Scalable Kernel Fusion for Memory-Bound GPU Applications

Mohamed Wahib, mohamed.attia@riken.jp, Naoya Maruyama, nmaruyama@riken.jp
Advanced Institute for Computational Science, RIKEN , Kobe, Japan
TEL: +81-78-940-5555 FAX: +81-78-304-4955 http://www.aics.riken.jp/en/

K computer
Computer simulations create the future

I- Problem Statement

Motivation

GPU implementations of HPC applications utilizing finite difference methods can include dozens of kernels that are memory-bound. Improving the performance of those kernels by reducing data access to off-chip memory is achievable by fusing kernels that share data arrays into new larger kernels and using on-chip cache to hold the data reused by instructions originally from different kernels. The identification of kernels to be fused for improving the performance most effectively while adhering to data dependencies and kernels order-of-execution is a nontrivial problem, especially for applications with a large number of kernels.

Target

A scalable heuristic to search the space of possible kernel fusions to identify optimal kernel fusions for large problem sizes.

Our approach to achieve target

- Define kernel fusion as a variation of Bin Packing combinatorial problem.
- Use a stochastic search method.
- Use a light-weight codeless performance bound projection to guide the search heuristic.
- Evaluate using a mini-app test suite and a real-world weather application.

II- Kernel Fusion

The optimization by kernel fusion is summarized to the following components. First, gathering metadata about the input of original kernels: kernels' performances and characteristics, the data dependency graph and kernel order-of-execution graph. Second, define a performance projection model that depends on the metadata from step one and architecture features. The performance model should be lightweight, i.e. requiring no intermediate step to generate some form of code representation. Third, introduce a search algorithm that searches the space of possible fusions. Finally, use the result of the search algorithm to generate the new fused kernels. Fig. 1 shows a dependency graph for RK routine in SCALE-LES weather application[1], RK routine is calculated to have almost 50% redundancy, i.e. the number of loads and stores from GPU device memory would be halved due to using data cached in on-chip shared memory. Fig. 2 shows an example extracted from RK code that illustrates how the original kernels are transformed to fused kernels by rearranging the kernels, fusing them, then caching the shared arrays.

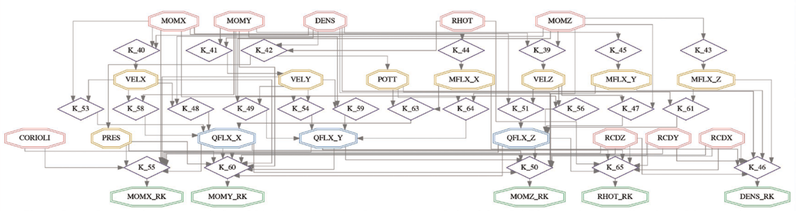


Fig 1: Data Dependency graph for GPU Kernel in the RK routine of the SCALE-LES weather application. Diamonds are Kernels invoked in the order of the numbered label, input-only arrays red octagon, output-only arrays green octagon, expandable input-output arrays blue and strict input-output arrays yellow octagon

```
< IRRELEVANT CODE >                                   Kernel 1
...
VELZ_XY(k,i,j) = 2.0_RP * MOMZ(k,i,j) / ( DENS(k+1,i,j)+DENS(k,i+1,j) )

< IRRELEVANT CODE >                                   Kernel 2
...
mflx_hi(k,i,j,ZDIR) = MOMZ(k,i,j) + num_diff(k,i,j,I_DENS,ZDIR) * rdtrk

< IRRELEVANT CODE >                                   Kernel 3
...
DENS_RK(k,i,j) = DENS0(k,i,j) + dtrk &
  * ( - ( ( mflx_hi(k,i,j,ZDIR)-mflx_hi(k-1,i, j, ZDIR) ) * RCDZ(k) &
  + ( mflx_hi(k,i,j,XDIR)-mflx_hi(k ,i-1,j, XDIR) ) * RCDX(i) &
  + ( mflx_hi(k,i,j,YDIR)-mflx_hi(k ,i, j-1,YDIR) ) * RCDY(j) ) )

< IRRELEVANT CODE >                                   Kernel 4
...
VELZ_XY(k,i,j) = 2.0_RP * MOMZ(k,i,j) / ( DENS(k+1,i,j)+DENS(k,i+1,j))
VELY_ZX(k,i,j) = 2.0_RP * MOMY(k,i,j) / ( DENS(k,i,j+1)+DENS(k,i,j))

< IRRELEVANT CODE >                                   Kernel 5
...
qflx_hi(k,i,j,ZDIR) = 0.25E0_RP * ( VELZ(k,i,j)+VELZ(k-1,i,j) ) &
          * ( FACT_N * ( MOMZ(k ,i,j)+MOMZ(k-1,i,j) ) &
          + FACT_F * ( MOMZ(k+1,i,j)+MOMZ(k-2,i,j) ) ) &
          + num_diff(k,i,j,I_MOMZ,ZDIR) * rdtrk
```

Original Kernels. Invoked by host in order

```
< IRRELEVANT CODE >                                   Kernel 1_4
...
s_DENS(k,i,j) = DENS(k,i,j)        ! Copy from GMEM to SMEM
call synchreads()                  ! Barrier
VELZ_XY(k,i,j) = 2.0_RP*MOMZ(k,i,j)/(DENS(k+1,i,j)+s_DENS(k,i+1,j))
VELZ_XY(k,i,j) = 2.0_RP*MOMZ(k,i,j)/(DENS(k+1,i,j)+s_DENS(k,i+1,j))
VELY_ZX(k,i,j) = 2.0_RP*MOMY(k,i,j)/(DENS(k,i,j+1)+s_DENS(k,i,j))

< IRRELEVANT CODE >                                   Kernel 2_3_5
...
real(DP), shared::     s_mflx_hi(K,I,J,ZDIR) ! Declare SMEM
s_MOMZ(k,i,j)          = MOMZ(k,i,j)         ! Copy from GMEM to SMEM
call synchreads()
s_mflx_hi(k,i,j,ZDIR) = s_MOMZ(k,i,j) + num_diff(k,i,j,I_DENS,ZDIR) &
                      * rdtrk
! Compute value for z direction of Halo layer
if(AbsolutethreadIDX == z_Boundary) then
  s_mflx_hi(k+1,i,j,ZDIR) = s_MOMZ(k+1,i,j) + &
                    num_diff(k+1,i,j,I_DENS,ZDIR) * rdtrk
Endif
call synchreads()                  ! Barrier
DENS_RK(k,i,j) = DENS0(k,i,j) + dtrk &
* ( - ( ( s_mflx_hi(k,i,j,ZDIR)-s_mflx_hi(k-1,i, j, ZDIR) ) * RCDZ(k) &
+ ( mflx_hi(k,i,j,XDIR)-mflx_hi(k ,i-1,j, XDIR) ) * RCDX(i) &
+ ( mflx_hi(k,i,j,YDIR)-mflx_hi(k ,i, j-1,YDIR) ) * RCDY(j) ) )

qflx_hi(k,i,j,ZDIR) = 0.25E0_RP * ( VELZ(k,i,j)+VELZ(k-1,i,j) ) &
* ( FACT_N * ( s_MOMZ(k ,i,j)+s_MOMZ(k-1,i,j) ) &
+ FACT_F * (s_MOMZ(k+1,i,j)+s_MOMZ(k-2,i,j) ) ) &
+ num_diff(k,i,j,I_MOMZ,ZDIR) * rdtrk
mflx_hi(k,i,j,ZDIR) = s_mflx_hi(k,i,j,ZDIR) ! Copy from SMEM to GMEM
```

Fused Kernels. Invoked by host in order

Fig 2: Kernel fusion example from SCALE-LES. Five kernels are transformed into two larger kernels. Colored text highlights the data reused by caching in SMEM (Ex. array DENS is shared in kernels 1 and 4)

III- Kernel Fusion as an Optimization Problem

Kernel fusion is represented as a variation of the classical Bin Packing problem. In the bin packing problem, objects of different volumes must be packed into a finite number of bins or containers each of a pre-defined volume in a way that minimizes the number of bins used. The problem in hand is similar; minimize the total runtime of fused kernels by fusing kernels of different performances into kernels that have a total runtime less than the original runtime.

Input Original kernels $K, |K| = n$ & Data arrays $D, |D| = l$

Output Fused kernels $F, |F| = m$

Minimize

$$\sum_{j=1}^{m} T_j \quad (1)$$

Subject to

$$\sum_{i \in F_k} P_i > T_k \quad (1.1)$$

$$\sum_{j=1}^{m} x_{ij} = 1, \forall i \in \{1, ..., n\} \quad (1.2)$$

$$x_{qr} = 1, \forall q \in \overline{k^{ab}}, x_{ar} = 1, x_{br} = 1 \quad (1.3)$$

$$x_{ij} \in \{0, 1\}, \forall i \in \{1, ..., n\} \forall j \in \{1, ..., m\} \quad (1.4)$$

$$DegKin(k_x, k_y) > 0, x \neq y, \forall k_x \forall k_y \in F_k, \forall F_k \in F \quad (1.5)$$

$$MEM(F_j) \leq Capacity, \forall j \in \{1, ..., m\} \quad (1.6)$$

$$REG(F_j) \leq R_{Max}, \forall j \in \{1, ..., m\} \quad (1.7)$$

Where

$x_{ij} = 1$ if kernel k_i is fused to kernel F_j

$\overline{k^{ab}} =$ the set of kernels in the path from k_a to k_b in the dependency graph of array d_l

$DegKin(a, b)$ returns *Degree of kinship* between kernels a, b

$MEM(k)$ returns on-chip memory used by kernel k

$Capacity$ is the device on-chip memory capacity

$REG(k)$ returns registers per thread used by kernel k

R_{Max} is the maximum registers allowed per thread

IV- Performance Projection Model

To have a scalable kernel fusion, a codeless projection of the bound of performance for fused kernels at each generation in the search process is used. The performance model is based on the model for projecting bounds on performance in [2].

Parameter	Description	Value
R_{SMX}	Register resource per SMX	64KB
R_{Max}	Max. No. registers per thread	255
Sh_{SMX}	SMEM (B) per SMX	48KB
$GMEM_{BW}$	Peak GMEM bandwidth (STREAM)	202 GB/S
R_{Max}	Max. No registers per thread	255
$P_{Theoritical}$	Theoritical peak performance (DP)	1.31 TFLOPS
SMX	No. SMX	14

Tab 1: Metadata of an original kernel

Parameter	Description
$Blocks_{SMX}$	No. active blocks per SMX
T_B	No. active threads per block
Thr	No. threads per block
B	No. blocks
R_T	No. registers allocated per thread
T	No. registers allocated for indices and addresses
Fl	Runtime of the kernel in seconds
$OrdShr(x)$	No. of threads in a block accessing an element of x
$Flop(x)$	No. of floating-point operations in relation to array x
$ShrLst$	List of arrays in kernel touched by other kernels
Hal	Size (B) of the halo region of a thread block

Tab 2: Kepler K20X architecture features

The model implicitly deduces the practical performance bound depending on CUDA runtime's ability of hiding the latency in a specific kernel via register and SMEM blocking factors:

$$Blocks_{SMX} * Thr * R_T \leq R_{SMX} \quad (2)$$

For *RegFac* the register reuse factor for stencil computation, $H_{TH} = Hal/Thr$ is the registers required for halo layers(s) divided evenly on threads per block. The value *c* is equal to one if halo layer(s) was used in the fused kernel to maintain coherence, otherwise equal to zero:

$$RegFac * max(OrdShr(x)) + c * H_{TH} + 1 < R_T \leq R_{Max} \quad (3)$$

For $R_{fetch} = T_B * (1+c*H_{TH})$ being the extra registers to fetch data from GMEM to SMEM, to accommodate the register capacity constrain in equation 1.7:

$$R_{fetch} + RegFac * max(OrdShr(x)) + c * H_{TH} + R_{adr} + 1 < R_T \leq R_{Max} \quad (4)$$

To accommodate the SMEM capacity constraint in equation 1.6:

$$(1 + c * H_{TH}) * T_B * Blocks_{SMX} * Size(ShrLst) \leq Sh_{SMX} \quad (5)$$

The SMEM blocking factor B_{Sh} is a direct function of the number of active threads, halo size and the number of shared arrays:

$$B_{Sh} = T_B * Blocks_{SMX} / ((1 + c * H_{TH}) * Size(ShrLst)) \quad (6)$$

For $B_{eff} = B_{Sh} * SMX/(Thr * B)$, the performance bounded by GMEM BW is estimated to be:

$$P_{MemBound} = \frac{B_{eff} * GMEM_{BW}}{8} \text{ GFLOPS} \quad (7)$$

If N is the set of original kernels fused to the current fused kernel, M is the set of original kernels that were fused into the current kernel and have to compute a halo layer(s) and S is the set of data arrays that required a halo layer(s). The projected runtime (Sec) of the fused kernel would be:

$$T_{pro} = \left(\sum^{N} Fl + \frac{\sum^{M} Flop(x) * Hal}{B * Thr}\right) * 10^{-9} \Big/ P_{MemBound}, \; x \in S \quad (8)$$

V- Performance Improvement by Kernel Fusion

Algorithm 1 shows the steps for improving performance by kernel fusion. Step 4 is the projection model used as an objective function for the search heuristic. Steps 3 to 8 show the stochasitc search method used. Step 9, manual fusion, can be automated.

Algorithm 1 Improving Performance by Kernel Fusion

1: Create metadata of original kernels $K_i, i = 1, , n$
2: For all shared arrays $D_o, o = 1, , l$, create dependency graph and order-of-execution graph
3: $G_0 \leftarrow$ Generate M individuals (initial population) each having set F of fused kernels
4: For all M, for fused kernel $f \in F$:
  i: Aggregate metadata of original kernels of f
  ii: Project the runtime of f from projected performance bound
5: $G_t^{Se} \leftarrow$ select $N \leq M$ individuals from G_{t-1} according to selection method
6: $p_t(x) = p_t(x \mid G_{t-1}^{Se}) \leftarrow$ Estimate the probability distribution of an individual being among N
7: $G_t \leftarrow$ Sample M individuals (new population) from $p_t(x)$
8: If termination criteria not met, go to step **4**
9: Use value of best individual to manually fuse the kernels

VI- Results

Test Suite from Mini-App: CloverLeaf

CloverLeaf [3] is a Lagrangian-Eulerian hydrodynamics mini-app that solves the compressible Euler equations on a 2D Cartesian grid. The test suite includes different-sized benchmarks based on CloverLeaf's main memory-bound six kernels.

#(K,D)	%	#(K,D)	%	#(K,D)	%	#(K,D)	%	#(K,D)	%
(10,20)	100	(20,20)	100	*(30,20)*	100	(40,20)	100	(40,20)	98.2
(10,40)	100	(20,40)	100	*(30,40)*	98.5	(40,40)	98.7	(50,40)	99.1
(10,60)	100	(20,60)	100	*(30,60)*	100	(40,60)	97.3	(50,60)	99.8

Tab 3: Percentage of best answers found in benchmarks With (# Kernels,# Arrays)

#(K,D)	T	#(K,D)	T	#(K,D)	T	#(K,D)	T
(80,140)	2.13	(80,180)	2.63	(100,140)	2.21	(100,180)	3.02
(80,160)	2.27	*(80,200)*	2.86	(100,160)	2.53	(100,200)	3.24

Tab 4:Elapsed time (hours) for best found solution in benchmarks with (# Kernels,# Arrays)

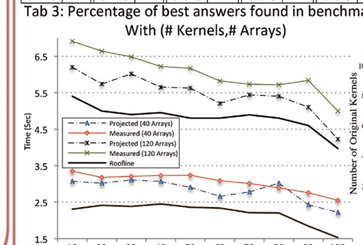
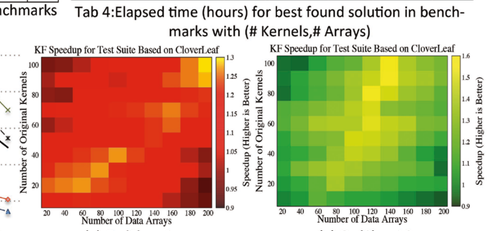


Fig 3: Time projected by performance model VS runtime



(a) OrdShr =4 (a) OrdShr =12

Fig 4: Test suite speedups. Note speedup scale (1.3x VS 1.6x)

Real-world Application: SCALE-LES

SCALE-LES is a next generation weather model designed to confirm the effects of 3D grid spacing, domain size, aspect ratio and precision on the model accuracy. SCALE-LES [1] includes over a hundred kernel of which most are memory-bound.

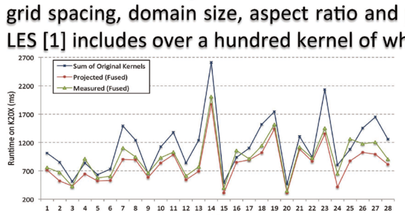


Fig 5: Runtime (ms) on K20X for the measured, projected and *original sum* of fused kernels in SCALE-LES. 111 Original kernels transformed to 30 (unchanged) + 81 (fused to 24 kernels)

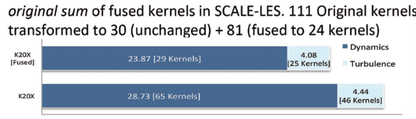


Fig 6: SCALE-LES main modules runtime (sec) for K20X

Fusion Efficiency (FE) compares the reduction in runtime of Kernel F to reduction in number of GMEM memory operations due to fusion:

$$FE = \frac{LD_F + ST_F}{\sum_{i=1}^{n} LD_i + ST_i} \Big/ \frac{T(F)}{\sum_{i=1}^{n} T(K_i)}$$

Attribute	Kernel_I4	Kernel_I9	Kernel_23	Kernel_27
Fusion Efficiency	0.89	0.96	0.92	0.91
Original Kernels	6	5	6	4
Kernel Anchor	5	4	3	3
Chain of Sharing	2	1	N/A	N/A
Kinship	4	2	1	1
Avg. Ord. Sharing	7.2	3.1	4.2	4.3

Tab 5: Features of the kernels with highest runtime

RIKEN Advanced Institute for Computational Science (AICS)

[1] M. Wahib and N. Maruyama. Highly Optimized Full GPU-Acceleration of Non-hydrostatic Weather Model SCALE-LES. IEEE CLUSTER'13.
[2] J. Lai and A. Seznec. Performance Upper Bound Analysis and Optimization of SGEMM on Fermi and Kepler GPUs, ACM CGO'13.
[3] J. A. Herdman, W. P. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis. Accelerating Hydrocodes with OpenACC, OpeCL and CUDA. SCC '12, pages 465–471, 2012.