Mapping the Conjugate Gradient Algorithm onto High Performance Heterogeneous Computers

Anas M. Alfarra
Electrical & Computer Engineering Dept.
Jackson State University
Jackson, MS
alfarra.anas@gmail.com

Jamory D. Hawkins
Electrical & Computer Engineering Dept.
Jackson State University
Jackson, MS
jamoryhawkins@gmail.com

Gerald R. Morris
Information Technology Lab.
US Army Engineer
Research and Development Center
Vicksburg, MS
gerald.r.morris@us.army.mil

Khalid H. Abed
Electrical & Computer Engineering Dept.
Jackson State University
Jackson, MS
khalid.h.abed@jsums.edu

Abstract

Mapping scientific kernels onto high performance heterogeneous computers (HPHCs) must comply with certain rules of thumb or heuristics. Previous research by Jackson State University's (JSU) HPHC research group has provided anecdotal evidence illustrating some of these rules/heuristics. The research highlighted by this paper corroborates the credibility of these rules. In particular, four versions (two pairs) of a floating-point sparse matrix conjugate gradient (CG) iterative solver are presented. JSU's state-of-the-art HPHC utilizes general purpose processors (GPPs) and heterogeneous computational hardware, in particular, a field programmable gate array (FPGA), to develop the CG kernels. The first version of the pair executes strictly on the GPP and the second uses both the GPP and FPGA to map the entire CG algorithm onto hardware. For the second pair, a refactored version of CG is used, which is statically analyzed to determine where the most computationally expensive operation occurs. This operation is the sparse matrix vector multiply (MVM) kernel. Based on this analysis, the software version of CG is refactored to call MVM as a subroutine. An FPGA version of the MVM algorithm is also developed and a static analysis of that algorithm suggests a speedup of the MVM kernel. All four versions of CG are executed using a specially designed set of sparse matrices and the results demonstrate that adherence to the rules of thumb and heuristics when mapping scientific kernels onto HPHC can lead to significant speedups.

1. Introduction

HPHCs, such as field programmable gate array (FPGA)-augmented reconfigurable computers (RC), can sometimes outperform their general purpose processor (GPP)-based counterparts. In the past, lack of support for floating-point arithmetic within FPGA tool suites often forced designers to use fixed-point or integer arithmetic [1]. Now, semiconductor technology scaling allows companies such as Altera to fit floating-point intellectual property (IP) cores onto contemporary FPGAs. As a result, there have been some successes at mapping iterative solvers onto FPGAs [2, 3, 4]. For floating-point applications, FPGA-based processors must satisfy several heuristics and rules of thumb to achieve a speedup compared with their GPP counterparts. This paper highlights the challenges in the computational mapping process while simultaneously showing that such mappings can result in significant speedups. The focus is to show the importance of “the three P’s,” which expresses the crucial relationship among performance, pipelining, and parallelism as well as several of the other heuristics[5]. This paper is organized as follows: Section 2 provides background on HPHC design considerations. Section 3 introduces the conjugate gradient method and a high-level design of the solver. Section 4 details the FPGA-based solver design. Section 5 describes the HPHC hardware and details the implementation. Section 6 describes the experiments, compares the runtime performance of the two conjugate gradient versions, and provides an analysis of the results. Section 7 presents the conclusions.
2. HPHC Design Considerations

The JSU HPHC research group has developed a set of heuristics to determine if a given algorithm is suitable for mapping onto HPHCs [6, 7]. These include a) the three p’s, b) resource utilization, c) control and memory intensive vs. compute intensive, d) monolithicity of the algorithm, e) available bandwidth, f) ability to reuse data, g) design stability of the algorithm, h) efficiency of the algorithm, and i) memory access patterns. As an example, a common approach to estimate speedup is via Amdahl’s Law shown in Equation 1,

\[ s_o = \frac{1}{1 - \frac{f_e}{f_e + f_e/s_o}} \tag{1} \]

Based on earlier work, a conservative value is \( s_o \approx 10 \) for the CG algorithm. A profile of the software version of CG showed matrix vector multiply (MVM) consumed nearly 67 percent of the runtime. By Amdahl’s Law, an overall speedup \( s_o = 1/(0.33 + 0.67/10) = 2.5 \) is anticipated.

3. Conjugate Gradient

3.1. CG Algorithm

Discussions of the CG iterative method can be found in many introductory numerical analysis textbooks including [8, 9]. CG is typically used when \( A \) is a sparse matrix. Sparse matrices are usually represented in a compressed format, which only stores the non-zero elements and provides some bookkeeping mechanism for determining the row and column number of each matrix entry. A representation of the conjugate gradient algorithm (as implemented in software) is shown in Figure 1 [10].

3.2. CG Operation

An arbitrary starting point \( x_0 \) is selected, from which the descent proceeds. The first \( A \)-orthogonal vector, \( p_1 \), is a search direction opposite to the gradient as depicted in line 2, therefore, the initial search direction is also the same as the initial residual as illustrated on line 3. The next line is a criteria for forcing the algorithm to enter the loop. Line 5 is part of the equation for calculating the residual. Rather than continually computing this value, it is simply calculated one time. At line 6, the body of the loop requires the dot product of the current residual, \( r_k \), and the previous residual, \( r_{k-1} \). Instead of calculating two dot products within the loop, the dot product from the previous iteration was reused, i.e., \( r_k^T r_{k-1} \). Line 7 creates the iteration index, \( k \), which is also part of the criteria for the while loop to prevent infinite looping. It can be shown that the matrix vector product is needed twice within CG, however, it is computed once on line 9 because it is an expensive \( O(n^2) \) algorithm. The next line computes the step size, \( \alpha_k \), along the direction of the residual. The first \( A \)-orthogonal vector, \( x_1 \) is illustrated in Figure 1.

\begin{verbatim}
1: algorithm CGSW(A, x, b)
2:  \( p_1 \leftarrow b - Ax_0 \)
3:  \( r_0 \leftarrow p_1 \)
4:  \( \Delta \leftarrow \varepsilon + 1 \)
5:  over\_norm \leftarrow 1/\|b\|
6:  \( r_{Told} \leftarrow r_0^Tr_0 \)
7:  \( k \leftarrow 1 \)
8:  while (\( \Delta > \varepsilon \)) AND. (\( k < k_{max} \)) do
9:      \( v_{ap} \leftarrow Ap_k \)
10:     \( \alpha_k \leftarrow r_{Told}/p_k^Tv_{ap} \)
11:     \( x_k \leftarrow x_{k-1} + \alpha_k p_k \)
12:     \( r_k \leftarrow r_{k-1} - \alpha_k v_{ap} \)
13:     \( r_{Tnew} \leftarrow r_k^Tr_k \)
14:     \( \beta_k \leftarrow r_{Tnew}/r_{Told} \)
15:     \( r_{Told} \leftarrow r_{Tnew} \)
16:     \( p_{k+1} \leftarrow r_k + \beta_k p_k \)
17:     \( \Delta \leftarrow \|r_k\| \cdot over\_norm \)
18:     \( k \leftarrow k + 1 \)
19:  end while
20: return (\( x_{k-1} \))
21: end algorithm
\end{verbatim}

Figure 1. Software CG algorithm

CG. For line 11, the new approximation for \( x \) is calculated by descending in the conjugate search direction a distance of step size. Line 12 computes the new residual. Then the dot product of the new residual, \( r_{Tnew} \), is calculated, which will be used in subsequent computations. Line 14 computes the projection operator \( \beta \) which removes from the residual all previous search directions. Since the dot product of the previous residual is no longer needed and to prevent calculating the dot product at the next iteration, line 15 retains the current residual for the next iteration. The new search direction can finally be calculated, by using the projection operator to remove the residual from all components along the previous conjugate search directions, i.e., \( r_k + \beta k p_k \). Line 17 computes the residual norm to check the algorithm for convergence. Line 18 increments the iteration index. If the algorithm has converged, the solution, \( x_{k-1} \), is returned.

4. CG Processor Detailed Designs

4.1. Two CG Versions

As mentioned above, two versions of CG were implemented in hardware. The first (monolithic) version offloads the entire CSR-based CG algorithm onto the FPGA without regard to the heuristics that previous research has shown to significantly impact performance. The second version offloads only the MVM kernel (which comprised some 67% of the software run time). In the algorithms below, compressed sparse row (CSR) format is used.
4.2. CG Hardware Design

A monolithic hardware version of CG, which was mapped onto the FPGA hardware, is shown below. The monolithic CG algorithm that was executed in software is essentially the algorithm depicted in Figure 1.

1: algorithm CGHW(kval, kcol, kptr, b, h, s, n, knz, x)
2:  parBegin // only two GCM banks so
3:  BUF_DMA_GCM1:OBM (kval, stripe-8)
4:  BUF_DMA_GCM2:OBM (kcol, stripe-8)
5:  parEnd
6:  parBegin // parallel DMA limited to 2
7:  STREAM_dmaGCM:BRAM (kptr)
8:  STREAM_DMA_GCM:BRAM (b)
9:  parEnd
10: for i in [1, n] do // x(0) = 0 : p(1) ← r(0) ← b
11:  xi ← 0
12:  p1 ← r1 ← b1
13:  MAC(b1, r1, rTrold) // calculate r(0)T r(0)
14: end for
15: δ ← 1
16: repeat
17: p1 ← ... ← p11 ← p
18: parBegin // compute v ← Ap(δ)
19: p1: // dotS's into V FIFO stream
20: for i in [1, knz] do
21: aS ← (a1 ··· aS) stripe-8 from kvali
22: aS ← (j1 ··· jS) stripe-8 from kcoli
23: Ps ← (p1j1 ··· p8jk)
24: V FIFO ← aS + Ps
25: end for
26: p2: // # dotS's per row into C FIFO stream
27: for i in [1, n] do
28: C FIFO ← kptri+1 − kptri
29: end for
30: parEnd
31: p3: // n dot products into S FIFO
32: S FIFO ← ΣSTREAM(V FIFO, C FIFO)
33: p4: // dotS = aT(δ)
34: for i in [1, n] do
35: vi ← S FIFO
36: MAC(vi, p9i, pTv) // calculate p(δ)Tv
37: end for
38: parEnd
39: α ← rd/IVAL // step size
40: for i in [1, n] do
41: xi ← xi + αpi
42: end for
43: MAC(r1, r1, rTrnew) // calculate r(δ)T r(δ)
44: end for
45: β ← rTrnew/rTrold // projection operator
46: rTrold ← rTrnew
47: for i in [1, n] do // new search direction: p(k+1)
48: p1i ← r1 + βpi+1
49: end for
50: r2b2 ← rTrold · b2 // ||p(δ)|| = rTrold
51: δ ← δ + 1
52: until (r2b2 ≤ ε OR δ > δmax)
53: BUF_DMA_BRAM_GCM1 (x)
end algorithm

Figure 2. Monolithic hardware CG algorithm

4.3. MVM Hardware Design

Because of the significant speed degradation of using multiple serialized multiply-accumulators (MACs) within the hardware version of CG after the parallel sections finish, a CSR-based MVM kernel was mapped onto the HPHC. All other modules of CG are executed via the GPP. The principle speedup is obtained via the four parallel sections p1 through p4, which communicate via a set of FIFO streams. This hardware MVM algorithm is idealized in Figure 3 and operates in three phases: input, compute, and output. During

1: algorithm MVMHW(kval, kcol, kptr, v, p, *first)
2: if (*first) then
3: parBegin // only two GCM banks so
4: BUF_DMA_GCM1:OBM (kval, stripe-8)
5: BUF_DMA_GCM2:OBM (kcol, stripe-8)
6: parEnd
7: parBegin // parallel DMA limited to 2
8: STREAM_dmaGCM:BRAM (kptr)
9: STREAM_DMA_GCM:BRAM (b)
10: parEnd
11: else
12: STREAM_DMA_GCM:BRAM (p)
13: end if
14: p1 ← ... ← p8 ← p
15: parBegin
16: p1: // feed values stream
17: for i in [1, knz] do
18: a ← (a1 ··· aS) stripe-8 from kvali
19: j ← (j1 ··· jS) stripe-8 from kcoli
20: p ← (p1j1 ··· p8jk)
21: V FIFO ← dot8Tree (a, p)
22: end for
23: p2: // feed counts stream
24: for i in [1, n] do
25: C FIFO ← kptri+1 − kptri
26: end for
27: p3: // streaming accumulator
28: S FIFO ← ΣSTREAM(V FIFO, C FIFO)
29: p4: // the dotN's
30: for i in [1, n] do
31: v ← S FIFO
32: end for
33: parEnd
34: STREAM_DMA_BRAM_GCM1 (v)
end algorithm

Figure 3. Sparse MVM Hardware Algorithm

input, two parallel blocks use direct memory access (DMA) to input the problem data from global common memory (GCM). Lines 3–10 constitute the compute phase. Since the FPGAs do not have multiorbit memory to support eight address and data buses on a single memory bank, and since parallel sections p1 . . . p3 operate simultaneously, independent banks are needed to avoid a multicycle pipeline. Therefore, line 14 creates eight copies of p for the dot product tree.
Parallel section \( p_1 \) (lines 15–21) is a fully pipelined 8·8 dot product unit. Each clock cycle it consumes the next eight \( a_{ij} \) values from \( kval \) and the matching eight values from \( p \) and outputs the resulting partial dot products (dot8s) to the \( V_{FIFO} \) stream. Parallel section \( p_2 \) (lines 22–24) calculates the number of dot8s for each row and sends them to the \( C_{FIFO} \) stream. Parallel section \( p_3 \) (line 25) is the streaming accumulator that consumes the \( V_{FIFO} \) and \( C_{FIFO} \) streams, computes the \( n \) dot products, \( \text{dotN}_i = \sum_j a_{ij} p_j \) for all \( i \), and feeds the results into the \( S_{FIFO} \) stream. Parallel section \( p_4 \) (lines 26–29) consumes the \( \text{dotN}_i \) from \( S_{FIFO} \) and streams the resultant vector into \( v_i \). During output, the resultant \( v \) is then DMAed to GCM as shown on line 30.

5. Implementation

5.1. High-level CG Design

The high-level design for the CG solver is shown in Figure 4. It consists of four major components: a main routine and matrix support libraries; several symmetric positive definite sparse matrices, \( A_1 \ldots A_m \); the software or hardware (FPGA-based) CG solver; and the output result and statistics files, \( x_1 \ldots x_m \) and \( \theta_1 \ldots \theta_m \). The \( b \) vectors are shown as inputs, but for the experiments in this research they are generated from a known \( x \) vector at runtime. The main routine is a driver program, which essentially measures how long it takes for CG to solve each set of equations. The coordinate-format matrices are read in using the Matrix Market I/O library [11] and converted to CSR format using Saad’s SPARSKIT library [12]. The software CG kernel implementation is based on the algorithm shown in Figure 1, and the FPGA-based CG kernel is based on the algorithm shown in Figure 2.

A compile-time decision selects either the software or FPGA-based version of CG. At runtime, main reads in each coordinate-format matrix, converts it to CSR format, and uses a known \( x \) vector to generate \( b \). It then invokes the selected CG kernel sending matrix, \( A \) (val, col, and ptr), starting point \( x^{(0)} \), and constant vector \( b \). After convergence, CG stores the result and returns. The main routine writes the solution to the results file; it also writes the input matrix name, number of iterations, and wall clock execution time to the statistics file and then terminates.

5.2. High-level MVM Design

To recap, there were four CG algorithms that were mapped onto the HPHC. The first pair was a monolithic version which comprised of a software and hardware version. The second pair also had a software and hardware version, however, this kernel involved a subroutine call that implemented MVM in software or hardware as illustrated in Figure 5. Macroscopically, the operation is similar to the operation of the monolithic CG described in the previous subsection.

5.3. Implementation Summary

The software and hardware CG designs described in Section 5 were coded for the SRC-7 platform. The same set of files were used in both implementations to ensure a valid side-by-side comparison. The only difference was the CG kernel. Most of the work involved implementing the hardware modules. The \( k = 8 \) dot product width was limited by the available configurable logic on the FPGA. Thus, only eight values per clock cycle are read from \( kval \) and \( kcol \). Since the \( kval \) and \( kcol \) CSR arrays are placed in OBMs and each OBM holds about half a million entries, the largest number of nonzero values that can be processed is \( n_{\text{max}} \approx 4M \). Lastly, as most of the BRAM stores the multiple copies of \( x \) needed to fully pipeline the loop, the largest matrix was limited to \( n_{\text{max}} = 8,192 \).
6. Results

6.1. UFL Sparse Matrix Collection

The matrices used to test CG came from the University of Florida Sparse Matrix Collection, managed by Dr. Tim Davis and Dr. Yifan Hu. This repository is a large and actively growing set of sparse matrices widely used by the numerical linear algebra community for the development and performance evaluation of sparse matrix algorithms [13] such as conjugate gradient. The collection covers a vast spectrum of domains ranging from mathematics and physics, to civil engineering and computer science. With over 2547 matrices, UFL’s Sparse Matrix Collection boasts its largest matrix as having a dimension of 118 million with almost 2 billion nonzero entries.

6.2. Monolithic Results

As shown in table 1, the monolithic approach, which ignored the heuristics developed by the JSU HPHC research group, did not result in a speedup. In fact, it resulted in a slowdown as depicted by the average of six samples of matrices.

6.3. MVM-only Results

In contrast, following the heuristics discussed in Section 2, it is possible to achieve a speedup. At the time we were preparing this paper, our target platform was undergoing upgrades at the vendor’s site so we do not yet have the results of the refactored CG hardware mapping. However, previous research mapping a Jacobi iterative solver via an analogous set of scenarios [5] resulted in a 3-fold increase in performance as illustrated in Figure 6. Given the slight increase in algorithmic complexity of CG versus Jacobi, the 3-fold speedup seen for Jacobi is in excellent agreement with the anticipated 2.5-fold speedup described in Section 2.

7. Conclusion

The research thus far has suggested strong evidence of an overall speed up of conjugate gradient (CG) when the rules and heuristics developed by the Jackson State University’s HPHC research group are adhered to. The negative impact of performance has also been illustrated when these rules and heuristics are ignored [14]. Finding the “sweet spot” when mapping scientific kernels onto HPHCs has indeed helped in demonstrating the value of using the heuristics developed by the JSU HPHC group during their extensive research within the HPHC domain [15, 16, 17, 18]. Because our target platform was being upgraded from ARO funds when this paper was drafted, we used the analogous set of results from earlier research [5] as shown in Figure 6. We plan to confirm our results at the earliest opportunity.

Acknowledgments

This work was supported in part by the Army Research Office grant number W911NF-07-1-0527, and in part by the U.S. Army Engineer Research and Development Center.
References


