Mapping Brook Stream Stencils to Imagine Architecture

EE482C Project, Spring 2002

Jacob Chang, Nathan Hill, Jae-Wook Lee, Alex Solomatnikov

Summary

In this project we implemented general framework for mapping Brook stencils to KernelC code for 1D and 2D streams. Access pattern to stream elements described by Brook stencil is used as an input to a Perl script, which generates KernelC code. This code contains a kernel that reads stream elements, communicates them between clusters according to access pattern, and does all state bookkeeping. To complete the KernelC program user should add computation code specified in the Brook kernel. To analyze performance we added an option to the script that generates convolution code inside the kernel loop. We generated convolution kernels for various stencil sizes, tested their correctness, and analyzed their performance.

For associative computation, i.e. which can be broken into independent pieces, we developed another script that generates optimized code for this case. We showed that with this optimization the performance of our example convolution kernel is almost always compute limited, i.e. limited by the multiplier.

We have found several shortcomings of current KernelC compiler: inability to restructure computation for performance optimization; “lazy” instruction scheduling, which increases register file pressure. Also, in many cases because of inefficient instruction scheduling the usage of scratchpad does not give any advantage.

1. Approach

In Brook program all parallelizable computation should be described in terms of kernels, i.e. stream functions, which are allowed to have only limited access to stream elements and no access to global data. The access pattern to stream elements is specified by stencils. For example, the following Brook statement:

```
streamStencil(t, s, 1, STREAM_BOUNDS_CLAMP, -3, 0)
```

specifies a stencil `t` such that each element of `t` consists of 4 elements of stream `s` with relative positions from -3 to 0. Kernel, which input is a stencil `t`, will operate on 4 elements of stream `s` at a time.

In Imagine 8 clusters read 8 stream elements in parallel as shown in Fig. 1a. After read stream elements should be communicated between clusters such that all four elements belonging to the stencil are "assembled" in one cluster as shown in Fig. 1b. Note, that 5th, 6th, and 7th elements are read in previous iteration and stored inside the cluster. Additional select operations are required to select between the elements read in the current iteration and elements from previous iteration(s). Once all elements of the stencil are "assembled" inside one cluster, the cluster can perform any computation on stencil elements specified in the Brook kernel. This step is beyond the scope of our project

Fig.1. Kernel with stencil [-3, 0]: a) after stream read; b) after intercluster communication

In order to map Brook stencils we developed KernelC for several 1D stencils with different sizes ($-a \le x \ge b$, where $a \ge 0$ and $b \ge 0$), and several 2D stencils ($-a \le x \ge b$, where $a \ge 0$ and $b \ge 0$) ($-a \le x \ge b$, $-c \le y \ge d$ where $a \ge 0$, $b \ge 0$, $c \ge 0$, and $d \ge 0$). Based on these examples we wrote Perl scripts that can generate KernelC code for arbitrary stencil size. For testing purposes our scripts have "-conv" option. This option generates additional code inside the stream loop, which convolves stencil elements with constant coefficients and writes the result to the output stream. Next section describes 1D case implementation and results.

2. 1D general case

Main implementation issue is the limited storage inside Imagine cluster. In general case we have to store all elements of the stencil inside the cluster plus, perhaps, additional constants needed for computation. In our test case these additional constants are convolution coefficients. KernelC allows to store values either in LRFs (as scalar variables or as expand arrays) or in the scratchpad (as persistent arrays). In the latter case the performance of the kernel is limited by the bandwidth of the scratchpad, which has only one read port.

We found that when both stencil elements and convolution coefficients are placed in LRFs, only relatively small stencils (less than 20 elements) can be efficiently supported, i.e. KernelC scheduler fails trying to schedule software pipelined code because of register allocation failure. To explore the limits of Imagine architecture and programming system we considered two other possible options: put stencil elements into scratchpad and coefficients into LRFs, or put everything into scratchpad. We expected the latter case to be able to support larger stencils.

Another problem we found when experimenting with KernelC code for different stencil sizes is inability of KernelC compiler to restructure computations. For example, the convolution is calculated as a sum of products:

$$Conv = c_0*x_0 + c_1*x_1 + c_2*x_2 + c_3*x_3 + c_4*x_4 + c_5*x_5 + c_6*x_6 + c_7*x_7 \quad (1)$$

The part of resulting schedule is shown in Fig.2. As one can see from Fig.2 the execution time of scheduled code is not limited by any resource, it is limited by the linear dependence between operations. Software pipelining can improve performance but not for the case of large stencils, when the scheduler fails because of register file overflow. Clearly, it is possible to improve scheduling just by restructuring the computation itself.

We tried to improve our results by generating convolution code as a binary tree:

$$Conv = ((c_0*x_0 + c_1*x_1) + (c_2*x_2 + c_3*x_3)) + ((c_4*x_4 + c_5*x_5) + (c_6*x_6 + c_7*x_7)) \quad (2)$$

In this case the critical path length is proportional to logarithm of the number of terms. Although operation dependence does not limit performance such code does not work for large stencils

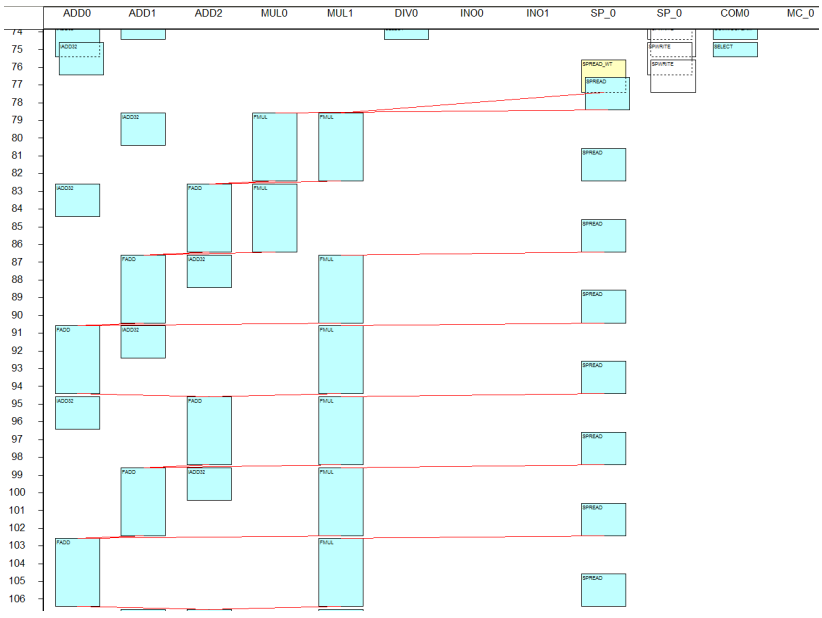


Fig.2. Schedule with linear operation dependence

because of another scheduler problem – “lazy” scheduling. This is illustrated in Fig.3. KernelC scheduler tries to schedule operation as late as possible, i.e. such that it does not increase the length of the critical path. Such scheduling policy results in maximizing the lifetime of temporary variables in the registers and overflow of register file. In the example shown in Fig.3 the data is stored in the scratchpad but the scheduler essentially tries to read all the data from the scratchpad to the temporary variables in register files and only after that perform calculations. As a result the capacity of scratch pad is irrelevant

To overcome this problem we generated convolution code as asymmetric tree:

$$Conv = (((c_0*x_0 + c_1*x_1) + (c_2*x_2 + c_3*x_3)) + (c_4*x_4 + c_5*x_5)) + (c_6*x_6 + c_7*x_7) \quad (3)$$

In this case the critical path length can be adjusted according to the usage of the most critical resource, i.e. scratchpad in our case. Using this approach we were able to maximize the size of stencils supported by Imagine.

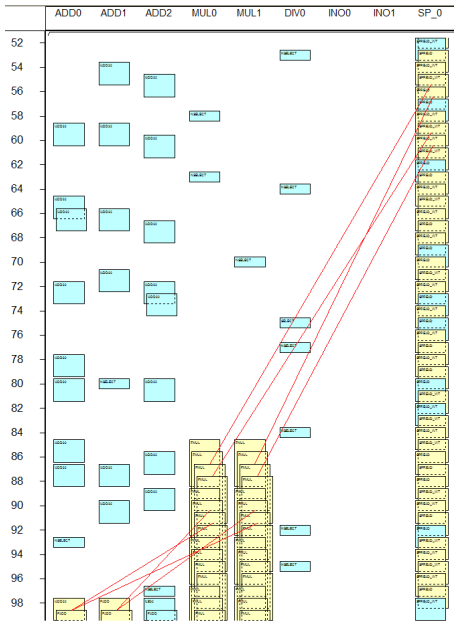


Fig.3. Schedule with “binary tree” dependence

Fig.4 shows the results for various stencil sizes. The x axis is the stencil size: negative part for the case [-x, 0], i.e. stencil includes current stream element plus some number of elements to the left; positive part is for the case [0, x], i.e. stencil includes current stream element plus some number of elements to the right. Dark blue line shows the number of cycles per iteration of the main loop, i.e. per one element of the input stream, without software pipelining. Software pipelining is very effective for small stencils: it reduces cycle count by a factor of more than 2. However, the scheduled code never achieves arithmetic bandwidth limit (light blue line in Fig.4) or scratchpad limit (yellow line in Fig.4). This is because scheduler almost never schedules scratchpad reads and writes in the same cycle.

As the stencil size increases, the software pipelining becomes less effective: for [-63, 0] stencil the difference between pipelined and non-pipelined versions is only about 15%. For large stencils

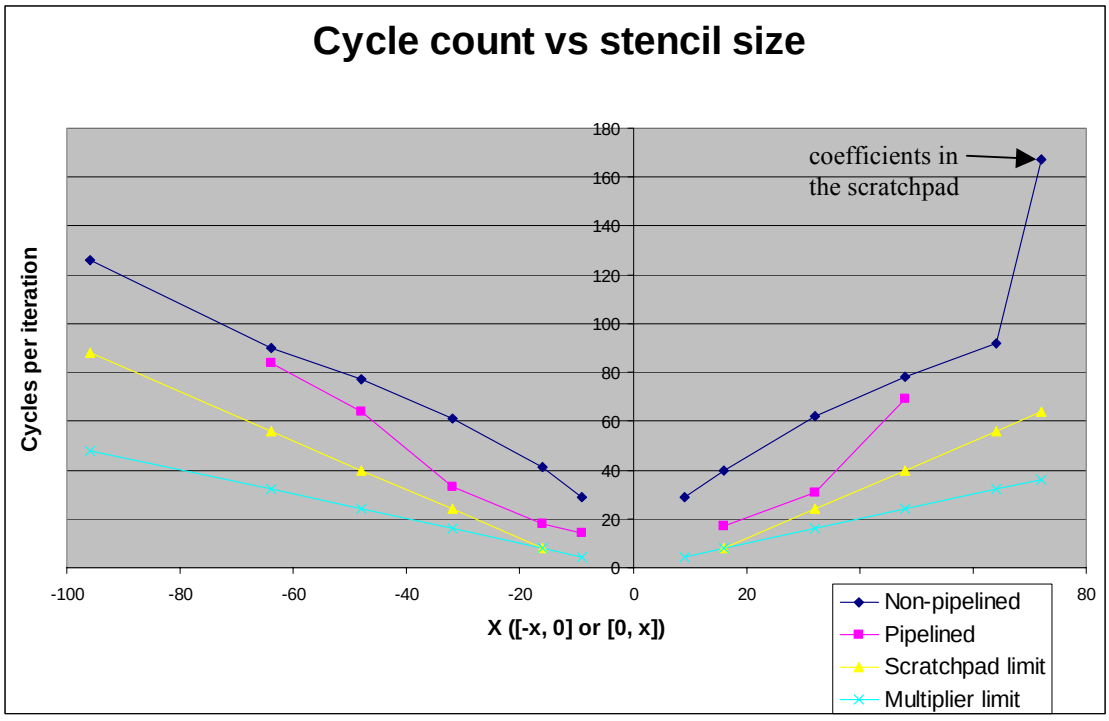


Fig.4. 1D stencil general case results

scheduler fails when it tries to compile code with software pipelining. Fig.4 shows that we are able to support stencils of up to [-95, 0] and up to [0, 71]. “Positive” stencils require to keep more state because the kernel should read stream elements to the right of the “zero” element before computation can be performed. We believe that this is the reason why the “positive” range is shorter than “negative”.

We found that placing convolution coefficients into the scratchpad does not significantly increase the range of stencils we can support. Kernel for stencil [-95, 0] can be scheduled whether the coefficients are in the LRFs or in the scratchpad, kernel for stencil [-103, 0] cannot be scheduled whether the coefficients are in the LRFs or in the scratchpad. The largest “positive” stencil with convolution coefficients in LRFs, which can be successfully scheduled, is [0, 63]; the largest “positive” stencil with coefficients in the scratchpad is [0, 71]. This result is counterintuitive since we expected to be able to support stencils of up to [-127, 0] and [0, 127] when both stream elements and convolution coefficients are placed in the scratchpad. Possible explanation is inefficient register allocation and instruction scheduling in KernelC compiler, which create too many temporary variables and overflow register file.

3. 1D associative case

We noticed that convolution computation can be broken into several parts, which can be performed independently. For example, convolution computation with 16 terms can be broken into 2 parts:

$$Conv_n = \sum_{i=0}^{15} c_i \times x_{n-i} = \sum_{i=0}^{7} (c_i \times x_{n-i}) + \sum_{i=8}^{15} (c_i \times x_{n-i}) \qquad (4)$$

Each part can be computed independently, and the final result will be just a sum of two parts. This property can be used for optimization of KernelC code: each iteration of the loop a cluster reads 1 stream element and performs communication to “assemble” 7 other sequential elements

inside. After the cluster received 8 sequential elements it can calculate 2 partial convolutions described in (4). One of them will be immediately used to calculate the output value for current iteration, while the other will be stored for the next iteration. The result of this optimization is the reduced storage requirements: instead of storing 8 input stream elements inside the cluster we need to store only 1 partial convolution. In general, for large stencil the amount of data that should be stored from iteration to iteration will be reduced by a factor of 8, e.g. for stencil [-63, 0] we have to keep only 7 partial convolutions for the next iteration instead of 56 stream elements.

A more subtle point is the reduction in bandwidth requirements: the kernel performs multiple partial convolutions on the same 8 elements of the stream, which are allocated inside LRFs, instead of reading all stencil elements from the scratchpad and calculating 1 full convolution.

The results are shown in Fig.5. Blue line shows cycle count per loop iteration for the case when convolution coefficients are in the LRFs. Cycle count is almost always arithmetic limited, i.e. limited by the multiplier. Cycle count for small stencils exceeds arithmetic limit because of inter-cluster communication. For largest stencils software pipelining is less effective because of the register file pressure.

We have tried another case: when convolution coefficients are placed in the scratchpad. Again we expected to be able to handle larger stencils with coefficients in the scratchpad although with lower performance limited by the bandwidth of the scratchpad. The result is shown in Fig.5 as a red line. It turns out that the largest stencil, which can be supported in this case is smaller. The result is again counterintuitive since we expected that usage of scratchpad for coefficients would reduce the number of registers required.

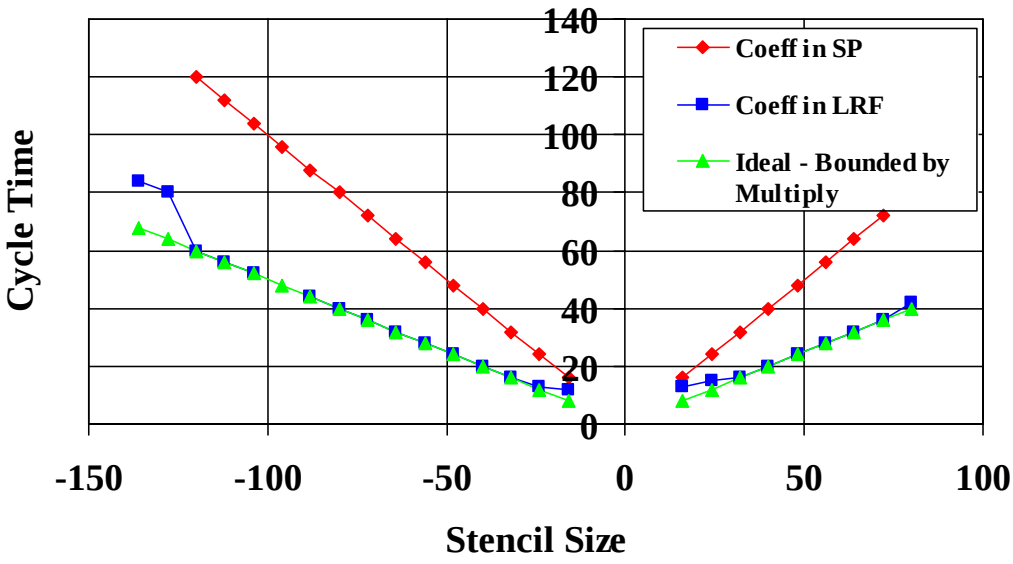


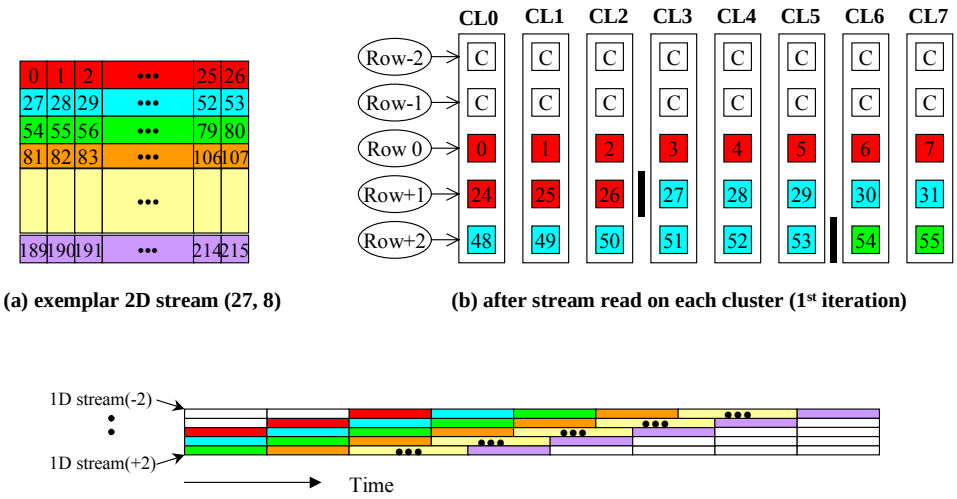
Fig.5. 1D stencil associative case results

We believe that associativity is common for many computational kernels; therefore, it would be useful to incorporate such optimization in the future Brook compiler.

4. 2D general case

General 2D case is handled by dividing W_{sten}-by-H_{sten} 2D stream into H_{sten} 1D streams. Then we can deal with each of several 1D streams independently in the same way as in 1D case, but there are also some issues specific to 2D case to guarantee correct functionality. These issues are discussed in the next several paragraphs.

Across-boundary access: If an element is accessed across a boundary of row (indicated by a thicker vertical line in Fig. 6 (b)), clamp constant should be read instead of an element from the stream. We have a variable in 2D stencil template to keep track of the location of boundary.



(a) exemplar 2D stream (27, 8)

(b) after stream read on each cluster (1st iteration)

(c) 5 shifted 1D streams to feed into kernel

exemplar 2D stencil: streamStencil(t, s, 2, STREAM_BOUND_HALO, -2, 2, -2, 2)

Fig.6. Implementation of a 2D stencil ([-2, 2]*[-2, 2])[1]

Misalignment between rows: If the width of 2D stream is not a multiple of 8, several 1D streams don't align to each other. (See misaligned elements 0, 27, and 54 in Fig. 6 (b)). Our template groups corresponding elements on the same column by proper shift operation.

Management of multiple streams: Currently, a kernel can manage up to 8 streams at the same time. In case of $H_{sten} > 8$, we can handle it with a preprocessing the stream for consolidation.

Scheduling issue: Since we are dealing with H_{sten} independent 1D streams, we expected that its resource requirement will be close to H_{sten} times as much as that of 1D stencil of length W_{sten}. In our simulation, it turned out that the requirement is even worse than this first-order expectation;

[1] [C] in Fig. 6 (b) denotes clamp constant which is 0 by default.

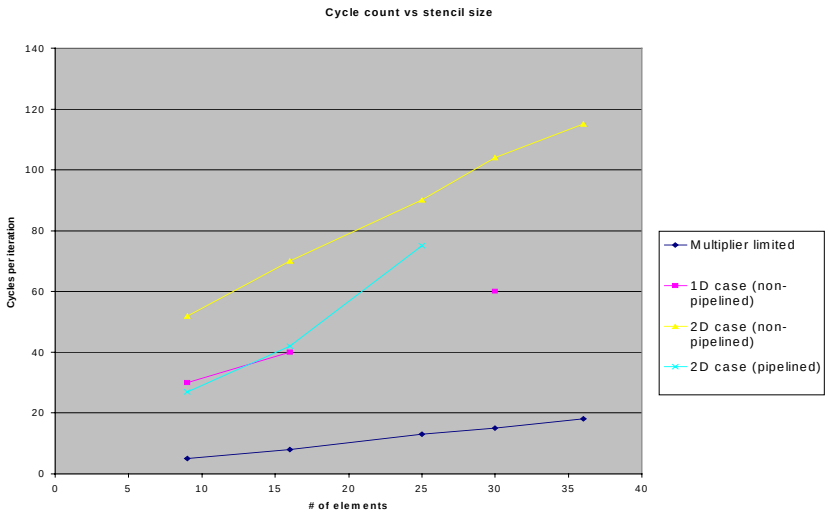


Fig.7. 2D stencil general case results

the largest 2D stencil we succeeded in scheduling with 2D convolution kernel has a size of 6-by-6 (36 elements). For any larger 2D stencil, register allocation failure occurs.

Fig.7 shows cycle counts for different size of 2D stencils. For all of the overhead mentioned above, a 2D stencil takes more cycles to execute than a 1D stencil with the same number of elements. Pipelining reduces the cycle counts nearly by a factor of 2 for small stencils, but it becomes less effective as the size of stencil grows as in 1D case. Basically, this implementation of 2D stencil is limited in performance by the same reason for 1D case, because the former is an extension of the latter. Unlike 1D case, we have tested only the case that all of the stencil values and history variables are stored in scratch pad, and the convolution coefficients in local register file. Another thing to mention is that the cycle count is not affected by the range of offsets in stencils very much; that is, [-2, 2]*[-2, 2] stencil takes almost the same number of cycles as [0, 4]*[0, 4] stencil.

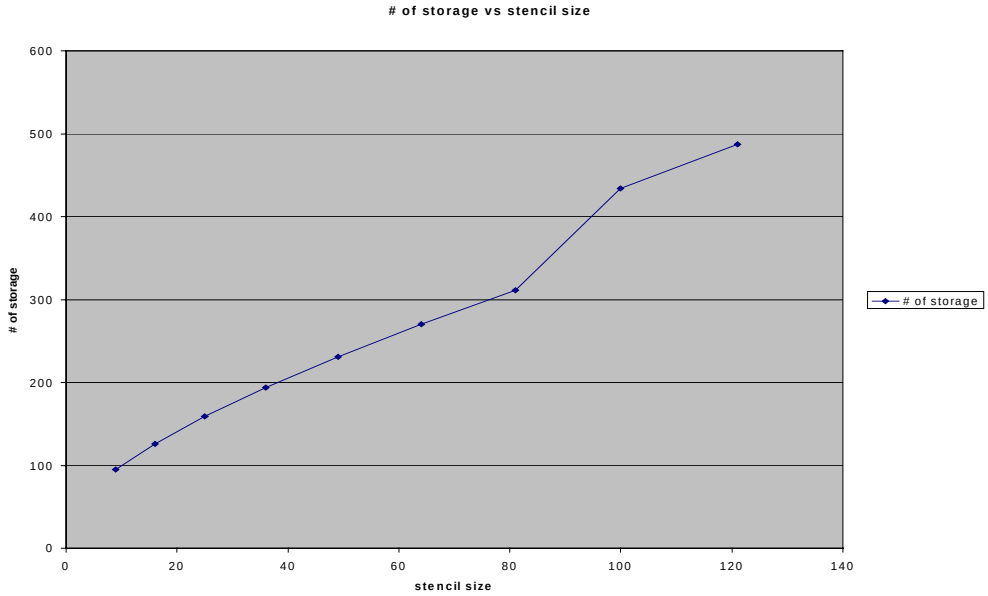


Fig.8. Storage size vs. stencil size

On the other hand, there is a limitation to storage (i.e. registers) in each cluster, which prevents a large stencil from being scheduled. According to our 2D stencil template (refer to Appendix A for details), the number of storage[2] is given as follows:

$$(\#\text{ of storage}) = \left(\left\lceil \frac{W_{sten} + (n_c - 1)}{n_c} \right\rceil + 1 \right) \times n_c \times H_{sten} + W_{sten} \times H_{sten} + C$$

$$where\ n_c = 8 \text{ and } C = (\text{const overhead} + \text{app specific overhead})$$

Here the first term corresponds to storage for history variables which store the value of past iteration, the second to storage for stencil itself, and the third to constants and computations specific constants.

As you can see in Fig.8 for 7-by-7 stencil this number is already more than 200, which causes scheduling failure because of register allocation error[3]. In general, 2D case requires more

[2] The actual number of registers needed is a little bit larger than this number, because this number counts the number of variables in kernelC, but we also need registers to keep intermediate results which are used without explicit assignment to a variable. However, this number is still okay for first-order estimate of storage requirement.

temporary registers than 1D case of the same size. Therefore, for stencils of a larger size, we need to aggressively exploit the properties of computation (like associativity in the next section).

5. 2D associative case

Using the concept described in section 3, we can optimize the kernels for 2D stencils. If the computation over stencil can be divided into several independent parts such that, each of them needs only elements from the same row of the stream, then the kernel can do computation on each row independently and store partially computed results. In general case the width of the 2D stream can be very large, so storing all intermediate results for the row inside the cluster is not possible. Instead of the scratchpad or the LRF, we use the SRF. Therefore, all partially computed results should be stored in the intermediate stream(s), and the kernel should be invoked separately for each row of 2D stream.

For simplicity, we implemented kernels for the case when stencil size and row length are multiples of 8. Fig.9 shows cycle count per iteration for convolution kernel with various stencil sizes. Three curves show cycle counts versus stencil height for 3 different stencil widths: 8, 16 and 24 elements. In most cases cycle count is limited by the arithmetic. The maximum possible stencil size, i.e. the total area=width*height, which can be handled is larger than in 1D case because intermediate results are now stored in the SRF instead of local register files.

[3] A jump between size of 81 and 100 is caused by wrap-around of history variables; that is, in the latter case, we need to look back one more iteration to get all of the values in the stencil. This jump happens every time when W_{sten} is increased by 8 due to ceiling function.

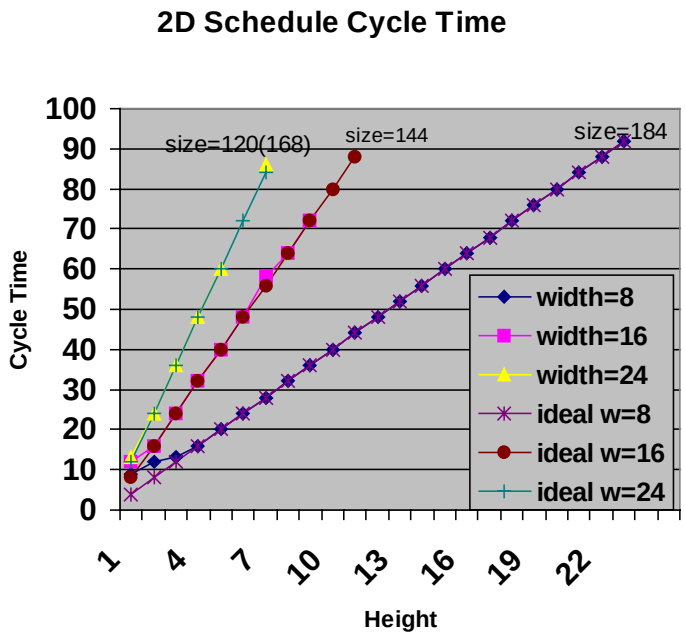


Fig.9. 2D stencil associative case results

6. Future work

This work can be extended in many directions. The most obvious is the higher dimension streams: 3D and so on. Another direction is stencils which are not rectangular, e.g. 2D stencil consisting of only right, left, top, and bottom neighbors but not including diagonal neighbors. Yet another direction is to support Brook streamGroup operator.

In our project we used KernelC as a target representation for Brook compiler. However, KernelC is not the best choice. First of all, as we discussed in previous sections current KernelC scheduler has problems such as inability to restructure computation and "lazy" instruction scheduling, which often make scheduling inefficient or even impossible. In our project we had to massage of KernelC code to achieve the best results.

Second, KernelC itself may be too high level representation: for example, it does not allow input-output streams, which would simplify storage of intermediate results in the SRF. Although it is

clear that input-output stream should be prohibited in the high level language (Brook) used by the programmer, they would be handy for compiler target language. This is similar to pointer arithmetic in standard programming languages: pointer arithmetic is prohibited in some high level languages such as Pascal but it is available in the assembly, which is the target for Pascal compiler.

Another useful feature is the ability to set conditional stream pointer to any element within the part of the stream in the SRF. This would greatly simplify the code for the 2D case when stencil size or row length is not a multiple of 8.

References

[1] http://cva.stanford.edu/ee482s/index.html

[2] I. Buck, “Brook: A Streaming Programming Language”, http://graphics.stanford.edu/streamlang/brook_v0.2.pdf