17, 18, 21]. In addition, stream processing is also well suited for some scientific applications [4, 23, 25].

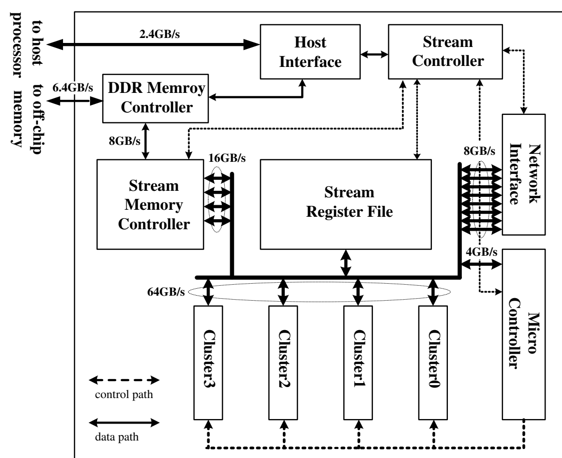


Figure 1: Block diagram of the FT64 stream processor.

We have recently designed and fabricated a 64-bit stream processor, FT64 [25], for media applications as well as certain scientific applications that are also amenable to stream processing. As shown in Figure 1, like Imagine [18], Cell [23] and Merrimac [4], FT64 can be easily mapped to the stream virtual machine architecture described in [12]. Such stream processor executes applications that have been mapped to the stream programming model: a program is decomposed into a sequence of computation-intensive kernels that operate on streams of data elements. Kernels are compiled to VLIW microprograms to be executed on clusters of ALUs, one at a time. Streams are stored in the SRF (Stream Register File), a software-managed on-chip memory. Expressing an application as streams exposes its inherent locality and parallelism. Kernels expose kernel locality by keeping temporary values local (in the non-shown local register files near the ALUs) and instruction-level parallelism (exploited by the multiple ALUs in each cluster). Streams expose producer-consumer locality between kernels — enabling some output streams produced by a kernel to be consumed by the next kernel in sequence — and data-level parallelism — enabling different elements of an input stream to be operated on simultaneously, one on each ALU cluster, in a SIMD fashion.

Research into advanced compiler technology for stream languages and architectures is still at its infancy. Among several chal-

lenges posed by stream processing for compilation [5], an efficient allocation of the scarce on-chip SRF is critical to performance. SRF, the nexus of a stream processor, is introduced to capture the widespread producer-consumer locality in media applications to reduce expensive off-chip memory traffic. Unlike conventional register files, however, SRF is non-bypassing, namely, the input and output streams of a kernel must be all stored in the SRF when a kernel is being executed. If the work set of a kernel is too large to fit into the SRF, strip mining can be applied to segment some large streams into smaller strips so that the kernel can then be called to operate on one strip at a time. Alternatively, some streams can be double-buffered [5] or spilled [22] until the data set of every kernel does not exceed the SRF capacity. Therefore, optimizing utilization of SRF is crucial for good performance.

We are aware of two existing SRF management techniques for stream processors [5, 22]. In [5], SRF utilization is optimized by applying First-Fit bin-packing heuristics. In our recent work [22], we have experimented with adopting a graph coloring approach that we introduced in [14] for scratchpad allocation to SRF allocation. This graph coloring approach requires the SRF to be partitioned into pseudo registers before graph coloring can be applied. Artificial aliases among pseudo registers may cause SRF fragmentation, reducing SRF utilization unnecessarily. On the other hand, First-Fit heuristics can be sub-optimal for many applications. For small applications, either technique suffices. For large applications with tens to hundreds of kernels, both need to be further improved.

In this paper, we present a new compiler algorithm that aims to optimize utilization of SRF for stream applications. The central machinery is the traditional interference graph (IG) representation except an IG here is a weighted (undirected) graph formed by the streams operated on by a sequence of kernels. The key discovery is that the IGs in many media applications are comparability graphs, enabling the compiler to obtain optimal colorings in polynomial time. This has motivated us to develop a new algorithm for optimizing utilization of SRF when allocating the streams in stream IGs to the SRF by comparability graph coloring. If the data set of a kernel still exceeds the SRF capacity after SRF allocation, live-range splitting (or spilling) and strip mining can be applied, as will be discussed in the concluding section of this paper.

In summary, this paper makes the following contributions:

- We propose, for the first time, to optimize utilization of SRF by comparability graph coloring and present an efficient algorithm designed for well-structured media and scientific applications amenable to stream processing.
- We show that our algorithm can find optimal and near-optimal colorings for stream IGs, thereby outperforming First-Fit heuristics.

The rest of this paper is organized as follows. For background information, Section 2 introduces the stream programming model by an example. In Section 3, we make precise the SRF management problem we solve. Section 4 casts it as a comparability graph coloring problem and presents our algorithm for solving the new formulation. Section 5 evaluates our approach. Section 6 discuss related work. Section 7 concludes by discussing future work.

2. Stream Programming Model

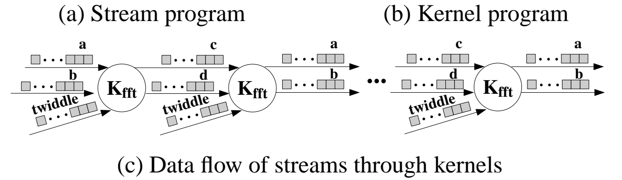
The central idea behind stream processing is to divide an application into *kernels* and *streams* to expose its inherent locality and parallelism. As a result, an application is divided into two programs, a *stream program* running on the host processor and a *kernel program* running on the stream processor. The stream program specifies the flow of streams between kernels and initiates the execution of kernels. The kernel program executes these kernels, one at a time.

```
1   complex xmat[2*N], ymat[2*N];
2   complex twiddlemat[log2(2*N)*N];
3   stream<complex> a(N), b(N);
4   stream<complex> twiddle(N);
5   stream<complex> c(N), d(N);

6   dataInit('xmatix.dat', xmat);
7   dataInit('twiddlematrix.dat', twiddlemat);

8   Load(xmat[0, N-1], a);
9   Load(xmat[N, 2*N-1], b);
10  for (int i = 0; i < log2(2*N); i+=2) {
11      Load(twiddlemat[i*N, (i+1)*N-1], twiddle);
12      Kernel('fft', a, b, twiddle, c, d);
13      Load(twiddlemat[(i+1)*N, (i+2)*N-1], twiddle);
14      Kernel('fft', c, d, twiddle, a, b);
15  }
16  Store(a, ymat[0, N-1]);
17  Store(b, ymat[N, 2*N-1]);

18  bitReverse(ymat);
19  dataSave('ymatrix.dat', ymat);
```

(a) Stream program

```
1   fft(stream<complex> a, stream<complex> b,
2       stream<complex> twiddle,
3       stream<complex> c, stream<complex> d)
4   {
5       complex a_tmp, b_tmp, c_tmp, d_tmp;
6       complex twiddle_tmp;

7       for (int i = 0; i < N/2; i++) {
8           a>>a_tmp;
9           b>>b_tmp;
10          twiddle>>twiddle_tmp;
11          c<<a_tmp+b_tmp;
12          c<<twiddle_tmp*(a_tmp-b_tmp);
13      }
14      for (i = N/2; i < N; i++) {
15          a>>a_tmp;
16          b>>b_tmp;
17          twiddle>>twiddle_tmp;
18          d<<a_tmp+b_tmp;
19          d<<twiddle_tmp*(a_tmp-b_tmp);
20      }
21  }
```

(b) Kernel program



(c) Data flow of streams through kernels

Figure 2: Stream and kernel programs for a radix FFT.

Figure 2 depicts the mapping of a $2N$-point radix-2 FFT to the stream programming model. The kernel *fft* is executed $\log_2(2N)$ times with explicit producer-consumer locality: every output stream from a kernel execution is used as an input for the next kernel execution in sequence.

Let us examine the stream program first. In lines 1 and 2, three arrays of sizes $2N$, $2N$ and $\log_2(2N) * N$ are declared, respectively. In lines 3 – 5, five streams of size N are declared. In lines 6 and 7, the function *dataInit* is called twice to initialize arrays *xmat* and *twiddlemat* residing in the off-chip memory with the two data files stored at the host processor. In line 8, the data in the first half of *xmat* are gathered into stream *a*. This will result in the loading of the data from *xmat* in off-chip memory into the space allocated to stream *a* in the SRF. In line 9, stream *b* is initialized from the second half of *xmat*. In line 10, the loop in a sequential FFT program is unrolled once to expose the producer-consumer locality between the kernel calls to *fft*. In line 11, the "twiddle factors" needed by FFT are gathered into stream *twiddle*. In line 12, the kernel *fft* is called to perform the core computation of FFT on the stream processor. As shown, *a*, *b* and *twiddle* are input streams and *c* and *d* are output streams. In line 13, stream *twiddle* is updated with new twiddle factors. In line 14, *fft* is called again with *c*, *d*, *twiddle* as input and *a* and *b* as output. After the kernel has run to completion, the final output streams are stored from the SRF into array *ymat* in off-chip memory (lines 16 and 17). Since the output is in bit-reversed order, In line 18, the function *bitReverse* reorders the data. In line 19, result is saved into a data file.

In the kernel program, a loop at line 7 first goes over the first half of each input stream. In line 8, the elements of stream *a* are read sequentially, one a time, into a temporary variable *a_tmp*. In lines 9 and 10, the elements of streams *y* and *twiddle* are read off similarly. In lines 11 and 12, the computations on these elements are performed with the results being appended to output stream *c*. In lines 14 – 20, these steps are repeated on the second half of the input streams, with the results bing appended to output stream *d*.

3. Problem Statement

The focus of this work is on optimizing utilization of the SRF. So only stream programs are relevant here. Given a stream program, this paper presents an algorithm that assigns the streams in the program to the SRF so as to minimize the total amount of space

taken by the streams. Such an algorithm can then be used by a stream compiler to produce a final SRF allocation by combining with live range splitting and strip mining, if necessary.

A stream program consists of a sequence of loops where each loop includes a sequence of kernels operating on streams. In a stream compiler, all loops are considered separately in SRF allocation. As shown in Figure 1, the DRAM controller supports two stream-level instructions, *Load* and *Store*, that transfer an entire stream between off-chip memory and the SRF. In stream programs as demonstrated in Figure 2, loads and stores are used to initialize some streams from the global input data residing in off-chip memory and write certain streams to off-chip memory, respectively.

The central machinery in our approach to allocating the streams in a loop to the SRF is the traditional interference graph (IG) except that it is a weighted (undirected) graph formed by the streams operated on by the kernels in the loop. All streams accessed in the loop are identified as live ranges to be placed in the SRF. If two live ranges interfere (i.e., overlap), they must be placed in non-overlapping SRF spaces. The live ranges of streams are computed by extending the *def/use* definitions for scalars to streams: *Load* defines a stream, *Store* uses a stream, and a kernel call (re)defines its output streams and uses its input streams. The live range of a stream starts from its definition and ends at its last use. Of course, streams are renamed using the SSA (static single assignment) form.

After the live ranges have been computed for a loop, its weighted (undirected) IG, denoted $\mathcal{G}$, is built in the normal manner, where a weighted node denotes a stream live range whose weight is the size of the stream and and an edge connects two nodes if their live ranges interfere with each other.

The SRF allocation problem can be naturally solved as an interval-coloring problem as formalized below. Allocating SRF spaces to stream live ranges in an IG is represented by an assignment of intervals to the nodes in the IG. Minimizing the span of intervals amounts to minimizing the required SRF size.

DEFINITION 1. Given a stream IG $\mathcal{G} = (V, E)$ with positively integral node weights $w : V \to \mathbb{N}$ (representing stream sizes), an *interval coloring* α of $\mathcal{G}$ maps each node x onto an interval α_x of the real line of width $w(x)$ such that adjacent nodes are mapped to disjoint intervals, i.e., $(x, y) \in E$ implies $\alpha_x \cap \alpha_y = \emptyset$.

It is well-known that interval coloring is NP-complete.

Our IG-based approach is flexible enough to accommodate prepass optimizations that are applied earlier to a program either by the programmer or the compiler. One example is to reorder some loads and stores to overlap memory transfers and kernel execution. Another is to split some long live ranges (in scientific applications) accomplished by inserting a pair of store and load instructions. We plan to automate their integration with this work in future.

4. Comparability Graph Coloring for Stream IGs

Section 4.1 recalls the basic results about interval coloring and comparability coloring [9], which provide a basis for understanding our approach and proving its optimality and near-optimality. Section 4.2 describes our key insight drawn from a careful analysis of the structure of stream IGs: a large number of stream IGs are comparability graphs, enabling their optimal colorings to be found in polynomial time. In Section 4.3, we turn this insight into an algorithm that can find optimal or near-optimal colorings for well-structured media and scientific applications when their stream IGs are expected to be decomposable into a set of comparability graphs.

4.1 Interval and Comparability Graph Coloring: Basics

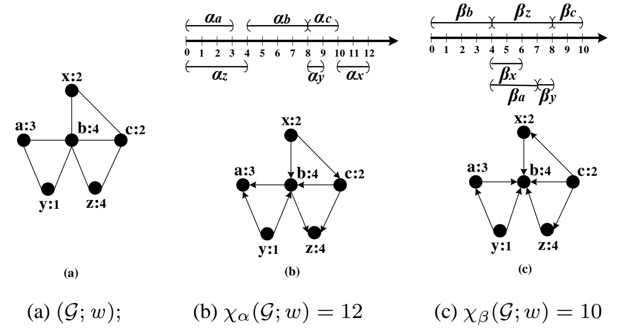
Given a (directed or undirected) graph $\mathcal{G} = (V, E)$ and a subset $A \subseteq V$, the *induced subgraph* by A is $\mathcal{G}(A) = (A, E(A))$, where $E(A) = \{(x, y) \in E \mid x, y \in A\}$. A subset $A \subseteq V$ of r nodes is an *r-clique* if it induces a complete subgraph. A clique is a *maximal clique* if it is not contained in any other clique.

Given an undirected graph $\mathcal{G} = (V, E)$ with the function w mapping nodes to positively integral weights, the total *width* (i.e., the number of hues) of an interval coloring α, $\chi_\alpha(\mathcal{G}; w)$, is $|\bigcup_{x \in V} \alpha_x|$. The *chromatic number* $\chi(\mathcal{G}; w)$ is the smallest width needed to color the nodes in $\mathcal{G}$. The *clique number* is defined as $\omega(\mathcal{G}; w) = \max\{w(K) \mid K \text{ is a clique of } \mathcal{G}\}$. As a fact, we have:

$$\chi(\mathcal{G}; w) \quad \geqslant \quad \omega(\mathcal{G}; w) \tag{1}$$

4.1.1 Interval Coloring vs. Acyclic Orientation

Figure 3 illustrates the equivalence between finding an interval coloring and finding an acyclic orientation for a weighted graph.



(a) $(\mathcal{G}; w)$; (b) $\chi_\alpha(\mathcal{G}; w) = 12$ (c) $\chi_\beta(\mathcal{G}; w) = 10$

Figure 3: Two interval colorings α and β of a weighted undirected graph together with their equivalent acyclic orientations.

Let $\mathcal{G}= (V, E)$ be an undirected graph. An *orientation* of $\mathcal{G}$ is a function α that assigns every edge a direction such that $\alpha(x, y) \in \{(x, y), (y, x)\}$ for all $(x, y) \in E$. Let $\mathcal{G}_\alpha$ be the digraph obtained by replacing each edge $(x, y) \in E$ with the arc $\alpha(x, y)$. An orientation α is *acyclic* if $\mathcal{G}_\alpha$ contains no directed cycles.

Every interval coloring α of $\mathcal{G}$ induces an acyclic orientation α' such that $(x, y) \in \alpha'$ if and only if $\alpha_x > \alpha_y$ i.e., an arc is directed from x to y if and only if α_x is to the right of α_y for all $(x, y) \in E$.

Conversely, an acyclic orientation α of $\mathcal{G}$ induces an interval coloring α'. For a sink node x, let $\alpha'_x = [0, w(x))$. Proceeding inductively, for a node y with all its successor nodes already being colored (i.e., α' defined at the successors), let t be the largest endpoint of their intervals and define $\alpha'_y = [t, t + w(y))$.

From an acyclic orientation, we can obtain an interval coloring in linear time by a depth-first search.

The problem of finding optimal colorings is NP-complete. In an optimal coloring, the chromatic number $\chi(\mathcal{G}; w)$ is related to the notion of *heaviest path* in an acyclic orientation of $\mathcal{G}$:

$$\chi(\mathcal{G}; w) \quad = \quad \min_{\alpha \in \mathcal{A}(\mathcal{G})} \big(\max_{\mu \in \mathcal{P}(\alpha)} \mathrm{w}(\mu) \big) \tag{2}$$

where $\mathcal{A}(\mathcal{G})$ is the set of acyclic orientations of $\mathcal{G}$ and $\mathcal{P}(\alpha)$ the set of directed paths in an orientation $\alpha \in \mathcal{A}(\mathcal{G})$. In other words, the orientation whose heaviest path is the smallest induces an optimal coloring. The heaviest-path-based formulation stated in (2) is exploited in the development of our coloring algorithm for stream IGs (Section 4.3).

In Figure 3(b), the heaviest path is $x \to c \to b \to z$ with a total weight of $\chi_\alpha(\mathcal{G}; w) = 12$. In Figure 3(c), the heaviest path is $c \to z \to b$ with a total weight of $\chi_\beta(\mathcal{G}; w) = 10$. The gap between the two colorings is 2 but can be larger (Figure 22). So there is a need to look for an optimal solution efficiently in practice.

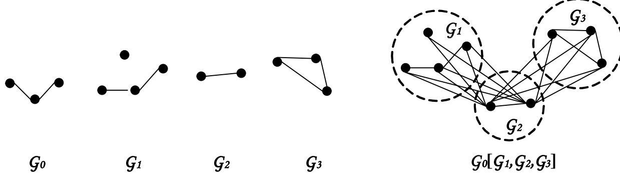


Figure 4: An illustration of Definition 4 ($n = 3$).

Due to the equivalence between acyclic orientations and interval colorings, we also write $\chi_\alpha(G; w)$ to mean the width of the interval coloring associated with an acyclic orientation α of G.

4.1.2 Comparability Graph Coloring

In the context of this work, we examine below a class of graphs that allows interval colorings to be found optimally in polynomial time.

DEFINITION 2. An orientation α of an undirected graph $\mathcal{G}$ is *transitive* if $(x, z) \in \mathcal{G}_\alpha$ whenever $(x, y), (y, z) \in \mathcal{G}_\alpha$.

DEFINITION 3. An undirected graph $\mathcal{G}$ is a *comparability graph* if there exists a transitive orientation of $\mathcal{G}$.

A transitive orientation is acyclic but the converse is not necessarily transitive. In Figure 3, α is not transitive since $(x, b), (b, a) \in \mathcal{G}_\alpha$ but $(x, a) \notin \mathcal{G}_\alpha$. However, β is transitive. Therefore, the graph shown in Figure 3(a) is a comparability graph.

Let α be a transitive orientation of a comparability graph $\mathcal{G}$. Restating (1), we have $\chi(\mathcal{G}; w) \geqslant \omega(\mathcal{G}; w)$. Due to transitivity, every path in $\mathcal{G}_\alpha$ is contained in a clique of $\mathcal{G}$. In particular, the heaviest path in $\mathcal{G}_\alpha$ equals to the heaviest clique in $\mathcal{G}$, i.e., $\chi(\mathcal{G}; w) \leqslant \chi_\alpha(\mathcal{G}; w) = \omega(\mathcal{G}; w)$. Hence, $\chi_\alpha(\mathcal{G}; w) = \chi(\mathcal{G}; w) = \omega(\mathcal{G}; w)$. This result is summarized below.

THEOREM 1. *For any transitive orientation α of $\mathcal{G}$, the interval coloring induced is optimal (and can be found in linear time).*

DEFINITION 4. *Let $\mathcal{G}_0$ be a graph with n nodes $v_1, v_2, \ldots, v_n$ and $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n$ be n disjoint graphs. These graphs may be directed or undirected. The* composition *graph $\mathcal{G} = \mathcal{G}_0[\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n]$, which is illustrated in Figure 4, is formed formally as follows. First, replace v_i in $\mathcal{G}_0$ with $\mathcal{G}_i$. Second, for all $1 \le i, j \le n$, make each node of $\mathcal{G}_i$ adjacent to each node of $\mathcal{G}_j$ whenever v_i is adjacent to v_j in $\mathcal{G}_0$. Formally, for $\mathcal{G}_i = (V_i, E_i)$, we define $\mathcal{G} = (V, E)$ as:*

$$
\begin{aligned}
V &= \cup_{1 \leqslant i \leqslant n} V_i \\
E &= \cup_{1 \leqslant i \leqslant n} E_i \cup \{(x, y) \mid x \in V_i, y \in V_j \textit{ and } (v_i, v_j) \in E_0\}
\end{aligned}
$$

THEOREM 2. *Let $\mathcal{G} = \mathcal{G}_0[\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n]$, where all $\mathcal{G}_i$'s are disjoint undirected graphs. Then $\mathcal{G}$ is a comparability graph if and only if each $\mathcal{G}_i (0 \le i \le n)$ is a comparability graph.*

Furthermore, the problems of recognizing a comparability graph $\mathcal{G} = (V, E)$ and finding a transitive orientation of $\mathcal{G}$ can both be done in $O(\delta \cdot \mid E \mid)$ time and $O(\mid V \mid + \mid E \mid)$ space, where δ is the maximum degree of a node in $\mathcal{G}$. Based on α, an optimal coloring of $\mathcal{G}$ can be obtained in linear time (Theorem 1).

4.2 Optimal Colorings of Comparability Stream IGs

In stream programs with producer-consumer locality but little global data reuse, the live ranges of streams are also local. A typical stream program (or a loop in such program) consists of a series of kernels, each producing intermediate streams to be consumed by the next kernel in sequence. We show below that if all stream live ranges in a stream IG do not span across more than two kernel calls, then the IG is a comparability graph and its optimal coloring can thus be found in polynomial time. This result is proved easily by a straightforward application of Theorem 2.

Figure 5 shows the IG for a series of three kernels, where all live ranges are no longer than two kernel calls. In particular, stream q is live from kernel '1' to kernel '2', streams u, v and w are live in kernels '2' and '3', and the remaining streams are only live at the kernels where they are operated on. In this example and the proofs of our results, whether a stream is an input or output is irrelevant.

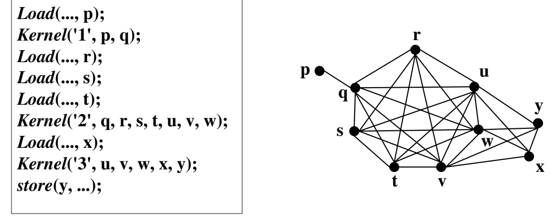


Figure 5: A stream program and its IG.

Let $\mathcal{G}_{\text{cg}}$ be the IG built from a loop containing N_{cg} kernels (numbered from 1) such that each live range in $\mathcal{G}_{\text{cg}}$ is not longer than two kernels. We partition all live ranges in $\mathcal{G}_{\text{cg}}$ into $2N_{\text{cg}}$ sets:

$$
K_1, K_{12}, K_2, K_{23}, K_3, \ldots, K_{(N_{\text{cg}}-1)N_{\text{cg}}}, K_{N_{\text{cg}}}, K_{N_{\text{cg}}1} \quad (3)
$$

where K_i consists of all streams accessed, i.e., live only in kernel i and $K_{i(i\oplus1)}$ all streams live only in kernels i and $i \oplus 1$. We define $i \oplus c$ to be $(i + c - 1)\% N_{\text{cg}} + 1$ and $i \ominus c$ to be $(i - c - 1)\% N_{\text{cg}} + 1$.

As illustrated in Figure 6, all streams accessed in a kernel in a loop form a maximal clique in the stream IG of the loop.

LEMMA 1. *The streams in $K_{(i\ominus1)i} \cup K_i \cup K_{i(i\oplus1)}$ form a maximal clique for every kernel i.*

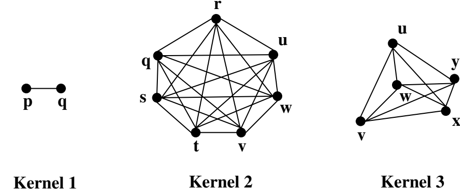


Figure 6: Kernel-induced cliques for the program in Figure 5.

Our main results are stated in two theorems, Theorem 3 is applicable when N_{cg} is even and Theorem 4 applicable when $K_{N_{\text{cg}}1} = \emptyset$, i.e., when cross-iteration reuse is absent. When neither condition holds, we can apply loop unrolling once to produce a loop with an even number of kernels so that Theorem 3 can be applied. For stream processors, unrolling a stream program that is executed on the host processor does not affect negatively program performance. (Code size expansion for the host is not a concern.)

THEOREM 3. *If N_{cg} is even, then $\mathcal{G}_{\text{cg}}$ is a comparability graph.*

PROOF. Let us assume first that all sets listed in (3) are not empty. By construction, the live ranges in every such a set are equal. Thus, the induced subgraph of $\mathcal{G}_{\text{cg}}$ by K_i ($K_{i(i\oplus1)}$) is a clique, denoted $\mathcal{G}_i$ ($\mathcal{G}_{i(i\oplus1)}$). So we have the following $2N_{\text{cg}}$ induced cliques:

$$
\mathcal{G}_1, \mathcal{G}_{12}, \mathcal{G}_2, \mathcal{G}_{23}, \mathcal{G}_3, \ldots, \mathcal{G}_{(N_{\text{cg}}-1)N_{\text{cg}}}, \mathcal{G}_{N_{\text{cg}}}, \mathcal{G}_{N_{\text{cg}}1} \quad (4)
$$

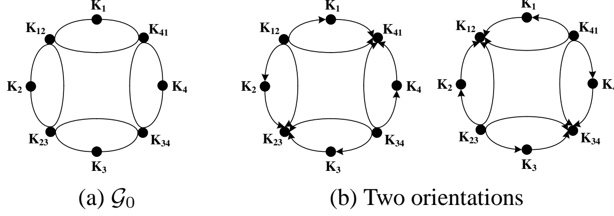
In addition, for any two sets K and K' listed in (3), either every live range $x \in K$ interferes with every live range $x' \in K'$ or there is no interference between the live ranges in K and those in K'.

By Theorem 2, in $\mathcal{G}_{\text{cg}}$, if we let $\mathcal{G}_i$ ($\mathcal{G}_{i(i\oplus1)}$) "collapse" into one node, identified by K_i ($K_{i(i\oplus1)}$), and denote the resulting

"decomposed graph" by $\mathcal{G}_0$, we have:

$$\mathcal{G}_{\text{cg}} = \mathcal{G}_0[\mathcal{G}_1, G_{12}, G_2, \ldots, \mathcal{G}_{N_{\text{cg}}}, \mathcal{G}_{N_{\text{cg}}1}]$$

A clique is a comparability graph. Thus, $\mathcal{G}_i, i = 1, 12, 2, \ldots, N_{\text{cg}}1$ given in (4) are all comparability graphs. Then, by Theorem 2, $\mathcal{G}_{\text{cg}}$ is a comparability graph if we show that $\mathcal{G}_0$ is. To achieve this, by Definition 3, it suffices if we can find a transitive orientation of



(a) $\mathcal{G}_0$ (b) Two orientations

Figure 7: Two transitive orientations of $\mathcal{G}_0$ ($N_{\text{cg}} = 4$).

$\mathcal{G}_0$. As shown in Figure 7, there are exactly two different transitive orientations since $K_{12}, K_{23}, \ldots, K_{N_{\text{cg}}1}$ must alternate to be a source or a sink (Lemma 2). This is possible since N_{cg} is even.

Finally, if any set listed in (3) is empty, the decomposed graph $\mathcal{G}_0$ is still a comparability graph since every induced subgraph of a comparability graph is a comparability graph. □

THEOREM 4. *If $K_{N_{\text{cg}}1} = \emptyset$, then $\mathcal{G}_{\text{cg}}$ is a comparability graph.*

PROOF. A transitive orientation of $\mathcal{G}_{\text{cg}}$ as shown in Figure 7 always exists even if N_{cg} is odd since the "ring" is broken at $K_{N_{\text{cg}}1}$. □

In fact, Theorem 4 holds as long as $K_{i(i\oplus 1)} = 0$ for some i.

Let us illustrate Theorem 4 in Figure 8 for the IG in Figure 5. Being a comparability graph, its optimal coloring is guaranteed. The optimality is independent of the node weights in the graph.

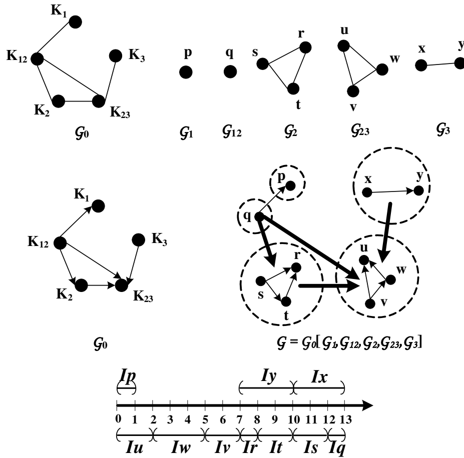


Figure 8: Optimal interval coloring of the stream IG given in Figure 5 (with the weights of p, q and r being 1, the weights of s, t, u and v being 2 and the weights of w, x and y being 3). To avoid cluttering, in the graph labelled by $\mathcal{G}_0[\mathcal{G}_1, \mathcal{G}_{12}, \mathcal{G}_2, \mathcal{G}_{23}, \mathcal{G}_3]$, a thicker arrow directing from a clique K to a clique K' symbolizes all directed edges (x, y), for all $x \in K$ and all $x' \in K'$.

The facts stated in Lemmas 2 and 3 are exploited in the development of our algorithm for coloring stream IGs in Section 4.3.

LEMMA 2. *Suppose $\mathcal{G}_{\text{cg}}$ is a comparability graph. Let $\mathcal{G}'_{\text{cg}}$ be an induced subgraph of $\mathcal{G}_{\text{cg}}$. If $\mathcal{G}'_{\text{cg}}$ is connected, then it has at most eight different transitive orientations.*

PROOF. If $\mathcal{G}'_{\text{cg}}$ is connected, there must exist two kernels i and j such that $K_{i(i\oplus 1)}, K_{(i\oplus 1)(i\oplus 2)}, \ldots K_{(j\ominus 2)(j\ominus 1)}, K_{(j\ominus 1)j}$ are in $\mathcal{G}'_{\text{cg}}$ and that these are the only sets containing two-kernel long live ranges listed in (3). Let us consider only the worst case when $K_i = K_j = \emptyset$. In a transitive orientation of $\mathcal{G}'_{\text{cg}}$, the middle $j - i - 2$ sets in the above list must alternate to serve as a source or a sink. So there are only two possibilities. In either case, edge $(K_{i(i\oplus 1)}, K_{i\oplus 1})$ may have at most two orientations, and similarly, edge $(K_{j\ominus 1}, K_{(j\ominus 1)j})$ may have at most two orientations. So there are at most $2 \times 2 \times 2 = 8$ different transitive orientations. □

LEMMA 3. *Suppose $\mathcal{G}_{\text{cg}}$ is a comparability graph. If all sets in (3) are nonempty, then $\mathcal{G}$ has exactly two transitive orientations.*

Proof. $\mathcal{G}_{\text{cg}}$ is connected and then apply Lemma 2 (Figure 7). □

4.3 A General Algorithm for Coloring Stream IGs

In some scientific applications (amenable to stream processing), the presence of temporal reuse in a few streams could make their live ranges longer than two kernels. In some media applications, there are also occasionally a few long producer-consumer live ranges. Furthermore, some live ranges may be extended by the programmer or a pre-pass compiler optimization in order to overlap memory transfers and kernel execution. Such stream IGs may or may not be comparability graphs. In this section, we generalize our work described in the preceding section to deal with these stream IGs.

The basic idea is to partition the node set V in $\mathcal{G} = (V, E)$ into:

$$\begin{aligned} V_s &= \{v \in V \mid v\text{'s live range spans at most two kernels}\} \\ V_l &= \{v \in V \mid v\text{'s live range spans more than two kernels}\} \end{aligned}$$

As a result, E is partitioned into the following three subsets:

$$\begin{aligned} E_s &= \{(x, y) \in E \mid x \in V_s, y \in V_s\} \\ E_l &= \{(x, y) \in E \mid x \in V_l, y \in V_l\} \\ E_{sl} &= \{(x, y) \in E \mid x \in V_s, y \in V_l\} \end{aligned}$$

By Theorems 3 and 4, the subgraph $\mathcal{G}(V_s)$ induced by V_s is a comparability graph. Our key observation is that the long live ranges in stream IGs are sparse and tend not to be live simultaneously. So we assume that the subgraph $\mathcal{G}(V_l)$ is a forest of disjoint trees (which are trivially comparability graphs). In the rare cases when $\mathcal{G}(V_l)$ is not a forest, we may, as part of future work, apply live range splitting to shorten some live ranges to make it so.

```
Load(..., k);
Kernel('1', k, l);
Load(..., m);
Load(..., n);
Kernel('2', l, m, n, o);
Load(..., p);
Load(..., q);
Kernel('3', o, p, q, r);
Load(..., s);
Kernel('4', r, s, t);
Load(..., u);
Kernel('5', m, t, u, v);
Load(..., w);
Kernel('6', p, v, w, x);
Kernel('7', x, y);
```

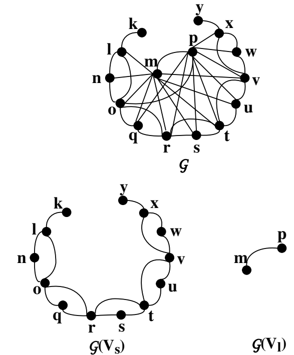


Figure 9: A program with two long live ranges m and p and its IG.

As illustrated in Figure 9, V_l consists of two long live ranges m and p: m is live from kernel 2 to kernel 5 and p is live from kernel 3 to kernel 6. Since both streams interfere with each other, the forest $\mathcal{G}(V_l)$ has only one tree, which is a line connecting m and p.

Algorithm 1 A general algorithm for coloring stream IGs.

1: **procedure** IG_CGC
2: **Input:** $\mathcal{G} = (V, E)$ with $V = \{V_s, V_l\}$ and $E = \{E_s, E_l, E_{sl}\}$
3: **Output:** An acyclic orientation, or equivalently, interval coloring α of $\mathcal{G}$
4: **if** a transitive orientation α of $\mathcal{G}$ can be found **then**
5:     **return** α
6: **end if**
7: $\chi_{\min}(\mathcal{G}) = +\infty$
8: Let $\mathcal{O}_s$ be the set of all transitive orientations of E_s, i.e., $\mathcal{G}(V_s)$
9: Let $\mathcal{O}_l$ be the set of $2^{\mathsf{trees}(\mathcal{G}(V_l))}$ transitive orientations of E_l, i.e., $\mathcal{G}(V_l)$
10: **for** each orientation $o_s \times o_l \in \mathcal{O}_s \times \mathcal{O}_l$ **do**
11:     **for** $(x, y) \in E_{sl}$, where $x \in V_s$ and $y \in V_l$ **do**
12:         Direct an arc from y to x (x to y) if y is a source (sink)
13:     **end for**
14:     Let α be the acyclic orientation of $\mathcal{G}$ thus found
15:     **if** $\chi_\alpha(\mathcal{G}) < \chi_{\min}(\mathcal{G})$ **then**
16:         $\chi_{\min}(\mathcal{G}) = \chi_\alpha(\mathcal{G})$
17:         Record the α as the current best
18:     **end if**
19: **end for**
20: **return** α
21: **end procedure**

In Section 4.3.1, we present a polynomial algorithm for coloring our stream IGs. In Section 4.3.2, we argue that why this algorithm tends to give optimal and near-optimal colorings in practice.

4.3.1 Algorithm

As shown in Algorithm 1, if $\mathcal{G}$ is a comparability graph (Definition 3), then by Theorem 1, an optimal coloring, represented by a transitive orientation, is returned immediately (lines 4 – 6). Otherwise, $\mathcal{G}$ is not a comparability graph, in which case, an optimal or near-optimal coloring, represented by an acyclic orientation, is found in three steps. Let $\mathsf{trees}(\mathcal{G}(V_l))$ be the number of trees in $\mathcal{G}(V_l)$. The basic idea is to enumerate the set $\mathcal{O}_s$ of all transitive transitive orientations of E_s, i.e., $G(V_s)$ (in Step 1) and enumerate the set $\mathcal{O}_l$ of all $2^{\mathsf{trees}(\mathcal{G}(V_l))}$ transitive orientations for the forest E_l, i.e., $\mathcal{G}(V_l)$ (in Step 2). As a result, for every possible combination $o_s \times o_l$ in $\mathcal{O}_s \times \mathcal{O}_l$, a unique orientation to E_{sl} is determined (in Step 3). Among $|\mathcal{O}_s| \times |\mathcal{O}_l|$ acyclic orientations of $\mathcal{G}$ found, the one whose heaviest (directed) path is the smallest is returned (lines 15 – 17). This is motivated by the fact stated in (2).

We restrict ourselves to transitive orientations in $\mathcal{G}(V_s)$ and $\mathcal{G}(V_l)$ only to both reduce the solution space to be searched for and minimize the width of the final interval coloring found.

In Step 1 (line 8), the set $\mathcal{O}_s$ of all transitive orientations of E_s, i.e., $\mathcal{G}(V_s)$ is found. In real code, $\mathcal{G}(V_s)$ is generally connected, resulting in exactly two transitive orientations by Lemma 3 as illustrated in Figure 7. There can be only a limited number of transitive orientations when $\mathcal{G}(V_s)$ is disconnected by Lemma 2.

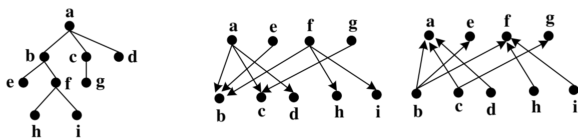


Figure 10: Two transitive orientations of a tree.

In Step 2 (line 9), we find all $2^{\mathsf{trees}(\mathcal{G}(V_l))}$ transitive orientations of the trees in $\mathcal{G}(V_l)$, i.e., E_l. There are these many orientations because a tree is a bipartite graph and has exactly two orientations (when it has more than one node). As shown in Figure 10, in one orientation, the edges are directed from the nodes at the odd-numbered levels and in the other, the edge directions are reversed.

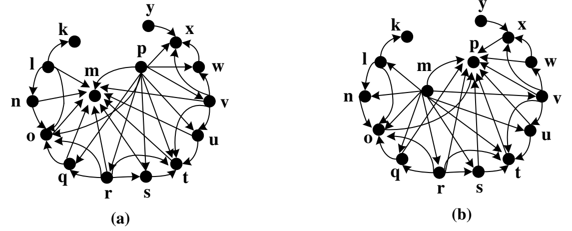


Figure 11: Two orientations for the program in Figure 1.

In Step 3 (lines 10 – 13), for each orientation $o_s \times o_l \in \mathcal{O}_s \times \mathcal{O}_l$ (line 10), a unique orientation of E_{sl} is determined (lines 11 – 13). For each edge $(x, y) \in E_{sl}$, where $x \in V_s$ and $y \in V_l$, its orientation is assigned based on the property of y. From Figure 10, it can be easily observed that y is either a source or a sink under o_l. If y is a source, direct the edge from y to x, namely, maintain y's property; otherwise, direct the edge from x to y.

Every orientation α of $\mathcal{G}$ found in line 14 is acyclic. This can be reasoned about as follows. No directed path confined to $\mathcal{G}(V_s)$ can be a cycle since $\mathcal{G}(V_s)$ is a comparability graph. In addition, no directed path that contains a node in $\mathcal{G}(V_l)$ can be a cycle since the node must be either a source or a sink (Figure 10).

IG_CGC is polynomial in practice. For comparability graphs, their recognition and optimal colorings are polynomial. In addition, $\mathcal{G}_s$ is mostly connected, resulting in a few orientations (Lemmas 2 and 3). Finally, $2^{\mathsf{trees}(\mathcal{G}(V_l))}$ is a small constant since $\mathcal{G}(V_l)$ has few trees.

Let us apply IG_CGC to the program given in Figure 9. In lines 4 – 6, $\mathcal{G}$ is detected to be a comparability graph. Its optimal coloring is found and returned immediately. Nevertheless, let us use this example to explain how the remaining steps of the algorithm work. $\mathcal{G}(V_s)$ is connected and happens to have only two transitive orientations. As $\mathcal{G}(V_l)$ has only one tree, there are two orientations. So there are a total of four orientations, two of which are shown in Figure 11, to consider in line 10. In this particular example, the one in Figure 11(a) is a possible solution found since it is transitive.

4.3.2 Analysis

In this section, we argue that IG_CGC finds optimal colorings for most stream programs. We show further that non-optimal colorings occur only infrequently and are near-optimal in the sense that they are only larger by the sum of one or two stream sizes in the worst case. Our claim is validated in our experiments in Section 5.

Recall that $\chi(\mathcal{G}; w)$ denotes the chromatic number of $\mathcal{G}$. In our algorithm, α is the best acyclic orientation found and $\chi_\alpha(\mathcal{G}; w)$ is its width. Let P_α be the heaviest directed path in $\mathcal{G}_\alpha$:

$$P_\alpha \quad =_{\text{def}} \quad v_1, v_2, \ldots, v_m \tag{5}$$

According to (2), we have $\chi_\alpha(\mathcal{G}; w) = w(P_\alpha)$. In addition, $w(P_\alpha)$ is the smallest among the heaviest directed paths in all orientations of $\mathcal{G}$ found in line 14 of IG_CGC given in Figure 1.

IG_CGC is *optimal* for $\mathcal{G}$ if $\chi_\alpha(\mathcal{G}; w) = \chi(\mathcal{G}; w)$.

All results presented below in this section are formulated and proved based on reasoning about the structure of P_α, which is uncovered in Lemma 4, resulting in five cases to be distinguished, and Lemma 5. In three of the five cases, our algorithm is optimal (Theorem 5). In the remaining two cases, our algorithm is also optimal for many stream IGs. Non-optimal solutions α are returned only infrequently when some strict conditions are met, and moreover, these solutions are near-optimal since $\chi_\alpha(\mathcal{G}; w) - \chi(\mathcal{G}; w)$ is small for reasonably large stream IGs (Theorems 6 and 7).

LEMMA 4. *Only v_1 or v_m may appear in the forest $\mathcal{G}(V_l)$.*

PROOF. Follows simply from the fact that for every orientation α of $\mathcal{G}$ found in line 14 of IG_CGC given in Figure 1, every node in $\mathcal{G}(V_l)$ is either a source or a sink under α. □

This lemma implies that all nodes in P_α are contained in $\mathcal{G}(V_s)$ except its start and end nodes.

Let $\mathcal{K}_i = K_{(i\ominus 1)i} \cup K_i \cup K_{i(i\oplus 1)}$. By Lemma 1, $\mathcal{K}_i$ is a maximal clique in $\mathcal{G}(V_s)$ (as illustrated in Figure 6).

LEMMA 5. *$v_2, v_3, \ldots, v_{m-1}$ form a clique $\mathcal{K}_i$ for some i.*

PROOF. α found by IG_CGC is a transitive orientation of $\mathcal{G}(V_s)$.

First, there cannot be two different nodes v_i and v_j in $v_2, v_3, \ldots, v_{m-1}$, where $i < j$, such that $v_i \in K_i$ and $v_j \in K_j$. This is because $(v_i, v_j) \notin \mathcal{G}(V_s)$. Otherwise, α cannot be transitive.

Second, there cannot be three distinct nodes v_i, v_j and v_k, where $i < j < k$, in $v_2, v_3, \ldots, v_{m-1}$ such that $v_i \in K_{i(i\oplus 1)}$, $v_j \in K_{j(j\oplus 1)}$ and $v_k \in K_{k(k\oplus 1)}$, because $(v_i, v_k) \notin \mathcal{G}(V_s)$.

Third, it is not possible for $v_2, v_3, \ldots, v_{m-1}$ to be contained in two "non-consecutive" $K_{(i\ominus 1)i}$ and $K_{j(j\oplus 1)}$, where $j \neq i$ and $i \ominus 1 \neq j \oplus 1$. Otherwise, we would end up in a situation that contradicts to the fact just established (in the second step).

So $v_2, v_3, \ldots, v_{m-1}$ must be contained in $\mathcal{K}_i$ for some i.

Finally, $\mathcal{K}_i$ is a clique, which must be formed by $v_2, v_3, \ldots, v_{m-1}$ since P_α is the heaviest path found by α. □

We distinguish five cases depending on the structure of P_α:

Case P1. P_α is contained in $\mathcal{G}(V_s)$

Case P2. P_α is contained in $\mathcal{G}(V_l)$

Case P3. $v_1, v_m \in \mathcal{G}(V_l)$ interfere with each other

Case P4. $v_1, v_m \in \mathcal{G}(V_l)$ do not interfere with each other

Case P5. Either v_1 or v_m is in $\mathcal{G}(V_l)$ (but not both)

THEOREM 5. *IG_CGC is optimal in Cases P1 – P3.*

PROOF. In Case P1, $\mathcal{G}(V_s)$ is a comparability graph. Thus, P_α must be contained in a clique K in $\mathcal{G}(V_s)$ (and also in $\mathcal{G}$). This means that $\chi_\alpha(\mathcal{G}; w) = w(P_\alpha) = w(K)$. Since $w(K) \leqslant \chi(\mathcal{G}; w)$, we must have $\chi_\alpha(\mathcal{G}; w) \leqslant \chi(\mathcal{G}; w)$. So α is optimal. The proof for Case P2 is similar since the forest $\mathcal{G}(V_l)$ is also a comparability graph. In Case P3, if v_1 and v_m interfere with each other, according to Lemma 5, it is easily deduced that v_1 and v_m must interfere with all nodes, i.e., $v_1, v_2, \ldots, v_{m-1}$, in $\mathcal{K}_i$, and consequently, all the nodes in P_α. Thus P_α must appear together in a clique K in $\mathcal{G}$. Thus, we can complete the rest of the proof for Case P3 in a similar way as for Case P1. □

THEOREM 6. *In Case P4, we have $\chi_\alpha(\mathcal{G}; w) - \chi(\mathcal{G}; w) \leqslant w(v_1) + w(v_m)$, where the equality holds if and only if $v_2, v_3, \ldots v_{m-1}$ happen to from the heaviest clique $\mathcal{K}$ in $\mathcal{G}$ such that $\chi(\mathcal{G}; w) = w(\mathcal{K})$.*

PROOF. By Lemma 5, we have $\chi_\alpha(\mathcal{G}; w) - \chi(\mathcal{G}; w) \leqslant w(v_1) + w(v_m)$. We now prove the "if" and "only if" for the equality.

The "if" part is true since $\chi_\alpha(\mathcal{G}; w) = w(P_\alpha) = w(v_1) + w(v_m) + w(\mathcal{K}) = w(v_1) + w(v_m) + \chi(\mathcal{G}; w)$. The "only if" part is true due to Lemma 5 and the given hypothesis $\chi_\alpha(\mathcal{G}; w) - \chi(\mathcal{G}; w) = w(v_1) + w(v_m)$. □

An analogue of Theorem 6 for Case P5 is given below.

THEOREM 7. *In Case P5, suppose that v_1 is contained in $\mathcal{G}(v_l)$ but v_m is not. Then $\chi_\alpha(\mathcal{G}; w) - \chi(\mathcal{G}; w) \leqslant w(v_1)$, where the equality holds if and only if $v_2, v_3, \ldots v_{m-1}$ happen to form the heaviest clique $\mathcal{K}$ in $\mathcal{G}$ such that $\chi(\mathcal{G}; w) = w(\mathcal{K})$.*

5. Experiments

Research into advanced compiler technology for stream processing is still at its infancy. There are presently no standard benchmarks available. Table 1 gives a list of 11 media and scientific applications available to us for the FT64 stream processor. NLAG-5 is a nonlinear algebra solver for two-dimensional nonlinear diffusion of hydrodynamic. QMR is the core iteration in the QMRCGSTAB algorithm for solving nonsymmetric linear systems. LUD is a dense LU Decomposition solver. As shown, the stream IGs in 10 benchmarks are comparability graphs. Their optimal colorings are guaranteed. Initially, the stream IG $\mathcal{G}$ for QMR is not a comparability graph and $\mathcal{G}(V_l)$ is not a forest. However, after unrolling its loop with a factor of two and performing some live range splitting, $\mathcal{G}(V_l)$ is a forest in the unrolled loop as depicted in Figure 12, to which our algorithm can now be applied. In fact, the IG for the unrolled loop happens to be a comparability graph. A transitive orientation of this comparability graph is shown in Figure 13. This example provides evidence that loop unrolling and live-range splitting can be used as enabling transformations to enable more applications to be optimally colored.

Benchmark	Source	IG
Laplace	NCSA	C
Swim-calc1	Spec2000	C
Swim-calc2	Spec2000	C
GEMM	BLAS	C
FFT	-	C
EP	NPB	C
NLAG-5	-	C
QMR	-	F
LUD	-	C
Jacobi	-	C
MG	NPB	C

Table 1: Media and scientific programs (C (F) indicates the corresponding stream IG $\mathcal{G}$ ($\mathcal{G}(V_l)$) is a comparability graph (forest)).

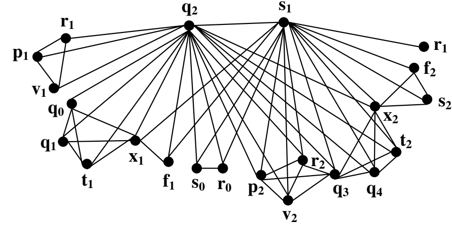


Figure 12: The IG for the unrolled QMR.

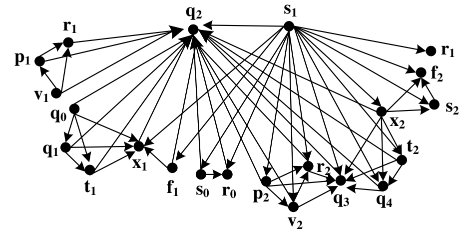


Figure 13: A transitive orientation of the IG for the unrolled QMR.

Below we demonstrate that IG_CGC can find optimal and nearly optimal colorings efficiently for a large number of randomly generated stream IGs that satisfy the characteristics of stream processing.

We have implemented an algorithm that randomly generates the stream IGs that satisfy the stream characteristics exploited in the development of our IG_CGC algorithm as discussed in Section 4.3.

All random numbers are in discrete uniform distribution generated by *unidrnd* in Matlab unless specified otherwise.

There are five steps involved in generating a stream IG $\mathcal{G}$. Step 1 generates the number of kernels, denoted *num_kernel*. In Step 2, we generate the set of short live ranges, namely $G(V_s)$. For each kernel i, we generate two sets K_i and $K_{i(i\oplus 1)}$. We generate a random number in the range [1,3] to represent the number of live ranges in K_i. Similarly, we generate another random number over [1,3] to represent the number of live ranges in $K_{i(i\oplus 1)}$. As a result, each kernel has at most nine short live range streams live at the kernel: three from $K_{(i\ominus 1)i}$, three from K_i and three from $K_{i(i\oplus 1)}$.

In Step 3, we generate the set of long live ranges, namely $G(V_l)$. We generate a random number p ranging from 1% to 20% to represent the percentage of long live ranges in $\mathcal{G}(V_l)$ over *num_kernel*. Thus, $|\mathcal{G}(V_l)| = p \times$ *num_kernel*. For each long live range i, we generate a random number, *length_i*, over [3,6] to represent the number of kernels spanned by i (longer live ranges should be split) and another random number over [1, *num_kernel*-*length_i*+1] to represent the kernel from which i starts to be live.

In Step 4, we keep $\mathcal{G}$ if $\mathcal{G}(V_l)$ is a forest and go back to Step 1 otherwise. In Step 5, we generate the stream sizes for all live ranges according to their characteristics in stream applications. In our experiments, node weights are chosen to have two different distributions, *Distribution* $\mathcal{U}$ and *Distribution* $\mathcal{L}$. We use Distribution $\mathcal{U}$, a discrete uniform distribution, to demonstrate generally the worst-case performance advantages of IC_CGC over First-Fit. In this case, node weights are randomly taken from the range [1,6]. For each program, we modify Distribution $\mathcal{U}$ to obtain Distribution $\mathcal{L}$ by simply replacing each stream size w by 2^w. In this second case, the fact that some streams may be geometrically larger than others in a program is explicitly taken into account. Distribution $\mathcal{L}$ is actually a uniform distribution in the logrithmic scale $\log_2$.

To test the scalability of IG_CGC, we have generated four groups of stream IGs. Groups $G^1_{\mathcal{U}}$ and $G^1_{\mathcal{L}}$ consist of IGs with between 3 to 50 kernels with their node weights generated using Distributions $\mathcal{U}$ and $\mathcal{L}$, respectively. Groups $G^2_{\mathcal{U}}$ and $G^2_{\mathcal{L}}$ consist of larger IGs with between 50 to 100 kernels. Each group consists of exactly 30 different IGs (with their node weights being ignored). For each IG in each group, there are 10 instances of that IG instantiated with different node weights (in Step 5). So each group consists of 300 different weighted IGs, giving rise to a total of 1200 stream IGs considered in our experiments. Due to space limitation, we restrict our discussions to Group $G^1_{\mathcal{L}}$ and Group $G^2_{\mathcal{U}}$. The node counts of the graphs in these two groups are shown in Figures 14 and 18, respectively, and their edge counts in Figures 15 and 19, respectively. The tree counts in the forests $\mathcal{G}(V_l)$ in these graphs are shown in Figures 16 and 20, respectively.

To check the optimality of IG_CGC, we have developed a formulation of the SRF allocation problem by integer linear programming (ILP). We ran the commercial ILP solver, CPLEX 10.1, to find an optimal coloring for each IG. If CPLEX does not terminate in five hours for an IG $\mathcal{G}$, its optimal coloring is estimated optimistically by (1). Therefore, all optimality results about IG_CGC reported here are conservative.

Table 2 shows that IG_CGC obtains optimal solutions in 99% of the 1200 IGs in all four groups. On the other hand, the solutions from First-Fit are mostly sub-optimal. First-Fit obtains optimal solutions in 26% of the 1200 IGs in all four groups.

The near-optimality of IG_CGC is achieved efficiently as validated in our experiments on a 3.2GHz Pentium 4 with 1GB memory. The longest time taken is 0.2 seconds for an IG with 409 nodes and 1836 edges, in which case $G(V_l)$ consists of 8 trees. For most of the other IGs, the times elapsed are less than 0.05 seconds each.

Let us look at the differences between IG_CGC and First-Fit in terms of their allocation results. For a given weighted IG $\mathcal{G}$, the quality of our solution α found by IG_CGC is measured as a *gap* with respect to that found by First-Fit defined as follows:

Group	#Weighted IGs	Optimal Solutions (%)	
		IG_CGC	First-Fit
$G^1_{\mathcal{L}}$	300	299 (99.67%)	55 (18.33%)
$G^1_{\mathcal{U}}$	300	298 (99.33%)	115 (38.33%)
$G^2_{\mathcal{L}}$	300	296 (98.67%)	38 (12.67%)
$G^2_{\mathcal{U}}$	300	295 (98.33%)	104 (34.67%)

Table 2: Optimality of IG_CGC and First-Fit for 1200 IGs.

Group	#Unweighted Graphs	$\geqslant 20\%$	$\geqslant 10\%$	$< 0\%$
$G^1_{\mathcal{L}}$	30	18	30	1
$G^1_{\mathcal{U}}$	30	0	18	2
$G^2_{\mathcal{L}}$	30	4	27	2
$G^2_{\mathcal{U}}$	30	1	19	1

Table 3: Gaps between IG_CGC and First-Fit.

$$\mathsf{gap}(\mathcal{G}) \quad = \quad \frac{\chi_{\text{First–Fit}}(\mathcal{G}; w) - \chi_{\alpha}(\mathcal{G}; w)}{\chi_{\text{First–Fit}}(\mathcal{G}; w)} \tag{6}$$

where $\chi_{\text{First–Fit}}(\mathcal{G}; w)$ is the optimal solution (i.e., the smallest width required for coloring all nodes in $\mathcal{G}$) found by First-Fit and $\chi_{\alpha}(\mathcal{G}; w)$ is the optimal solution found by IG_CGC.

Table 3 shows the gaps between IG_CGC and First-Fit for the four groups, $G^1_{\mathcal{L}}$, $G^1_{\mathcal{U}}$, $G^2_{\mathcal{L}}$ and $G^2_{\mathcal{U}}$, with 30 unweighted graphs in each group. As shown in Column 3, for 18 out of 30 graphs in Group $G^1_{\mathcal{L}}$, there are always weight assignment(s) achieving a gap of over 20%. The largest for this set of 30 graphs is 41% for a graph consisting of 114 nodes and 455 edges. In addition, as shown in Column 4, for at least 18 out of 30 graphs in each group, there are some weight assignment(s) achieving a gap of over 10%. Finally, we observe from Column 5 that IG_CGC may perform slightly worse than First-Fit in six different weighted graphs. The gaps for five of these graphs are between -5.69% — -2.78% and the gap for the remaining one is -13.89%. In general, the gaps depend on the distribution of the node weights, i.e., the sizes of streams in a program. The major advantage of IG_CGC is that if an IG is a comparability graph, then the optimal allocation is guaranteed regardless of what its node weights are, and in addition, even when an IG is not a comparability graph, IG_CGC can still achieve near-optimal colorings as proved in Theorems 6 and 7 and confirmed by the experimental data in Table 2. However, the performance of First-Fit is sensitive to the structure of an IG and the values of its node weights. For example, the gap as shown in Figure 22 is 50%.

Let us see why IG_CGC achieves better SRF allocation than First-Fit. First-Fit places the streams in an IG in a certain order, say, according to the order in which the streams start to live and then their weights without considering the structure of the IG, resulting in SRF fragmentation. We demonstrate this with a simple program shown in Figure 22(a). Figure 22(b) shows the SRF allocation under First-Fit. The streams S_1 and S_2 live at kernel 1 are allocated before S_3. S_1, which is heavier than S_2, is allocated first followed by S_2. However, since S_2 is also live in kernel 2, S_3, which is heavier than S_1, can only be placed after S_2. Based on the IG and the assigned transitive orientation in Figure 22(c), the optimal SRF allocation found by IG_CGC is shown in Figure 22(d).

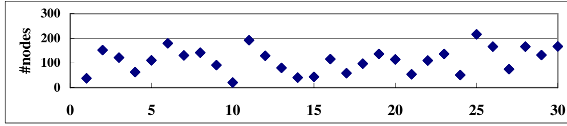


Figure 14: Number of nodes in Group $G^1_{\mathcal{L}}$.

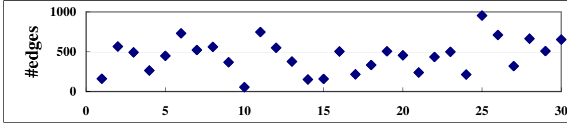


Figure 15: Number of edges in Group $G^1_{\mathcal{L}}$.

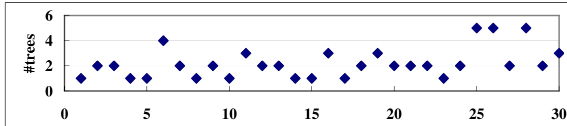


Figure 16: Number of trees in Group $G^1_{\mathcal{L}}$.

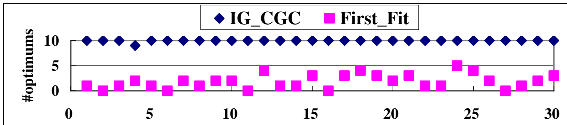


Figure 17: Number of optimal solutions found by IG_CGC and First-Fit in 300 weighted IGs from Group $G^1_{\mathcal{L}}$.

6. Related Work

Let us examine in more detail the two existing SRF management techniques for stream processors [5, 22]. Stream scheduling introduced in [5] was earlier implemented in the StreamC compiler to compile stream programs for Imagine [5, 18]. Stream scheduling associates (if possible) all stream accesses to the same stream with the same buffer in the SRF. All such SRF buffers are placed in the SRF by applying some greedy First-Fit-like bin-packing heuristics. The key idea is trying to position each buffer at the smallest possible SRF address, always complete the current buffer before starting another, and finally, position the largest buffers first so that smaller buffers can fill in the cracks. On encountering spills, their algorithm resorts to double buffering to reduce the sizes of some buffers. Strip mining is applied before SRF allocation.

In [22], we apply graph coloring to place streams in the SRF. This entails a partitioning of the SRF into pseudo registers before graph coloring can be applied. For some applications, such SRF partitioning introduces aliases among pseudo registers, resulting in SRF fragmentation and wasted free spaces. The idea of partitioning a software-managed cache first this way and then applying graph coloring to perform cache allocation was first proposed in [14]. In [26], we focused on how to identify and represent loop-dependent reuse between streams in stream programs.

In this paper, we present a comparability graph coloring algorithm for SRF allocation developed based on a careful analysis of the characteristics of stream IGs. This new approach relies on neither First-Fit heuristics, which can be sub-optimal, nor SRF partitioning, which can cause SRF fragmentation. Our algorithm can achieve near-optimal colorings as validated in our experiments and outperform First-Fit in a large number of stream IGs tested.

Graph coloring is a popular technique used in register allocation. Based on Chaitin's original formulation [2], a variety of graph

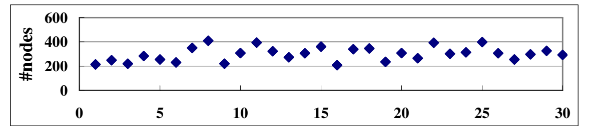


Figure 18: Number of nodes in Group $G^2_{\mathcal{U}}$.

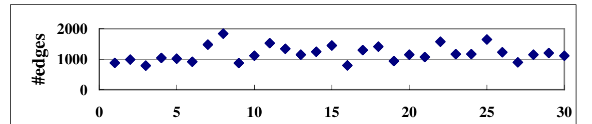


Figure 19: Number of edges in Group $G^2_{\mathcal{U}}$.

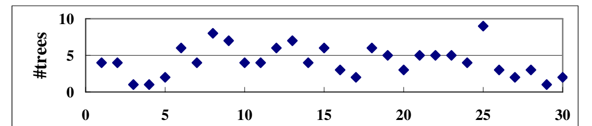


Figure 20: Number of trees in Group $G^2_{\mathcal{U}}$.

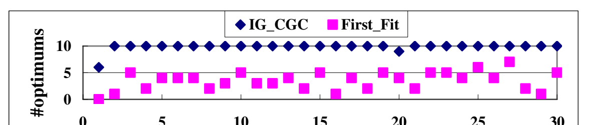


Figure 21: Number of optimal solutions found by IG_CGC and First-Fit in 300 weighted IGs from Group $G^2_{\mathcal{U}}$.

coloring based register allocators have been developed [1, 3, 7]. Recently, Smith et al. [19] present a generalized algorithm for irregular architectures with register aliases and non-disjoint register classes. Li et al. [14, 16] apply this generalized algorithm to assign arrays in embedded programs to scratchpad memory (SPM). Wang et al. [22] apply it further in SRF allocation as discussed above.

Fabri [6] discovered the connection between interval coloring and compile-time memory allocation. Since then some approximation algorithms have been proposed [8, 11]. Lefebvre and Feautrier [13] use interval coloring to minimize the number of data structures to rename in storage management for parallel programs. By continuing their graph coloring work [14], Li et al. [15] apply interval coloring to assign arrays in embedded programs to SPM.

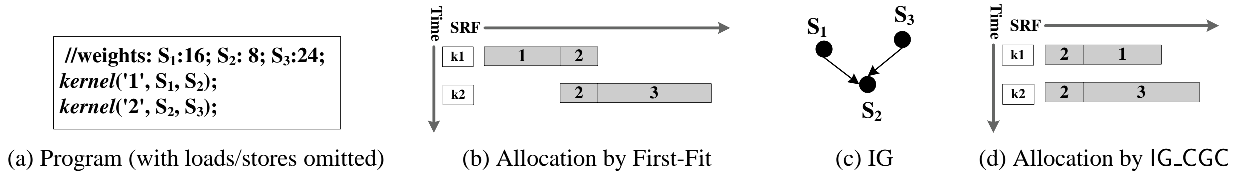
Govindarajan and Rengarajan [10] studied a compile-time buffer allocation problem for so-called regular stream flow graphs.

In [24], some improvements of [5] to LRFs (Local Register Files) allocation in stream processor are presented. The LRFs are register files near the ALU clusters. Unlike SRF, LRFs play a similar role as the register files in general-purpose processors.

7. Conclusion

This paper presents a new approach to optimizing utilization of the SRF for stream processors. The key insight is that the interference graphs (IGs) in media and scientific applications amenable to stream processing are comparability graphs or decomposable into well-structured comparability graphs (like trees). This has motivated the development of a new algorithm that is capable of finding optimal or near-optimal colorings efficiently, thereby outperforming First-Fit heuristics that are presently used in the literature.

This new approach also facilitates its integration with strip mining, an important optimization for stream processors for improving the performance of a stream application. There are at least two ways



(a) Program (with loads/stores omitted) (b) Allocation by First-Fit (c) IG (d) Allocation by IG_CGC

Figure 22: An example demonstrating the superiority of IG_CGC over First-Fit.

in which our algorithm can be combined with strip mining. One is to apply the algorithm to perform SRF allocation for a set of given strip sizes and return the best according to some performance-based cost model. Another is to combine our algorithm and strip mining to find a good strip size in a symbolical manner. All stream sizes are multiples of a strip size variable. So the best SRF allocation returned by our algorithm gives an upper bound on the strip size to be used. By combining this upper bound constraint with the execution time estimate of a stream application also expressed as a function of the strip size variable, the best strip size can be solved analytically or numerically using, say, MatLab.

Finally, our IG-based algorithm allows other pre-pass optimizations such as live range splitting and stream prefetching to be well integrated into the same compiler framework.

8. Acknowledgments

This work is supported in part by the NSF Grant of China (60621003) and a Chinese Council Scholarship and in part by the Australian Research Council Grant (DP0881330) and the UNSW Engineering-International Research Collaboration Grant (PS16380).

References

[1] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, 1994.

[2] G. J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–101. ACM Press, 1982.

[3] Fred C. Chow and John L. Hennessy. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.*, 12 (4):501–536, 1990.

[4] William J. Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, and Jung-Ho Ahn et al. Merrimac: Supercomputing with streams. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 35. IEEE Computer Society, 2003.

[5] Abhishek Das, William J. Dally, and Peter Mattson. Compiling for stream processing. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 33–42, New York, NY, USA, 2006. ACM.

[6] Janet Fabri. Automatic storage optimization. *SIGPLAN Not.*, 14(8): 83–91, 1979. ISSN 0362-1340.

[7] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, 1996.

[8] Jordan Gergov. Algorithms for compile-time memory optimization. In *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 907–908, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

[9] Martin Charles Golumbic. *Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics, Vol 57)*. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 2004.

[10] R. Govindarajan and S. Rengarajan. Buffer allocation in regular dataflow networks: An approach based on coloring circular-arc graphs. In *HIPC '96: Proceedings of the Third International Conference on High-Performance Computing (HiPC '96)*, page 419, 1996.

[11] H. A. Kierstead. A polynomial time approximation algorithm for dynamic storage allocation. *Discrete Math.*, 87(2-3):231–237, 1991.

[12] Francois Labonte, Peter Mattson, William Thies, Ian Buck, Christos Kozyrakis, and Mark Horowitz. The stream virtual machine. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 267–277, 2004.

[13] Vincent Lefebvre and Paul Feautrier. Automatic storage management for parallel programs. *Parallel Comput.*, 24(3-4):649–671, 1998.

[14] Lian Li, Lin Gao, and Jingling Xue. Memory coloring: A compiler approach for scratchpad memory management. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 329–338, 2005.

[15] Lian Li, Quan Hoang Nguyen, and Jingling Xue. Scratchpad allocation for data aggregates in superperfect graphs. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 207–216. ACM, 2007.

[16] Lian Li, Hui Feng, Quan Hoang Nguyen, Lin Gao, and Jingling Xue. Compiler-directed scratchpad memory management via graph coloring. *ACM Transactions on Architecture and Code Optimization*, 2009. To appear.

[17] John D. Owens. *Computer Graphics on a Stream Architecture*. PhD thesis, Stanford University, November 2002.

[18] John D. Owens, Ujval J. Kapasi, Peter Mattson, Brian Towles, Ben Serebrin, Scott Rixner, and William J. Dally. Media processing applications on the imagine stream processor. In *Proceedings of the IEEE International Conference on Computer Design*, pages 295–302, September 2002.

[19] Michael D. Smith, Norman Ramsey, and Glenn Holloway. A generalized algorithm for graph-coloring register allocation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 277–288. ACM, 2004.

[20] Michael Bedford Taylor and Jason Kim et al. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[21] W. Thies, M. Karczmarek, M. Gordon, D. Maze, J. Wong, H. Ho, M. Brown, and S. Amarasinghe. StreamIt: A compiler for streaming applications, 2001. MIT-LCS Technical Memo TM-622.

[22] Li Wang, Xuejun Yang, Jingling Xue, Yu Deng, Xiaobo Yan, Tao Tang, and Quan Hoang Nguyen. Optimizing scientific application loops on stream processors. In *LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 161–170. ACM, 2008.

[23] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM.

[24] Nan Wu, Mei Wen, Ju Ren, Yi He, and Chunyuan Zhang. Register allocation on stream processor with local register file. In *ACSAC '06: Proceedings of the 11th Asia-Pacific Computer Systems Architecture Conference*, pages 545–551, 2006.

[25] Xuejun Yang, Xiaobo Yan, Zuocheng Xing, Yu Deng, Jiang Jiang, and Ying Zhang. A 64-bit stream processor architecture for scientific applications. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 210–219. ACM, 2007.

[26] Xuejun Yang, Ying Zhang, Jingling Xue, Ian Rogers, Gen Li, and Guibin Wang. Exploiting loop-dependent stream reuse for stream processors. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 22–31, 2008.