A Performance Model and Efficiency-Based Assignment of Buffering Strategies for Automatic GPU Stencil Code Generation

Yue Hu\textsuperscript{1,2}, David M. Koppelman\textsuperscript{1,2}, and Steven R. Brandt\textsuperscript{1,3}

\textsuperscript{1}Center for Computation and Technology, Louisiana State University, Baton Rouge, LA, USA
\textsuperscript{2}Division of Electrical & Computer Engineering, Louisiana State University, Baton Rouge, LA, USA
\textsuperscript{3}Division of Computer Science, Louisiana State University, Baton Rouge, LA, USA
yhu14@lsu.edu, koppel@ece.lsu.edu, sbrandt@cct.lsu.edu

Abstract—Stencil computations form the basis for computer simulations across almost every field of science, such as computational fluid dynamics, data mining, and image processing. Their mostly regular data access patterns potentially enable them to take advantage of the high computation and data bandwidth of GPUs, but only if data buffering and other issues are handled properly.

Finding a good code generation presents a number of challenges, one of which is the best way to make use of memory. GPUs have three types of on-chip storage: registers, shared memory, and read-only cache. The choice of type of storage and how it’s used, a buffering strategy, for each stencil array (grid function, GF) not only requires a good understanding of its stencil pattern, but also the efficiency of each type of storage for the GF, to avoid squandering storage that would be more beneficial to another GF. Our code-generation framework supports five buffering strategies. For a stencil computation with \( N \) GFs, the total number of possible assignments is \( 5^N \). Large, complex stencil kernels may consist of dozens of GFs, resulting in significant search overhead.

In this work, we present an analytic performance model for stencil computations on GPUs, and study the behavior of read-only cache and L2 cache. Next, we propose an efficiency-based assignment algorithm, which operates by scoring a change in buffering strategy for a GF using a combination of (a) the predicted execution time and (b) on-chip storage usage. By using this scoring an assignment for \( N \) GFs and \( b \) strategy types can be determined in \((b-1)N(N+1)/2\) steps. Results show that the performance model has good accuracy and that the assignment strategy is highly efficient.

I. INTRODUCTION

Stencil computations’ mostly regular data access patterns potentially enable them to take advantage of the high computation and data bandwidth of GPUs, but only if data buffering and other issues are handled properly. There have been a number of stencil frameworks \cite{1, 2} designed to make it easier for programmers to write stencil codes and to generate good code, and in some cases that is done using auto-generation and autotuning of code. These frameworks evaluate proposed configurations using a performance model or a trial run. The latter method is time consuming, limiting configuration space exploration, especially when dynamic compilation is employed. But the configuration space that must be searched for multi-level tuning can slow even model-driven autotuning techniques.

Assignment of a buffering strategy is challenging due to the multiple storage options available on some accelerators, and due to the fact that all must be used for bandwidth-limited code in order to realize the device’s potential. With 5 possible strategies and 20 GFs, an exhaustive search would have to explore up to \( 5^{20} \approx 9.54 \times 10^{13} \) possibilities. The proposed efficiency-based assignment algorithm reduces this from \( 6^N \) to \((b-1)N(N+1)/2\) evaluations, where \( N \) is the number of GFs and \( b \) is the number of buffering strategies.

To cull the search space, we need to understand the implications of buffering strategy choice on the efficiency of GPU execution. The use of registers and shared memory on the one hand, can reduce off-chip memory traffic while, on the other hand, may limit thread-level parallelism and that will result in poor latency hiding. Moreover, it is important to have an accurate prediction of the cache behavior. For example, is it wise to have more than one GF access the read-only cache? We are unable to give a good answer unless we can accurately predict the cache behavior, which is very challenging.

In this work, we firstly present an analytic performance model for stencil computations on GPUs. In addition, we design an efficiency-based assignment algorithm, in which an efficiency function is defined to select a proposal. A proposal is a change in buffering strategy of one GF. The efficiency function balances the performance improvement of the change with its resource consumption.

The main contributions of this work are:

- We present an accurate performance model which considers reduced data traffic at off-chip memory through buffering, as well as the side effect of low occupancy. More importantly, the behavior of read-only cache and L2 cache is also studied.
- We design an efficiency-based assignment algorithm which can find suboptimal solution in \( O(N^2) \) time, instead of exponentially increasing design space with \( N \), where \( N \) is the number of GFs in a kernel.
- The algorithm is evaluated using a micro-benchmark...
GPUs. That is the relative size of registers, shared memory, al. [8] proposed a reconfigurable on-chip storage design for registers are the bottleneck for low concurrency. Gebhart et to offload the register pressure to shared memory when [6] found buffering too many GFs in shared memory would can noticeably increase performance as well. Yang et al. methods for stencil computations, and they found shared

B. Tradeoff of Buffering Strategies

Maruyama and Aoki [5] study a series of buffering strategies: buffering with shared memory and buffering with shared memory and registers, are considered. Experiments show good prediction accuracy, but read-only cache strategy is not studied. Moreover, buffering strategies of all GFs in a kernel have to be identical.

B. Tradeoff of Buffering Strategies

Maruyama and Aoki [5] study a series of buffering methods for stencil computations, and they found shared memory and registers are very helpful, and read-only cache can noticeably increase performance as well. Yang et al. [6] found buffering too many GFs in shared memory would limit concurrency and degrade performance as well. Hayes and Zhang [7] observed that on-chip storage of GPUs should be utilized in a balance way. For example, it is necessary to offload the register pressure to shared memory when registers are the bottleneck for low concurrency. Gebhart et al. [8] proposed a reconfigurable on-chip storage design for GPUs. That is the relative size of registers, shared memory, and cache can be reconfigured based on fact that different codes favor different on-chip storage assignments.

III. Preliminaries

The following is a brief description of the GPU used here, an NVIDIA Tesla K20xm. GPU execution starts with the launch of a kernel which executes as $g$ blocks each consisting of $b$ threads. Blocks are assigned to SMs, there are 14 in a K20xm. The block size, $b$, is limited by the resources available on an SM and the demands of the kernel. The resources include shared memory, 48 kiB, and registers, $2^{16}$. An SM also has a 48 kiB opt-in read-only (RO) cache. The small size and large latency of the L2 cache make it unsuitable for working set storage. (See Table I for values of additional device parameters.) The maximum block size is 1024 threads. The latency of arithmetic instructions is high (by CPU standards), on the order of 11 cycles, and so a large number of threads is needed to hide this latency and so avoid underutilization. That’s a concern for kernels with high per-thread shared memory or register demands, since they are limited to using fewer threads.

The issue rate varies by instruction, for double precision floating-point (FP) operations (including MADD operations) it is 64 instructions per cycle. At 732 MHz that works out to 1312 GFLOPS peak. Off-chip bandwidth is 250 GB/s, which works out to computation to communication ratio of 42 (operations per element).

IV. Background

In preparation for code generation Chemora[9] generates calculations from a user-provided PDE system description. PDE $\frac{dB}{dt} = \frac{\partial A}{\partial x}$ would be written as $D_{xx} B = D_{xx} A$. A calculation is essentially the body of a loop iterating over a grid space of grid points. At run time the Chemora code generator generates a CUDA kernel from these calculations. The code generator decides how to assign grid points to threads and how best to handle data reuse. The above equation translates into something like the loop below:

\[
\text{for } (i=imin \text{ to } imax, j=jmin \text{ to } jmax, k=kmin \text{ to } kmax) \\
\text{RHS}_B[i,j,k]=\text{A}[i,j-1,k]+2\text{A}[i,j,k]+\text{A}[i,j+1,k]/dxsq;
\]

$A$ and $\text{RHS}_B$ are 3D arrays called grid functions (GFs) and are indexed using loop indices and constant offsets. A stencil is the set of offsets for a particular GF. Reuse occurs in $A$ due to the offsets, most elements being accessed three times. The example above consists of just a single GF write, but Chemora calculations can consist of any number of GF offset loads and stores and can use any number of intermediate variables and loop-invariant values.

The experienced reader may have recognized that for the loop above a good configuration might assign one block to each value of $k$, assign one thread in a block to each value of $i$ (maximizing GPU memory request usage), and have threads iterate over $j$ (so that values loaded in one iteration can be used in the next). The configuration might be modified if the loop over $i$ is too short (multiple $k$’s per block) or if it is too long (split $i$ between blocks).

Chemora uses its performance model and efficiency-based algorithm to make such decisions. The goal is to choose a block configuration for the kernel and to choose a buffering strategy for each GF making use of any combination of available storage. A block configuration, denoted $C_{\text{Block}}$, is specified by four values: an iteration direction $e \in \{y, z\}$, the dimensions of the grid space plane covered by the block’s threads, $t_x$ and $t_f$, and the number of grid points processed per thread along the $e$-axis, $t_{ee}$, where $f$ denotes the axis orthogonal to $x$-axis and $e$-axis.

An assignment, $C_{\text{Assignment}} = (u, r, b, p, c)$, is a mapping of GFs to strategies. Each set $u, r, b, p,$ and $c,$ represents a different strategy. The grid functions are partitioned
A. Strategy u: Unbuffered

In the unbuffered strategy each offset load is realized by a global load instruction. Latency will be on the order of 189 cycles (L2 hit) to 300 cycles (L2 miss) and L2 misses consume off-chip bandwidth. Although the performance of the un-buffered strategy is relatively low, it uses no shared memory and the fewest registers, saving these resources for more performance-sensitive GFs.

B. Strategy v: Register

In the register strategy registers carry loaded values from one iteration to the next. It is applicable when the offset axis matches the iteration axis. This is the case in the example above if threads iterate in the j direction. Since there is less than one instruction per offset load, the strategy uses the least instruction bandwidth but consumes the most registers. For example, consider references to a GF at offsets $A[0, 1, 0]$ and $A[0, 0, 0]$ when iterating in the $+j$ direction. A load would be generated for reference $A[0, 1, 0]$ in one iteration and the loaded value would also be used for $A[0, 0, 0]$ in the next iteration, eliminating the need for any instruction to load the value for the second reference (in all but the first iteration). However the register holding the value cannot be used for other purposes. Overuse of registers lowers occupancy and even result in register spills, causing significant performance degradation.

C. Strategy b: Shared Only

In the shared only strategy, shared memory is used to hold GF elements. The amount of shared memory per GF is based on the volume of the grid points assigned to threads extended by the stencil offsets. The strategy also requires instructions to load shared memory and barriers to isolate shared memory updates. Because data buffered in shared memory can be accessed by all threads of a block, this buffering strategy can be applied to GFs with all types of stencil patterns. However, this strategy may require a large amount of shared memory, and so may result in a great drop in occupancy when it is applied.

D. Strategy p: Plus

The plus strategy, referred to elsewhere as 2.5D tiling, is a hybrid of the shared-only and register strategies. Like the register strategy, a global load is performed for the GF access having offset $(0, 0, 0) \in S$ with the largest $o_e$, and the value is kept in a register for use by offsets $(0, d, 0) \in S : d < o_e$, where $S$ is the stencil for the GF. The value is copied to shared memory for use by offsets $(o_e, 0, 0) \in S$. Global loads are used for all other offsets, such as $(1, 1, 1)$. For example, suppose that iteration is along the $y$ axis and the stencil for GF $A$ is $A[0, 1, 0]$, $A[0, 0, 0]$, $A[0, -1, 0]$, $A[1, 1, 0]$, $A[1, 0, 1]$. A global load will be used for $A[0, 1, 0]$ and the loaded value will be used for $A[0, 0, 0]$ and $A[0, -1, 0]$, shared memory will be used for $A[1, 1, 0]$ and a global load for $A[1, 1, 0]$. The amount of shared memory is the grid space occupied by the threads extended by the stencil only in the axis normal to iteration, a significant savings.

E. Strategy c: ROC (Read-Only Cache)

The ROC strategy uses read-only cache instructions for offset loads. This strategy consumes ROC space, but due to peculiarities of the NVIDIA Kepler instruction set (lack of immediate offsets), additional registers and arithmetic instructions are also used. As with any cache, performance depends upon several uncontrollable factors such as compiler instruction scheduling by the PTX compiler, warp execution ordering, and cache replacement.

V. PERFORMANCE MODEL

The efficiency-based assignment algorithm relies upon a performance model that estimates the execution time of a calculation for some block configuration and assignment running on some GPU. This model accounts not just for the data volume differences between the different buffering strategies, but also for any exposed latencies in the instructions needed to implement them.

A. Calculation Characterization

A calculation is characterized by a stencil for each input grid function, and three additional parameters: the number of FP operations, the number of GF stores, and the number of intermediate registers used. See Table IIb. These are mainly used to calculate on-chip memory usage. For example, $(t_e + s_e)(t_e + s_e)(t_f + s_f)$ elements of storage would be used for a GF under the buffered only with shared memory strategy.

Reuse factors, shown in the lower-half of the table, are computed from a GF's stencil once a block configuration and an assignment are chosen. These indicate the number of memory accesses performed per grid point for different axis alignments. For example, $r_e$ is the number of accesses
with offsets along the $e$-axis. These are used to estimate hit ratios in the read-only and L2 caches.

### B. GPU Characterization

Table IIIa shows hardware-related parameters and their values for NVIDIA GPU K20xms. Parameters $K_{\text{reg}}, K_{\text{smem}},$ and $K_{\text{roc}}$ are obtained from the product literature [10]. The other parameters are measured using micro-benchmarks. Details on micro-benchmarks will not be discussed in this paper due to limited space. Table IIIb lists parameters that are hard to obtain but important to performance modeling. These parameters are determined by hardware device, compiler, and stencil applications. For example, $\lambda_s$ is proportional to the number of registers allocated for data loaded from global memory. It could be as small as one when only one register is allocated. Therefore, for these parameters we estimate their values to the best of our knowledge.

### C. Execution Time Prediction

$$T_{\text{total}} = T_{\text{la}} + T_{\text{la}} + T_{\text{la}} + T_{\text{la}} + T_{\text{la}} + T_{\text{la}} + T_{\text{la}} + T_{\text{la}} + T_{\text{la}} + T_{\text{la}}$$

Table IV: Equations used for performance modeling

The execution time model considers three factors, kernel launch overhead, $T_{\text{launch}}$, data traffic, $T_{\text{data}},$ and exposed latency, $T_{\text{lat}}$. Exposed latency is the execution time when accounting for operation latency (including arithmetic and memory accesses) but ignoring memory congestion and issue bandwidth. It is possible in some situations to schedule instructions such that execution time is determined only by the larger of the two execution times. $T_{\text{total}} = T_{\text{launch}} + \max \{T_{\text{data}}, T_{\text{lat}}\}$. 

When loads are bunched together and near consuming instructions data bandwidth time cannot be hidden, in the worst case $T_{\text{total}} = T_{\text{launch}} + T_{\text{data}} + T_{\text{lat}}$. The more common

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_{\text{reg}}$</td>
<td>Number of registers used to store intermediate values</td>
<td></td>
</tr>
<tr>
<td>$n_{\text{gf}}$</td>
<td>Number of GF operations it takes to calculate a grid point</td>
<td></td>
</tr>
<tr>
<td>$n_{\text{fp}}$</td>
<td>Number of FP operations it takes to calculate a grid point</td>
<td></td>
</tr>
</tbody>
</table>

(a) Calculation parameters (except GF loads).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_k$</td>
<td>Magnitude of minimum $x$ offset.</td>
<td>$s_k = s_x + s_y$</td>
</tr>
<tr>
<td>$s_{xy}$</td>
<td>Difference between maximum and minimum $x$ offset.</td>
<td></td>
</tr>
<tr>
<td>$s_{xy}$</td>
<td>Difference between maximum and minimum $e$ offset.</td>
<td></td>
</tr>
<tr>
<td>$L_r$</td>
<td>Re-use factor for all offsets, $r = r_x + r_y$.</td>
<td></td>
</tr>
<tr>
<td>$r_o$</td>
<td>Number of offsets $(o_x, o_y, 0)$: $o_x \neq 0, o_y = 0$.</td>
<td></td>
</tr>
<tr>
<td>$r_o$</td>
<td>Number of offsets $(o_x, 0, o_y)$: $o_x \neq 0, o_y \neq 0$.</td>
<td></td>
</tr>
</tbody>
</table>

(b) GF load parameters derived from its offsets.

### Table II: Model input.

#### (a) Hardware parameters. *unit: elements (elts) / cycle / SM.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W_{\text{L},2}$</td>
<td>Measured L2 cache bandwidth</td>
<td>3.96 (3245 Gb/s)</td>
</tr>
<tr>
<td>$W_{\text{GM}}$</td>
<td>Measured global memory bandwidth</td>
<td>1.92 (1575 Gb/s)</td>
</tr>
<tr>
<td>$L_g$</td>
<td>Global memory latency (cycles)</td>
<td>305</td>
</tr>
<tr>
<td>$L_{L,2}$</td>
<td>L2 cache latency (cycles)</td>
<td>189</td>
</tr>
<tr>
<td>$L_{v,2}$</td>
<td>Shared memory latency (cycles)</td>
<td>32</td>
</tr>
<tr>
<td>$L_{r}$</td>
<td>Barrier latency (cycles)</td>
<td>483</td>
</tr>
<tr>
<td>$L_{r}$</td>
<td>Read-only cache latency (cycles)</td>
<td>108</td>
</tr>
<tr>
<td>$L_{r}$</td>
<td>FP operation latency (cycles)</td>
<td>10</td>
</tr>
<tr>
<td>$L_{c}$</td>
<td>Line size of read-only cache (elts)</td>
<td>16</td>
</tr>
<tr>
<td>$L_{v,2}$</td>
<td>Line size of L2 cache (elts)</td>
<td>6</td>
</tr>
<tr>
<td>$K_{\text{reg}}$</td>
<td>Register the capacity (elts/SM)</td>
<td>64</td>
</tr>
<tr>
<td>$K_{\text{smem}}$</td>
<td>Shared memory capacity (elts/SM)</td>
<td>6144 (48 kbit)</td>
</tr>
<tr>
<td>$K_{\text{roc}}$</td>
<td>Read-only cache capacity (elts/SM)</td>
<td>6144 (48 kbit)</td>
</tr>
</tbody>
</table>

#### (b) Software parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda$</td>
<td>Parallel factor for global memory access</td>
<td>3</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>Parallel factor for shared memory access</td>
<td>3</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>Parallel factor for barrier synchronization overhead</td>
<td>3</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>Parallel factor for read-only cache access</td>
<td>3</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>Parallel factor for FP operation</td>
<td>3</td>
</tr>
<tr>
<td>$L_o$</td>
<td>Overhead in cycle per iteration per thread block</td>
<td>100</td>
</tr>
</tbody>
</table>

Table III: Parameters used for modeling on GPU K20xm
where $\beta \in [0, 1]$ is the degree of non-interference of the two factors. An empirically determined value of .8 works well for the codes studied. Assuming the bottleneck of data traffic is either L2 cache or global memory, we have:

$$T_{data} = \max \left( \frac{D_{L2}^{total}}{W_{L2}}, \frac{D_{GM}^{total}}{W_{GM}} \right)$$

(2)

where $D_{L2}^{total}$ denotes the total volume of data loaded from and stored to the L2 cache, while $W_{L2}$ denotes the realistically achievable bandwidth. The definitions of $D_{total}^{GM}$ and $W_{GM}$ are similar. Note the unit of data volume is the number of data elements it takes to compute a grid point.

1) Traffic between the L2 Cache and SM: The total volume of data traffic between the L2 and SM cache consists of six parts as shown in Table [IVa]. These are shown using round-to-line operators $[a]^{(C)} = [a/LSC] \cdot LSC$ and $[a]^{(+C)} \equiv [a/LSC + \frac{1}{2}] \cdot LSC$. The first five parts, $D_{u}^{L2}$, $D_{L2}^{L2}$, $D_{b}^{L2}$, and $D_{p}^{L2}$ are for L2 cache reads from GFs with different buffering strategies. The last part $\alpha_{GM}$ is for L2 cache writes, where $\alpha$ is defined for the overhead of possible unaligned accesses.

For the unbuffered case $D_{u}^{L2}$, all reads are directed to the L2 cache, while the buffered cases have fewer reads. Take $D_{L2}^{L2}$, $D_{b}^{L2}$, and $D_{p}^{L2}$ for example. The reads are for loading data to on-chip memory in an iteration are independent of a GF’s data reuse $r_i$. Note for $D_{L2}^{b}$, only data along iteration direction E, and the one on XF plane that is orthogonal to E can be buffered as discussed in Section IV-D. Therefore irregular points (i.e. $r_p$), if they exist, are left unbuffered.

Equation $D_{L2}^{L2}$ needs care as it depends on the read-only cache’s behavior. It consists of two parts: $M_{roc}$ for cold misses and the others for capacity misses. $M_{roc}$ is equal to both the volume of data brought to the cache for the first time, and the minimum cache size to eliminate capacity misses in the equation for $R_{SND}^{RO}$ in Table [IVa]. Note here we assume data reuse on the read-only cache exists only along the X direction. The Supply and Demand Ratio of the read-only cache, $R_{SND}^{RO}$, is used to model the impact of capacity misses which exist when $R_{SND}^{RO}$ is smaller than one.

2) Volume of traffic at global memory: Similarly, Table [IVb] lists the equations for computing the total volume of data traffic at global memory. For $D_{GM}^{GM}$, $D_{b}^{GM}$, and $D_{p}^{GM}$, the data has no reuse so the volume of global memory is equal to the volume of L2 cache.

Equation $D_{u}^{GM}$ and $D_{c}^{GM}$ needs care as the L2 cache is shown to be helpful in reducing traffic to global memory for data with reuse. Similarly $R_{SND}^{L2}$, the Supply and Demand Ratio of the L2 cache, is used to model the impact of capacity misses. $M_{u}^{L2}$, $M_{L2}^{L2}$, $M_{b}^{L2}$, and $M_{p}^{L2}$ denote the minimum L2 cache size that is required to avoid capacity misses by GFs buffered in different strategies. $N_{SM}$ denotes threads running on all SMs are sharing the same L2 cache.

3) Analysis of exposed latency: To quantify the impact of thread-level parallelism on execution time, we propose a model in Table [IVc] where exposed latency is high when thread-level parallelism (i.e. smaller $t_s$, $t_d$) is low.

In addition to parallelism, various operation latencies have impact on $T_{lat}$. The latencies we modeled include global memory accesses, L2 cache accesses, shared memory accesses, barrier operations, read-only cache accesses, floating point operations, and a thread block’s time overhead $L_m$ per iteration. The parameter $\lambda_g$ represents the number of global memory load and store instructions that can be issued in parallel. Correspondingly, $N_g L_g$ denotes the exposed latency from global memory accesses. The definitions for the other $N$’s, $L$’s, and $\lambda$’s are similar.

D. On-Chip Memory Usage Affects Parallelism

Table [IVd] shows how usage of registers and shared memory affects parallelism. We can use the two inequalities in Table [IVd] to solve for the variables $t_i$, given a value for $t_x$. This gives us the block configuration $C_{Block}$.

VI. EFFICIENCY-BASED ASSIGNMENT

Algorithm 1 Efficiency-based on-chip memory allocation

1. Gather metadata of $n$ GFs
2. Initialize $C_{Assignment}: u = \{1, 2, ..., n\} \text{ and } r = b = p = c = \emptyset$
3. $next = true$
4. while $next \neq false$ do

5. $m_{rbp} = m_c = NULL$
6. for $i \in u$ & $(r_i > 0)$ do

7. for every $(r, b, p, c)$ as target set $t$ do

8. //Propose a move $m$; move $i$ from $u$ to $t$

9. if $m$ is invalid then

10. continue

11. end if

12. if $t$ is not $c$ then

13. $m_{rbp} = m$ if $\mathcal{E}(m) > \mathcal{E}(m_{rbp})$

14. else

15. $m_c = m$ if $\mathcal{E}(m) > \mathcal{E}(m_c)$

16. end if

17. end for

18. end for

19. if $min(\Delta T(m_{rbp}), \Delta T(m_c)) > 0$ then

20. $next = false$

21. else

22. if $\Delta T(m_{rbp}) < \Delta T(m_c)$ then

23. Accept $m_{rbp}$

24. else

25. Accept $m_c$

26. end if

27. end if

28. end while

Let $C = \{u, r, b, p, c\}$ be some incomplete assignment and let $m$ denote a proposed reassignment of some $gf \in u$. 
At the heart of the efficiency-based assignment algorithm is an efficiency function, \( E(C, m) \in R \), which gives the efficiency (desirability) of assignment \( m \) given an incomplete assignment \( C \). The efficiency function is evaluated for all moves from an incomplete assignment, the best of which is used to generate a new (possibly) incomplete assignment.

The efficiency function has been designed to avoid back-tracking, so that an entire assignment can be computed in \((b-1)N(N+1)/2\) steps where \( b \) and \( N \) denote the number of buffering strategies and GFs separately, as compared to \( b^N \) steps for an exhaustive search.

Algorithm 1 shows how on-chip memory is allocated to improve performance step-by-step in a memory-efficient way. At the beginning, all GFs are un-buffered. Line6-7: shows all possible moves \( m \) to buffer an un-buffered GF in one of four different ways. Note that GFs must have data reuse (i.e., \( r_i > 0 \)) to be buffered. Note9-11: a proposed move \( m \) could be invalid when a buffering strategy does not apply to a GF due to its stencil pattern.

The performance improvement of a move \( m \) is defined by \( \Delta T \), which denotes the change in execution time. We define the efficiency function \( E(m) \), Eq. 2, to identify steps that reduce the execution time, but that at the same time to penalize changes that take too much memory and reward changes that reduce memory. Buffering a GF may reduce memory since it may reduce the number of threads while, for example, the number of used registers per thread is kept constant. The variables \( is_{reg\_limited}, is_{smem\_limited}, \) and \( is_{roc\_limited} \) are either 1 or 0 and denote whether on-chip memory (in the form of registers, shared memory, or read-only cache) is limited or not. For example, \( is_{reg\_limited} \) is 0 if the number of registers is large enough to buffer all unbuffered GFs at the same occupancy.

\[
E(m) = -\triangle T(m) \left( \exp(-is_{reg\_limited} \triangle M_{reg}/K_{reg}) + \exp(-is_{smem\_limited} \triangle M_{smem}/K_{smem}) + \exp(-is_{roc\_limited} \triangle M_{roc}/K_{roc}) \right)
\]

(3)

L12-16: \( m_{rbp} \) is the most efficient move for strategies that buffer in registers, shared memory, and shared memory and registers, while \( m_c \) is the most efficient strategy for buffering in read-only cache. The reason we have two types of most efficient move is \( M_{roc} \) denotes the minimum cache size to avoid a capacity miss. Unlike \( M_{reg} \leq K_{reg} \) and \( M_{smem} \leq K_{smem} \), \( M_{roc} \) could be greater than \( K_{roc} \).

L19-21: the algorithm terminates when the execution time cannot be reduced. L22-26: accept \( m_{rbp} \) or \( m_c \) depending on which one results in greater execution time drop.

VII. EXPERIMENTAL METHODOLOGY

The efficiency-based assignment algorithm was evaluated in a set of runs on a system consisting of an Intel Xeon E5-2670 CPU and an NVIDIA Tesla K20xm GPU. The GPU code was prepared with CUDA version 6.0.1. In addition, a micro-benchmark and a number of real-world applications are used for evaluation. Table V lists their characteristics.

The micro-benchmark we designed is a 3D stencil computation with one output GF \( B \) and ten input GFs \( A_n \), \( n \in \{0, 1, \ldots, 9\} \), as shown in Eq. 1, \( c_n[m] \) with \( m \in \{0, 1, \ldots, 8\} \) are constants with different values. The stencil pattern of GF \( A_n \) is the same as that of a GF in a real-world application Fluam. We designed such micro-benchmark for three reasons. First, we expect it to be representative of a class of real applications. Second, all input GFs have the same stencil pattern making it easy to demonstrate the performance model. Third, the benchmark has enough GFs so that an efficiency-based assignment algorithm is needed to search the design space.

\[
B[i,j,k] = \sum_{n=0}^{9} (c_n[0] \times A_n[n, i, j, k] + c_n[1] \times A_n[i - 1, j, k] + c_n[2] \times A_n[i + 1, j, k] + c_n[3] \times A_n[i, j - 1, k] + c_n[4] \times A_n[i, j + 1, k] + c_n[5] \times A_n[i, j, k - 1] + c_n[6] \times A_n[i, j, k + 1] + c_n[7] \times A_n[i - 1, j + 1, k] + c_n[8] \times A_n[i - 1, j, k + 1])
\]

(4)

Fluam is a GPU application (in CUDA C) developed for fluctuating hydrodynamics with the finite volume method. It provides fluid solvers for the compressible and incompressible Navier-Stokes equations using an Eulerian-Lagrangian approach. It offers 18 simulation schemes such as third order Runge-Kutta (RK3), thermo-statistics, and particle-wall. In this work the default RK3 scheme, which consists of kernel RK3_1 and RK3_2, is being evaluated.

SCALE-LES is a large-eddy simulation model developed for weather and climate study. Wahib and Maruyama manually implemented the model using CUDA Fortran and OpenACC. The stencil kernels we evaluated (i.e. Scale_1 and Scale_2) are two of the most time-consuming ones.

McLachlan code is a part of Einstein Toolkit, and BSSN formulations are a set of coupled nonlinear partial differential equations with 25 GFs. We study Advect and Dalphda_1 out of BSSN’s 11 calculations because of their representative stencil patterns and high data reuse.
VIII. RESULTS

We first demonstrate the accuracy of the proposed performance model, we then demonstrate the effectiveness of the efficiency-based algorithm running stencil applications.

A. Validation of the Performance Model

Figure 1a shows data transaction at L2 cache and global memory with different number of GFs being buffered and in three different ways. It can be seen that the predicted and measured number of data transactions is close for all buffering methods, though the predicted values are slightly smaller in general. This discrepancy is probably because our model assume GPU hardware and the compiler can avoid unnecessary data transaction in a very efficient way. Another interesting observation is that the L2 cache efficiently reduces the number of data transactions reaching global memory.

Figure 1b shows a comparison between the predicted and measured execution time, and the corresponding thread-level parallelism. Execution time is shown in cycle per grid point per SM. We can see that the predicted execution time and the measured ones match well except when thread-level parallelism changes. Take buffering strategy using shared memory for example, noticeable gap exists for 2, 3, 4, and 5 GFs being buffered. This is because as thread-level parallelism decreases, operation latency (including arithmetic and memory accesses) will be worse hidden. However, it is hard to accurately estimate the degree of latency hidden, compared with the data transaction.

Figure 1: Evaluating performance model using a micro-benchmark.

B. Effectiveness of the Efficiency-Based Scheme

The effectiveness of the efficiency-based assignment algorithm was evaluated by running a set of benchmarks on systems using the efficiency-based scheme, Chemora_ByEff, and one that chooses based only on performance Chemora_ByPerf. In addition, configurations were run in which assignment is based on a large search and manual tuning, Chemora_Best, and hand-tuned versions of the code, Original_Code. The run times are plotted in Figure 2 normalized to Chemora_Best.

As we can see, Chemora_ByEff is better than Chemora_ByPerf on two runs, and equal on the others. Chemora_ByEff has an edge where GFs’ data access patterns within a kernel vary, where all GF have identical patterns the two schemes are equal.

For comparison, the performance of the original CUDA
code for Fluam [11] and Scale-les [12] running on the test system is shown. Chemora outperforms the original code for a number of reasons, such as dynamic compilation and efficient memory access and buffering code. These aspects of Chemora will be reported elsewhere.

Figure 3 shows a case study of the proposed efficiency-based assignment scheme. Initially, all GFs a0-a9 are un-buffered. At each step, a GF will be buffered in one of three ways, depends on their performance improvement or buffering efficiency. Scheme ByPerf buffers two GFs using shared memory in the first steps, results in fast drop of occupancy and no improvement to buffer the 6th GF. In contrast, scheme ByEff has 7 GFs being buffered at the end, resulting in minimum predicted execution time.

**IX. Conclusions**

In this work, we first propose a performance model which supports five buffering strategies. Next, we present an efficiency-based assignment algorithm which can effectively reduce searching time from $O(5^N)$ to $O(N^2)$ where $N$ denotes the number of GFs in a kernel. Experiments show good prediction accuracy. In addition, the performance achieved by the algorithm is 95.4%, on average, that of the manual tuned implementation. Future work could be evaluation: a) on other GPUs; b) with kernel fusion and fission.

**X. Acknowledgments**

We thank Frank Löffler, Peter Diener, Roland Haas, Ian Hinder, and Jian Tao as the main authors of Chemora. Cactus and Chemora are currently supported by NSF awards 0905046 PetaCactus: Unraveling the supernova — Gamma-ray burst mystery; 0941653 Enabling Science at the Petascale: From Binary Systems and Stellar Core Collapse To Gamma-Ray Bursts; 1212401, 1212426, 1212433, 1212460 The Einstein Toolkit — An open source general relativistic multi-physics infrastructure for relativistic astrophysics; 1265449, 1265434, 1265451 Using PDE descriptions to generate code precisely tailored to energy-constrained systems including large GPU accelerated clusters. We would also like to thank NVIDIA Corporation for providing some of the GPUs used for this study.

Our research was made possible by access to HPC resources at Louisiana State University, in particular the Shelob cluster using the allocations hpc_numrel and hpc_hyrel.

**REFERENCES**


