Augmenting the Roofline Model via Lower Bounds on Data Movement

Technical Report OSU-CISRC-4/14-TR10, April 2014

V. Elango*, N. Sedaghati*, F. Rastello†, L.-N. Pouchet‡, J. Ramanujam§, R. Teodorescu*, P. Sadayappan*
*The Ohio State University †INRIA ‡University of California at Los Angeles §Louisiana State University

Abstract

The roofline model is a popular approach to bottleneck-bounds performance analysis. It models upper bounds on performance as a function of operational intensity, ratio of arithmetic operations per byte of data moved from/to memory. While operational intensity can be measured for a given implementation, it is of interest to characterize the upper limits of operational intensity for different algorithms. Currently there is no systematic systematic methodology to do so. In this paper, we address the problem of developing upper bounds on the operational intensity of computations as a function of cache capacity. We demonstrate the utility of the approach in comparative analysis of algorithms, as well as in assessing fundamental performance limits for different algorithms as architectural parameters are varied.

I. Introduction

Advances in technology over the last few decades have yielded significantly different rates of improvement in the computational performance of processors relative to the speed of memory access. Because of the significant mismatch between computational latency and throughput when compared to main memory latency and bandwidth, the use of hierarchical memory systems and the exploitation of significant data reuse in the faster (i.e., higher) levels of the memory hierarchy is critical for high performance. Although hardware techniques for data pre-fetching and overlapping of computation with communication can alleviate the impact of memory access latency on performance, the mismatch between maximum computational rate and peak memory bandwidth is much more fundamental; it is of critical importance to limit the volume of data movement to/from memory by enhancing data reuse in registers and higher levels of the cache.

The roofline model [1] is a popular approach to bottleneck-bounds analysis that focuses on the critical importance of the memory bandwidth in limiting performance. In its most basic form, a performance roofline

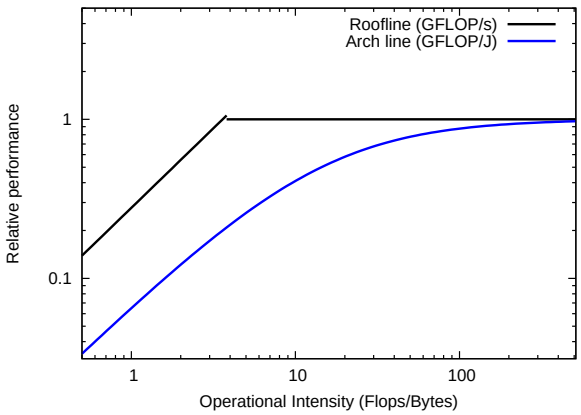

Fig. 1: Roofline model: Performance and Energy rooflines

model comprises of two straight lines that represent asymptotic upper bounds on performance limits due to the processor cores and memory bandwidth, respectively. This is shown by the black lines in Fig. 1. The horizontal line represents the peak computational performance, in GFLOPs, of the processor. The inclined line has a slope corresponding to a critical machine balance parameter: ratio of peak arithmetic performance to peak main-memory bandwidth. The horizontal axis represents the "operational intensity" of the computation, defined as the ratio of number of computational operations performed per byte of data moved between main memory and the processor. The lower the operational intensity, the more constrained is the computation by the memory bandwidth. As the operational intensity increases, the upper bound on

achievable performance steadily increases up to the point of intersection of the two lines. At the intersection point, the operational intensity is sufficiently high that the available memory bandwidth is no longer a fundamental constraint on achievable computational throughput.

The roofline model has recently been adapted [2], [3] to capture upper bounds on energy efficiency, i.e., operations/Joule, as a function of operational intensity. For a particular operational intensity I, the energy cost of at least one memory access must be expended for every I computational operations. The minimal per-operation energy cost for performing a set of N operations is the sum of the energy cost for actually performing the operation and the energy cost for $\frac{N}{I}$ memory operations. Fig. 1 shows an energy roofline (in blue) for the same architecture as the performance roofline (pair of intersecting lines in black). The energy roofline is smooth since the total energy is the sum of compute and data movement energies, in contrast to execution time, which is bounded by the larger of the data movement time and compute time (since data movement and computation may be overlapped).

Different algorithms have very different inherent data reuse characteristics. Hence, a key question is: *Given a particular algorithm, for example, a* 1024×1024 *2D FFT computation, what is the appropriate value to use for operational intensity in using the roofline model?* The challenge is that the number of operations performed per byte of data moved between main memory and the processor is not some constant for a given computation, but can vary over a range of values for different implementations of the same set of operations. In particular, it can be significantly affected by:

- **Cache Size**: If the complete working set of a computation, i.e., data footprint can completely fit within cache, we will only experience initial cold misses from memory. This scenario corresponds to the maximal possible value of operational intensity. In practice, for most compute intensive applications of interest, caches are not large enough to hold all the data, and cache misses will result, thereby lowering the operational intensity. At the other extreme, if we have no cache, every memory reference will result in movement of data from main memory. The operational intensity for this extreme can be calculated by analysis of the source code, but will be too pessimistic and not useful in providing any guidance to the applications developer.
- **Code Transformations**: It is well known that loop tiling can result in significant reduction in the number of cache misses and therefore in the amount of data moved between main memory and cache.

Consider the simple stencil code in Fig. 2(a). For simplicity, let us only consider data moves from main-memory to the processor and not writes. Just examining the statement in the loop body shows that a total of 7 arithmetic operations are performed per iteration, and four distinct array elements are accessed. However, it would be incorrect to simply conclude that the operational intensity is 7/4. This is because across different iterations, the same array elements are accessed multiple times. Indeed, for this code, if the cache capacity were at least 2*N+1, each array element would only need to be brought in from main memory once, and it would be retained in cache for all references to it across the different iterations. In this case, we would have an operation intensity of 7. But if the cache were smaller, we would have a lower operational intensity. Finally, consider the code in Fig. 2(b). It has the same statement body as that in Fig. 2(a), but has an outer-most "time" loop around the spatial loops. In this case, there are 4*T reads of each array element, leading to the possibility of an even higher operational intensity than 7.

The example demonstrates that for a given code, the operational intensity used with the roofline model is not a fixed quantity, but depends on cache capacity, and will also be affected by dependence-preserving loop transformations. For a given implementation of an algorithm on a target system, it is feasible to use hardware counters to measure the operational intensity and thereby conclude whether or not it is memory bandwidth bound. In case the implementation is memory-bandwidth bound, it is of great interest to the applications developer to understand the prospects for improving performance by improved implementation (via code transformations such as tiling and fusion) or by changing system parameters (e.g., larger cache). We currently do not have any systematic methodology to characterize the range of feasible values of the operational intensity for a given algorithm.

In this paper, we address this problem. We make the following contributions:

- We develop techniques to bound the maximum possible operational intensity of a computation as a

```
for (i=1; i<N-1; i++)
  for (j=1;j<N-1; j++)
    A[i][j] = c1*A[i][j-1] + c2*A[i][j+1]
              + c3*A[i-1][j] + c4*A[i+1][j]);
```

(a)

```
for (t=0; t<T; t++)
 for (i=1; i<N-1; i++)
  for (j=1;j<N-1; j++)
    A[i][j] = c1*A[i][j-1] + c2*A[i][j+1]
              + c3*A[i-1][j] + c4*A[i+1][j]);
```

(b)

Fig. 2: Two stencil examples.

function of cache size. This bound applies to all possible valid schedules for the operations constituting the computation, i.e., the bound is a fundamental upper limit for the operational intensity, irrespective of any optimization/transformation like loop tiling, fusion etc.

- We demonstrate the use of bounds on operational intensity on algorithm-architecture co-design:
  1) We model maximal achievable performance for different algorithms (operations per second) for a given VLSI technology, considering different fractional chip area being allocated to cache versus cores.
  2) We model maximal achievable energy efficiency (operations per Joule) for different algorithms on a given VLSI technology, considering variation in number of cores, cache capacity, and voltage/frequency scaling.
- We demonstrate use of operation intensity bounds in comparative analysis of alternate algorithms for a problem and in assessing the scope for improvement of operational intensity, compared to that achieved by an existing implementation of a computation,

II. Background and Overview

The range of possible values of operational intensity for a particular algorithm has inherent upper limits for any given capacity of last level cache. The upper bound on attainable operational intensity for a computation is inversely related to the lower bound on the minimal number of data movement operations to execute a computation in a two-level memory hierarchy. Given an execution of a computation that executes W operations and performs D bytes of data transfer to/from main memory during execution, the operational intensity is defined as $\frac{W}{D}$. If the number of operations W is invariant across different execution scenarios to be considered (including dependence preserving program transformations and change in the cache capacity), the problem of maximizing the operational intensity is equivalent to the problem of minimizing the amount of data transfered between main memory and cache. Since it is generally infeasible to determine the highest achievable operational intensity (or equivalently the lowest achievable amount of data movement from/to main memory) among a combinatorially explosive number of configurations, it is of great interest to develop upper bounds on the best achievable operational intensity. The complementary problem of finding lower bounds on the minimal required data movement in a memory hierarchy has been much studied in the literature [4]–[20] (where the term I/O lower bound is often used to refer to a data movement lower bound). In this section, we provide some background on prior work on I/O lower bounds problem, followed by a presentation of new approaches we develop to address this problem (Sections III and IV). We then demonstrate use of the I/O lower bounding techniques in two scenarios of use for upper bounds on operational intensity (Sections V and VI).

The model of computation commonly used to formalize I/O lower bound problem is a computational directed acyclic graph (CDAG), where computational operations are represented as graph vertices and the flow of values between operations is captured by graph edges. The seminal work of Hong and Kung [4] was the first to develop an approach to bounding the minimum data movement in a two-level hierarchy for execution of a CDAG. We use the notation of Bilardi & Peserico [9] to describe the CDAG model used by Hong & Kung:

Definition 1 (CDAG)**.** *A computational directed acyclic graph (CDAG) is a 4-tuple $C = (I,V,E,O)$ of finite sets such that: (1) $I \subset V$ is the input set and all its vertices have no incoming edges; (2) $E \subseteq V \times V$ is the set of edges; (3) $G = (V,E)$ is a directed acyclic graph;(4) $V \setminus I$ is called the operation set and all its vertices have one or more incoming edges; (5) $O \subseteq V$ is called the output set.*

The inherent I/O complexity of a CDAG is the minimal number of I/O operations needed while optimally playing the *Red-Blue pebble game*. This game uses two kinds of pebbles: a fixed number of red pebbles that represent the small fast local memory (could represent cache, registers, etc.), and an arbitrarily large number of blue pebbles that represent the large slow main memory.

Definition 2 (Red-Blue pebble game [4])**.** *Let $C = (I,V,E,O)$ be a CDAG such that any vertex with no incoming (resp. outgoing) edge is an element of I (resp. O). Given S red pebbles and an arbitrary number of blue pebbles, with an initial blue pebble on each input vertex, a complete calculation is any sequence of steps using the following rules that results in a final configuration with blue pebbles on all output vertices:*

- **R1 (Input)** *A red pebble may be placed on any vertex that has a blue pebble (load from slow to fast memory),*
- **R2 (Output)** *A blue pebble may be placed on any vertex that has a red pebble (store from fast to slow memory),*
- **R3 (Compute)** *If all immediate predecessors of a vertex of $V \setminus I$ have red pebbles, a red pebble may be placed on (or moved to) that vertex (execution or "firing" of operation),*
- **R4 (Delete)** *A red pebble may be removed from any vertex (reuse storage).*

The number of I/O operations for any complete calculation is the total number of moves using rules R1 or R2, i.e., the total number of data movements between the fast and slow memories. The inherent I/O complexity of a CDAG is the smallest number of such I/O operations that can be achieved, among all complete calculations for that CDAG. An *optimal* calculation is a complete calculation achieving the minimum number of I/O operations.

While the red-blue pebble game provides an operational definition for the I/O complexity problem, it is generally not feasible to determine an optimal calculation on a CDAG. Hong & Kung developed a novel approach for deriving I/O lower bounds for CDAGs by relating the red-blue pebble game to a graph partitioning problem defined as follows.

Definition 3 (Hong & Kung S-partitioning of a CDAG [4])**.** *Let $C = (I,V,E,O)$ be a CDAG. An S-partitioning of C is a collection of h subsets of V such that:*

(P1) $\bigcap_{i=1}^{h} V_i = \emptyset$, *and* $\bigcup_{i=1}^{h} V_i = V$

(P2) *there is no cyclic dependence between subsets*

(P3) $\forall i, \quad \exists D \in \mathsf{Dom}(V_i) \quad$ *such that* $\quad |D| \leq S$

(P4) $\forall i, \quad |\mathsf{Min}(V_i)| \leq S$

where a dominator set of V_i, $D \in \mathsf{Dom}(V_i)$ is a set of vertices such that any path from I to a vertex in V_i contains some vertex in D; the minimum set of V_i, $\mathsf{Min}(V_i)$ is the set of vertices in V_i that have all its successors outside of V_i; and for a set A, $|A|$ is the cardinality of the set A.

Hong & Kung showed a construction for a $2S$-partition of a CDAG, corresponding to any complete calculation on that CDAG using S red pebbles, with a tight relationship between the number of vertex sets h in the $2S$-partition and the number of I/O moves q in the complete calculation, as shown in Theorem 1.The tight association theorem between any complete calculation and a corresponding $2S$-partition provides the

key Lemma 1 that serves as the basis for Hong & Kung's approach to deriving lower bounds on the I/O complexity of CDAGs typically by reasoning on the maximal number of vertices that could belong to any vertex-set in a valid $2S$-partition.

Theorem 1 (Pebble game, I/O and $2S$-partition [4])**.** *Any complete calculation of the red-blue pebble game on a CDAG using at most S red pebbles is associated with a 2S-partition of the CDAG such that* $S\,h \geq q \geq S\,(h-1)$*, where q is the number of I/O moves in the complete calculation and h is the number of subsets in the 2S-partition.*

Lemma 1 (Lower bound on I/O [4])**.** *Let* $H(2S)$ *be the minimal number of vertex sets for any valid* $2S$*-partition of a given CDAG (such that any vertex with no incoming – resp. outgoing – edge is an element of I – resp. O). Then the minimal number Q of I/O operations for any complete calculation on the CDAG is bounded by:* $Q \geq S \times (H(2S)-1)$

III. Convex Min-Cut based I/O Lower Bound

In this section, we develop an alternative I/O lower bounding approach. It is motivated from the observation that the Hong & Kung 2S-partitioning approach does not account for the internal structure of a CDAG, but essentially focuses only on the boundaries of the partitions. In contrast, we develop an approach that captures internal space requirements using the abstraction of wavefronts. The central idea is the definition of two kinds of *wavefronts* and a relation between them.

Definitions: We first present needed definitions. Given a graph $G=(V,E)$, a cut is defined as any partition of the set of vertices V into two parts $\mathcal{S}$ and $\mathcal{T}=V-\mathcal{S}$. An $s-t$ cut is defined with respect to two distinguished vertices s and t and is any $(\mathcal{S},\mathcal{T})$ cut satisfying the requirement that $s\in\mathcal{S}$ and $t\in\mathcal{T}$. Each cut defines a set of cut edges (the cut-set), i.e., the set of edges (u,v) where $u\in\mathcal{S}$ and $v\in\mathcal{T}$. The capacity of a cut is defined as the sum of the weights of the cut edges. The minimum cut problem (or min-cut) is one of finding a cut that minimizes the capacity of the cut. We define vertex u as a cut vertex with respect to an $(\mathcal{S},\mathcal{T})$ cut, as a vertex $u\in\mathcal{S}$ that has a cut edge incident on it. A related problem of interest for this paper is the *vertex min-cut* problem which is one of finding a cut that minimizes the number of cut vertices.

Given a DAG $G=(V,E)$ and some vertex $x\in V$, the set $\mathsf{Anc}(x)$ is the set of vertices from which there is a non-empty directed path to x in G ($x\notin\mathsf{Anc}(x)$); the set $\mathsf{Desc}(x)$ is the set of vertices to which there is a non-empty directed path from x in G ($x\notin\mathsf{Desc}(x)$). Using those two notions, we consider a convex cut $(\mathcal{S}_x,\mathcal{T}_x)$ associated to x as follows: $\mathcal{S}_x$ includes $x\cup\mathsf{Anc}(x)$; $\mathcal{T}_x$ includes $\mathsf{Desc}(x)$; in addition, $\mathcal{S}_x$ and $\mathcal{T}_x$ must be constructed such that there is no edge from $\mathcal{T}_x$ to $\mathcal{S}_x$. With this, the sets $\mathcal{S}_x$ and $\mathcal{T}_x$ partition the graph G into two convex partitions. We define the wavefront induced by $(\mathcal{S}_x,\mathcal{T}_x)$ to be the set of vertices in $\mathcal{S}_x$ that have at least one outgoing edge to a vertex in $\mathcal{T}_x$.

Schedule wavefront: In any complete calculation $\mathcal{P}$ of the red-blue pebble game on a CDAG C, consider the step that applies the firing rule R3 to compute vertex x. Associated with each vertex x of V, we identify a schedule wavefront $W_{\mathcal{P}}(x)$ to be the set of vertices of V that have already fired before x, but have at least one successor that has not yet been computed. $W_{\mathcal{P}}(x)$ thus represents the set of *live* values in the computation at the time that x is computed.

Graph min-cut wavefront: A convex cut of C is a partition of its vertices into two sets $\mathcal{S}$ and $\mathcal{T}$ such that there is no edge from $\mathcal{T}$ to $\mathcal{S}$. The cut-vertex-set is $\mathsf{Min}(\mathcal{S})$, i.e., vertices of $\mathcal{S}$ that have a successor in $\mathcal{T}$. We define a graph min-cut wavefront $W_G^{\min}(x)$ to be a constrained cut-vertex-set with respect to a vertex $x\in V$, as a minimum cardinality cut-vertex-set with an additional constraint that all ancestors of x fall in the partition $\mathcal{S}$ and all descendants of x fall in the partition $\mathcal{T}$. We define $w_G^{\max}=\max_{x\in V}\left(|W_G^{\min}(x)|\right)$.

The key idea behind the approach is that for any vertex x, the schedule wavefront $W_{\mathcal{P}}(x)$ must have at least as many vertices as $W_G^{\min}(x)$ and that the size of any schedule wavefront imposes a lower bound on I/O due to the limited number of red pebbles. Consider the vertex x in the Diamond DAG shown in Fig. 3. A graph min-cut wavefront $W_G^{\min}(x)$ is shown by the dashed line that includes x and five other vertices (this wavefront is not unique, but all min-cut wavefronts for x will include six vertices). The relationship

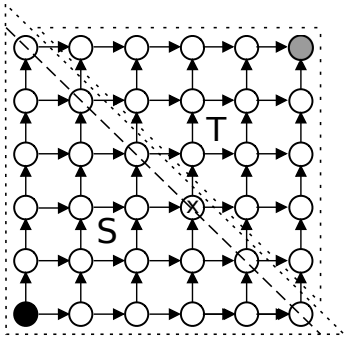

Fig. 3: Diamond DAG with constrained min-cut with respect to vertex x

between schedule wavefronts and graph min-cut wavefronts means that in any complete calculation, at the time x is fired, there must be at least six live values. If the number of red pebbles S is less than six, $6-S$ of these live vertices in the CDAG must necessarily have been spilled (no re-pebbling is permitted in the pebble game model we use), representing a corresponding minimal amount of I/O. A proof of the following lemma, and further details of the min-cut based lower bounding approach are provided in a technical report [21].

Lemma 2. *Let $C=(\emptyset,V,E,O)$ be a CDAG with no inputs. For any $x\in V$, $2\left(|W_G^{min}(x)|-S\right)\leq IO(C)$. In particular, $2\left(w_G^{max}-S\right)\leq IO(C)$.*

Tighter Bounds via Partitioning: If applied to the whole CDAG, Lemma 2 will usually lead to a very weak bound. To overcome this limitation, the idea is to decompose it into smaller sub CDAGs, and sum up their individual I/Os. The following theorem formalizes this approach and its proof is provided in a technical report [21].

Theorem 2 (Min-Cut with divide and conquer)**.** *Let $C=(I,V,E,O)$ be a CDAG. Let $V_1,\ldots,V_p$ be a (not necessarily acyclic) disjoint partitioning of V, and $C_1,\ldots,C_p$ be the induced partitioning of C ($I_i=V_i\cap I$, $E_i=E\cap V_i\times V_i$, $O_i=O\cap V_i$). Let for each i, $C_i'=(\emptyset,V_i',E_i',\emptyset)$ be the sub-DAG obtained from C_i by deleting all input and output vertices ($V_i'=V_i-I_i-O_i$, $E_i'=E_i\cap V_i'\times V_i'$, and $G_i'=(V_i',E_i')$). Then the minimum I/O of C can be bounded by:*

$$\sum_{i=1}^{p} 2\left(w_{G_i'}^{max}-S\right)+|I|+|O|\leq IO(C)$$

IV. Automated Bounds for Arbitrary CDAGs

All approaches for I/O lower bounds reported in the literature so far have relied on specific properties about the structure of the analyzed computation. In this section, we develop an automated approach to deriving I/O lower bounds for any arbitrary CDAG, regular or irregular, without any restrictions on the structure or reliance on any properties of the CDAG. It uses convex graph min-cut technique introduced in Section III for which a graph-based heuristic is detailed hereafter.

A. High-Level Meta-Heuristic

Our heuristic IOhierarchical combines both techniques using a recursive decomposition of the CDAG. The high-level principle is to (1) first run the mincut heuristic on the full CDAG. The result of the min-cut based heuristic IOmincut is stored on Q_{mincut}. (2) The CDAG is then bisected into C_1 and C_2 and IOhierarchical is run independently on each of the two sub-CDAGs. The results are stored into Q_1 and Q_2, then summed up into $Q_{rec}=Q_1+Q_2$. (3) The best of the two $max(Q_{mincut},Q_{rec})$ is returned.

The precise pseudo-code for IOhierarchical is given in Figure 4. The description of IOmincut is given in Section IV-B.

```
IOhierarchical(C,V)
Inputs:
   C: CDAG
   V: vertex (sub-)set of C
Outputs:
   Q: lower bound of IO(C) restricted to V
Q ← 0
Q_mincut ← IOmincut (C,V)
Q ← max(Q, Q_mincut)
if |V| > 2S then
      (V1, V2) ← Bisect (C,V)
      Q1 ← IOhierarchical (C, V1)
      Q2 ← IOhierarchical (C, V2)
      Q ← max(Q, Q1 + Q2)
end if
return Q
```

Fig. 4: Recursive decomposition algorithm for computing I/O bound on an arbitrary CDAG, C

The min-cut based heuristic is given a timeout proportional to the size of the graph (the implementation details for this are not reported in the pseudo-codes). If no result is found within this timeout, the corresponding procedure returns -1, and the result is not used. Note that there are circumstances under which it is clear that it is not worth bisecting the current CDAG. A simple example is when IOmincut has not been interrupted by the timeout (so the returned result is an accurate value of $\max\left(0, 2(W_G^{\max} - S)\right)$) and it returns 0. The goal of the bisection Bisect is to (almost) equally partition the DAG into two sub DAGs C_1 and C_2 such that an (almost) minimum number of edges between C_1 and C_2 are cut. Ideally this partitioning should be convex (there are edges from C_1 to C_2 but no edges from C_2 to C_1). But Theorem 3 stated below (see [21] for proof) allows to compose the results of non-convex partitioning and we thus can use standard graph bisection library that does not impose any convexity constraint.

Theorem 3 (Decomposition)**.** *Let $C = (I,V,E,O)$ be a CDAG. Let $V_1, V_2, \ldots, V_p$ be an arbitrary (non necessarily acyclic) disjoint partitioning of V ($i \neq j \Rightarrow V_i \cap V_j = \emptyset$ and $\bigcup_{1 \leq i \leq p} V_i = V$) and $C_1, C_2, \ldots, C_p$ be the induced partitioning of C ($I_i = I \cap V_i$, $E_i = E \cap V_i \times V_i$, $O_i = O \cap V_i$). Then $\sum_{1 \leq i \leq p} IO(C_i) \leq IO(C)$. In particular, if Q_i is a lower bound on the IO of C_i, then $\sum_{1 \leq i \leq p} Q_i$ is a lower bound on the I/O of C.*

B. Heuristic for Min-Cut Approximation

The min-cut based heuristic is a direct implementation of Lemma 2. As the requirement for this lemma is for the CDAG to have no input vertices, the first operation of IOmincut is to delete input vertices from V. It also removes output vertices. The number of such deleted vertices is then added to the final I/O bound

```
ConvexMinVertexCut(G', x)
Input:
  G' = (V', E'): DAG
  x: vertex about which min-cut is computed
Output:
   W: W_G'^min(x)
W ← Optimal solution to the following LP:
   ∀ v ∈ V', t_v ≥ 0, c_v ≥ 0
   minimize Σ_{v∈V'} c_v
   s.t. ∀ (v,w) ∈ E', t_w ≥ t_v
        ∀ (v,w) ∈ E', c_v ≥ t_w − t_v
        t_x = 0
        ∀ v ∈ Anc(x), t_v = 0
        ∀ v ∈ Desc(x), t_v = 1
return W
```

Fig. 5: Compute Convex vertex-min-cut

```
IOmincut(C,V)
Input:
   C: CDAG
   V: vertex (sub-)set of C
Outputs:
   Q: lower bound of IO(C) restricted to V
dI ← V ∩ I
dO ← V ∩ O
V' ← V − dI − dO
E' ← E ∩ V' × V'
W ← 0
forall x ∈ V' do
  W_x ← ConvexMinVertexCut ((V',E'), x)
  W ← max(W, W_x)
end do
Q' ← max(0, 2 × (W − S))
Q ← Q' + |dI| + |dO|
return Q
```

Fig. 6: Compute Minimum I/O cost for a CDAG using MinCut

result. For this modified graph, say $G' = (V', E')$ with no input/output vertices, the variable W that represents $W_{G'}^{max}$ is computed as the maximum value of $\mathsf{ConvexMinVertexCut}(G', x)$ over all $x \in V'$. Vertices of V' are actually enumerated in random order so that in the presence of a time-out any intermediate maximum value gives a correct lower-bound of $W_{G'}^{max}$. The corresponding algorithm is shown in Fig. 6.

As explained in Section III, the computation of $W_{G'}^{min}(x)$ (computed through the call to $\mathsf{ConvexMinVertexCut}(x)$) corresponds to: (1) a partitioning of V' into two sub-DAGs $\mathcal{S}_x$, $\mathcal{T}_x$ such that; (2) there is no edge from $\mathcal{T}_x$ to $\mathcal{S}_x$; (3) $\mathcal{S}_x$ contains x plus all the ancestors $\mathsf{Anc}(x)$ of x; (4) $\mathcal{T}_x$ contains all the descendants $\mathsf{Desc}(x)$ of x; (5) The out set of $\mathcal{S}_x$ ($\mathsf{Out}(\mathcal{S}_x)$: set of vertices of $\mathcal{S}_x$ that have a successor in $\mathcal{T}$) is of minimum size.

This ConvexMinVertexCut, detailed in Fig. 5, is formulated as a Linear Programming (LP) problem that minimizes the number of cut vertices. Each vertex, $v \in V'$ is associated with two non-negative variables, c_v and t_v. Variable c_v captures the cut vertex count, and the variable t_v determines if vertex v belongs to $\mathcal{S}_x$ or $\mathcal{T}_x$, depending on whether $t_v = 0$ or 1, respectively. The convexity is enforced by the first constraint that given an edge $(v,w) \in E'$, if v is in $\mathcal{T}_x$, then w also belongs to $\mathcal{T}_x$. The constraint, $c_v \geq t_w - t_v$, accounts for the cut vertex count whenever a vertex $v \in \mathcal{S}_x$ has its successor in $\mathcal{T}_x$. Finally, vertices $v = x \cup v' \in \mathsf{Anc}(x)$ $(resp.\ v \in \mathsf{Desc}(x))$ are restricted to the set $\mathcal{S}_x$ $(resp.\ \mathcal{T}_x)$ by setting $t_v = 0$ $(resp.\ 1)$. Since the constraint matrix of this LP formulation is totally-unimodular, this linear optimization problem always has an integral optimum. The minimization objective $(min \sum_{v \in V'} c_v)$ implicitly ensures that $c_v \leq 1$ and $t_v \leq 1$.

C. Discussion

The scalability of the automated tool is an important issue and being able to address large problems is currently a challenge. The ConvexMinVertexCut pseudo-code, as reported here, has an average-case complexity of $O(|V|^3)$, leading to an overall complexity for IOmincut of $O(|V|^4)$. In other words, the largest sub CDAG that can be treated within time credit of T will have at most $\Theta\left(\sqrt[4]{T}\right)$ vertices. In other words, given a CDAG of size $|V| = n$, a time credit of T (assuming a scalable heuristic is used to perform bisection), the CDAG will have to be decomposed in at least $\sqrt[3]{\frac{T}{n^4}}$ equal size parts. Further bisections, that may improve the quality of the bound, will not impact the overall complexity.

Several techniques could be used to lower the overall complexity. First, as W_G^{max} computed by IOmincut is independent of the pebble count, S, the IOmincut heuristic can be run just once for a given problem size, collecting the W_G^{max} of each (sub-)CDAG in a file and finally post-processing the output at a cheaper computational cost to obtain the lower bounds corresponding to any value of S. Also, the computation of W_G^{max} could be lowered to $O(n^3)$ by taking advantage of an incremental update of $W_G^{min}(x)$ (using a direct graph based implementation). It can also be lowered by taking the maximum value of $W_G^{min}(x)$ on a randomly

generated set of vertices x. Further, IOmincut can be parallelized by executing ConvexMinVertexCut for different vertices in parallel.

V. Operation Intensity Bounds for Algorithm Analysis and Optimization

In this section, we demonstrate the use of bounding analysis of operational intensity for comparing alternate algorithms for a problem. We use the example of sorting, where indeed the use of the automated tool for analysis produced unexpected results and insights that resulted in an improved implementation.

We consider two algorithms for sorting: quicksort and odd-even sort. Quicksort has an average computational complexity of $O(n \log_2 n)$ to sort n elements, while odd-even sort is has an asymptotically higher complexity of $O(n^2)$, but a lower coefficient for quadratic complexity term. So efficient sorting routines often use a combination of an $O(n \log_2 n)$ algorithm like quicksort along with a routine like odd-even sort, and a threshold value of n (usually set to a value between 16 and 20) for switching between the two algorithms. We were interested in comparing the bounds on operation intensities of the two sorting algorithms. Fig. 7 shows the results of running the analysis tool described in Sec. III on the traces generated from running quicksort and odd-even sort. We show upper bounds on operational intensity as a function of S. On the same graph, we also see the actual operational intensity for the sequence of operations executed by the code – by playing the red-blue pebble game using the code's sequential execution order for firing of CDAG vertices It may be seen that while odd-even sort has a lower operational intensity for the actually executed schedule than quicksort for the code's sequential execution order, the upper bounds from automated CDAG analysis reveal a significantly lower bound for odd-even sort than quicksort.

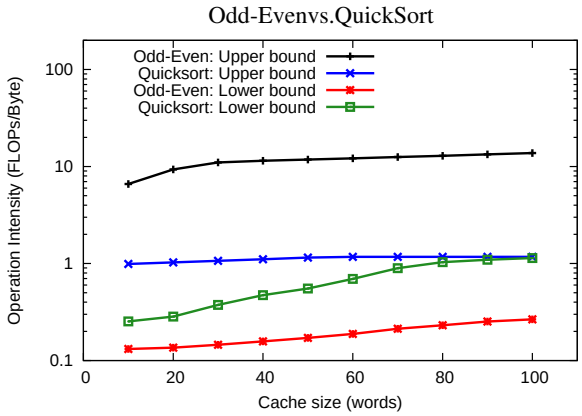

Fig. 7: Execution time for original and tiled odd-even sort relative to quick sort

Upon examining the CDAG for odd-even sort more carefully, it became apparent that it was amenable to skewing followed by loop tiling to significantly enhance data locality. In contrast, for the recursive quicksort algorithm, we could not identify any opportunity for loop transformations that would improve data locality. We implemented a register-tiled version of odd-even sort and evaluated performance for different register-tile sizes. Fig. 8 displays relative time for odd-even sort over quicksort as a function of array size (using Intel ICC, for int data type), for the standard version as well as the modified register-tiled version. For the original version, the cross-over was around 16, i.e., odd-even sort was more efficient for sorting fewer than 16 elements, while quick sort was more efficient for sorting more than 16 elements. But with the tiled version of odd-even sort, the cross-over point was much higher - around 110 elements. The experiment was run multiple times with different random inputs and average time was taken. Boundary cases like fully sorted and reverse sorted inputs were also tested and we found similar trends.

VI. Operational Intensity Bounds and Architecture Parameters

In this section, we show how the analysis of upper bounds on operational intensity of an algorithm as a function of cache size can be used to model the limits of achievable performance and energy efficiency of

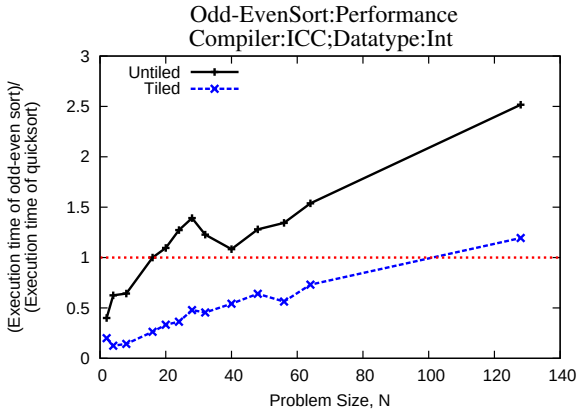

Fig. 8: Execution time for original and tiled odd-even sort relative to quick sort

algorithms. We perform two types of analysis:

- For a particular set of technology parameters, what are the upper bounds on achievable per-chip performance (GFLOP per second) for different algorithms, considering a range of possible ways of dividing the chip's area among processor cores versus cache memory?
- For a given set of technology parameters, what are the upper bounds on energy efficiency (Giga-operations per Joule) for different algorithms?

A. *Notations*

We consider a chip with a fixed total area A, where a portion $\alpha.A$ is allocated to the last level cache and the remaining $(1-\alpha).A$ area is allocated to cores. While such an analysis could be directly extended to model constraints with respect to multiple levels of cache, we only model the impact of the last level of cache (LLC) in this analysis. We also do not account for the area required for inter-connects and other needed logic. It is important to note that since our modeling is purely an upper bound analysis on performance and energy efficiency, the results obtained are of course valid under these simplifications - including more details of the architecture can allow tighter bounds, but all results presented later in this section are assertable upper limits on performance and energy efficiency for the modeled technology. For example, when the analysis shows that large-FFT (size too large to fit in LLC) performance cannot exceed 60 GFLOPs aggregate perforance for a multi-core chip built using the modeled technology, it represents an upper bound that cannot be exceeded by any possible actual implementation with those technology parameters - a more detailed model that accounts for additional levels of cache, on-chip interconnect, control logic, etc. can only serve to tighten or reduce the upper bound to a lower value than 60 GFLOPs.

We consider P and S (as defined below) to be tightly related to A. α and B_{mem} are supposed to be fixed. $F_{flops}(f)$ varies with the clock frequence f.

A	Total chip area
$\alpha.A$	Area occupied by LLC
$(1-\alpha).A$	Area occupied by cores
f	Clock frequency (GHz)
P	Number of cores
$F_{flops}(f)$	Peak performance of a single core at frequency f (GFLOPs)
B_{mem}	Peak memory bandwidth to LLC (Gbytes/sec)
S	Last level cache size in bytes

Performance: The total computation time T follows the roofline model and can be expressed as:

$$T = \max\left(\frac{W}{P.F_{flops}(f)}, \frac{Q_{mem}}{B_{mem}}\right) = \max\left(\frac{W}{P.F_{flops}(f)}, \frac{W}{B_{mem}.OI(S)}\right) \quad (1)$$

with,

W	Total number of arithmetic operations
Q_{mem}	Total number of bytes for memory accesses
T	Total computation time
$OI(S)$	Operational Intensity as a function of S

For a given application, W can be expressed in terms of problem size parameters and lower bounds on Q_{mem} can be expressed through our analysis in terms of S. As an example, for matrix mutmult(N), we have $W = 2N^3$, $Q_{mem} \geq \frac{N^3}{2\sqrt{2S}} + 3N^2$.

Energy: The total energy consumption is modeled as:

$$E = W.\varepsilon_{flop}(f) + Q_{mem}.\varepsilon_{mem} + (P.\pi_{cpu} + \pi_{cache}(S)).T \tag{2}$$

where the quantities used are defined below.

π_{cpu}	Static leakage power per core
$\pi_{cache}(S)$	Static leakage power for cache of size S
$\varepsilon_{flop}(f)$	Energy per arithmetic operation at frequency f
ε_{mem}	Energy per byte of data access from/to memory

B. Architectural Parameters

We demonstrate the utility of modeling upper bounds on operational intensity by performing an analysis of possible architectural variants for a given technology. We use architectural parameters for an enterprise Intel Xeon processor (codename *Nehalem-EX*) [22] and use the published and available statistics to estimate area, power and energy for different compute and memory units, as well as the operations.

Parameter	Value
Die Size	684 mm2
Technoloy Node	45 nm
Num of Cores	8
Num of LLC Slices	8
LLC Slice Size	3MB
LLC Size (total)	24 MB
DRAM Channels	4
Core Voltage	0.85 1.1 V
Core Max Clock	2.26 GHz
TDP	130 W
Threads per core (SMT)	2
Leakage (total)	21 W
L1/L2	32KB / 256 KB

TABLE I: Nehalem-EX processor spec.

Table I shows the physical specification for a testbed CPU.

Using the die dimensions, and by processing the die photo, we computed the area (in mm^2) for the chip-level units of interest. For this study, we model core area, which includes private L1 and L2 caches. We also model a range of values for the shared last level cache (LLC).

Table II shows the extracted dimensions for the Nehalem-EX Core and LLC. In order to explore LLC with different sizes, we used CACTI [23]. We fixed LLC design parameters based on the chip specification, varying only the size of the cache.

Table III shows modeled cache sizes and their corresponding area (from CACTI). Using the area percentage for each unit, and the reported total leakage power for the chip (i.e. 21W or 16%), we modeled the static power consumed by each core on an area-proportional basis.

In order to model dynamic power per core, we used McPAT [24], an analytical tool to model CPU pipeline and other structures. To estimate core parameters, we extended the available Xeon model to allow for more

Unit	Width (mm)	Height (mm)	Area (mm^2)	Area (%)
Die	31.9	21.4	684	100
Core	7.2	3.7	26.7	3.0
LLC Slice	7.8	3.7	29.3	4.3

TABLE II: Nehalem-EX die dimensions.

Size (KB)	H (mm)	W (mm)	Area (mm^2)
4	0.341	0.912	0.311
8	0.368	0.912	0.336
16	0.472	0.966	0.456
32	0.580	1.010	0.586
64	0.815	1.015	0.827
128	0.848	2.032	1.723
256	0.870	4.060	3.532
384	1.096	4.066	4.456
512	2.798	2.778	7.773
1024	4.632	2.800	12.970
2048	5.080	5.138	26.101
3072	7.446	5.353	39.858
4096	5.859	9.173	53.745
8192	9.518	9.791	93.191

TABLE III: Modeled LLC Slices using CACTI [23]

number cores. All modifications were based on real parameters published by Intel [22]. In order to model the impact on energy efficiency of voltage/frequency scaling, we extracted the maximum and minimum operating voltage range for the processor, and the corresponding frequencies at which the processor can operate. Using those V/F pairs, different "power states" were modeled for the processor using McPAT [24]

Voltage (V)	Clock (MHz)	Peak Chip Dyn (W)	Peak Core Dyn (W)
1.10	2260	111.14	102.68
1.05	2060	95.94	87.49
1.00	1860	81.82	73.35
0.95	1660	62.23	60.78
0.90	1460	58.12	49.66
0.85	1260	48.39	39.93

TABLE IV: Nehalem-EX: effect of changing voltage and frequency on core/chip power

Table IV shows how changing voltage and frequency effects total chip and core power.

Below, we summarize the parameters for the testbed architecture used for the modeling in the following subsections: $F_{flops}(f) = 9.04$ GFLOPS @ 2.26 GHz, $B_{mem} = 40$ GB/s, $A = 684$ mm^2, $A_{core} = 26.738$ mm^2, $\varepsilon_{flop}(f) = 1.3$ nJ/flop, $\varepsilon_{mem} = 0.63$ nJ/byte, $\pi_{cpu} = 0.819$ W, and $\pi_{cache}(S) \approx 0.01$ W for cache sizes of 4, 8 and 16 KB.

C. Bounds on Maximum Performance

First, we consider the implications of upper bounds on operational intensity of an algorithm on the maximal performance (operations per second) achievable on a chip. As described above, we assume that the chip area can be partitioned as desired among processor cores or cache. The size of the last level cache

was varied from a tiny 4Kbyte size (representing a fraction of under 0.1% of the chip of approximately 700 mm^2 area) to 64 Mbytes (filling almost the entire chip area). The general characteristic is that the operational intensity of an algorithm is some monotonic non-decreasing function of the cache capacity. However, the shape of this function can be very different for different algorithms. For demonstration purposes, we analyze three benchmarks: Matrix multiplication, FFT, and a Conjugate Gradient (CG) iterative linear system solver. In all cases, the problem sizes are set to be much larger than last level cache. Using analysis techniques described earlier, we can derive the following upper bounds on operational intensity for the algorithms, as a function of cache capacity (S words):

- Matrix multiplication: $4\sqrt{2S}$
- FFT: $log(S)/2$
- CG: 10/3 (it is independent of cache capacity)

Derivations are provided in the appendix.

Fig. 9 shows the variation of upper bounds on operational intensity as a function of fractional chip real estate used for cache. It may be seen that the trends are very different for the three benchmarks. Matrix multiplication shows a significant increase in OI as the fraction of chip area occupied by cache is increased (the plot is logarithmic on the y axis). FFT shows a very mild rise in OI as the cache fraction is increased, and the operational intensity is low – between 1 and 2 over the entire range. CG has a flat and very low value of OI, irrespective of the amount of cache deployed.

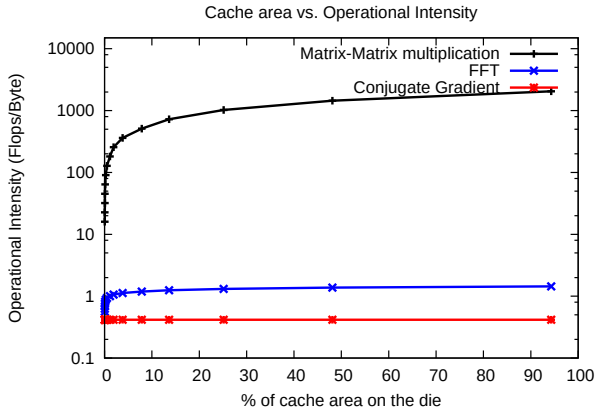

Fig. 9: Operational intensity upper bounds

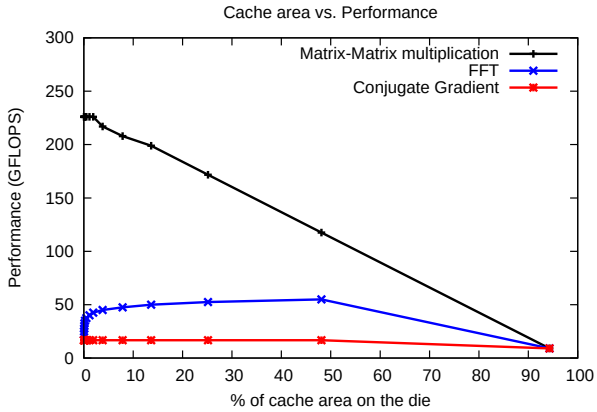

Fig. 10: Upper bounds on performance

Fig. 10 presents upper bounds on performance as a function of fractional chip real estate used for cache. Of the three benchmarks, only matrix multiplication has the potential to achieve good performance on large problems. Both FFT and CG are very bandwidth limited, with extremely low aggregate performance bounds, irrespective of how the chip is configured with respect to cache versus cores. It is interesting to note that the initial trends as a function of cache capacity are in opposite directions for FFT and matrix multiplication. With matrix multiplication, the operational intensity grows very rapidly as a function of cache size - $O(\sqrt{S})$, and thus quickly becomes compute-bound even for a tiny 16Kbyte last level cache. Further increases in cache size cannot reduce the lower bound on total time, which is modeled as the larger of the bounds on compute time and data movement time. However, there is a decrease in the fractional chip area that can be used for cores. Hence there is a decrease in the upper bounds on achievable performance as the fractional chip area of the cache is increased. In contrast, FFT is highly memory-bandwidth limited, and using a large number of cores on the chip is of no use. Increasing the size of the last level cache can improves operational intensity and thus improve performance, but the rate of growth is very slow - $O(log(S))$. So the upper limits on achievable performance grows slowly with increasing cache size although the number of cores on chip decrease. When a very high fractional chip area (over 50%) is used for the cache, performance then decreases as FFT becomes compute bound with only a few cores on the chip.

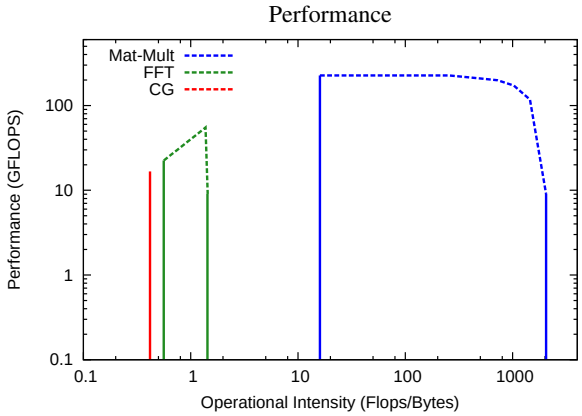

Fig. 11: Upper bounds on performance

Fig. 11 shows the range of operational intensities and corresponding performance roofline for each of the three benchmarks on a performance roofline plot. A standard roofline plot has one or more sloping lines representing bandwidths for different levels of the memory hierarchy, and one or more extended horizontal lines corresponding to the peak computational performance of different resources. For multicore systems, different horizontal lines could be used to represent the peak performance with use of different numbers of cores. The plot shown in Fig. 11 represents a consolidated analysis that takes into account the interdependence between the size of LLC and the maximum number of cores - the larger the LLC, the less the remaining area on the chip for cores. The analysis was performed as follows. For a given LLC size, the maximum number of cores is found. The maximal operational intensity for a given algorithm is computed for the LLC size, and the intersection of that vertical OI line on the roofline plot is located. It is possible for that intersection to be on the sloping memory-bandwidth roofline or the multicore roofline for the maximum number of processors possible for the chosen LLC size. This analysis is repeated for the range of LLC sizes. The results for the three benchmarks show disjoint ranges of operational intensities.

CG has no variation in OI as a function of S and is always in the bandwidth-limited region. Hence it is represented by a single line (in red) in the roofline plot, FFT straddles the two regions, being fully bandwidth bound for low values of cache size and getting into the compute bound region for high cache size. But as this happens at very large LLC values, the peak processor performance decreases with decreasing number of cores on the chip, so that the performance upper bound drops for increasing values of OI. Mat-mult has a range (as the size of the last level cache is varied) that is always in the compute-bound region of the roofline. But as the LLC size is increased, the number of cores on chip must decrease, and as with FFT, the performance upper bound drops at very high values of OI.

The analysis shows that two currently widely used algorithms, FFT and CG, will not be well suited for solution of very large problems relative to the cache capacity, unless the bandwidth between main memory and cache is substantially increased relative to representative parameters of current systems.

D. Bounds on Maximum Energy Efficiency

An important metric is energy efficiency, defined as the ratio of number of executed operations to the energy expended in the execution. The upper bounds on operational intensity can also be used to bound the maximal achievable energy efficiency. The total energy of execution is modeled as the sum of energy for performing the operations, the energy for data movement from DRAM access, and an additional component for the static leakage energy in the cores and cache. Fig. 12, Fig. 13, and Fig. 14 show the variation in the upper bounds on energy efficiency for FFT, matrix multiplication, and CG, respectively, as a function of: 1) number of cores used, 2) the voltage and clock frequency used, and 3) the capacity of last-level cache.

The horizontal axis marks three different clock frequencies (2260 Mhz, 1860 Mhz, and 1460 Mhz), representing voltage/frequency scaling, and for each of the frequencies, five choices of processor count (1,

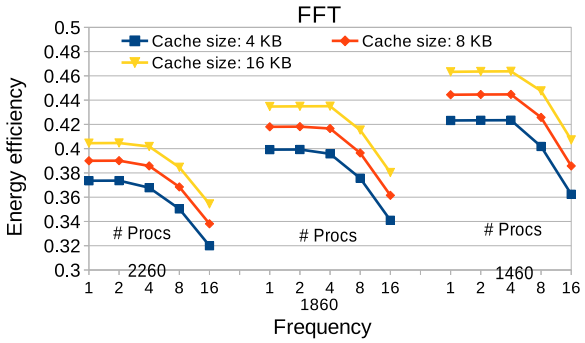

Fig. 12: Energy Efficiency: FFT

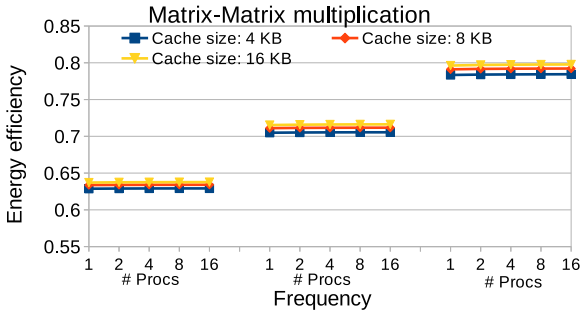

Fig. 13: Energy Efficiency: Matrix Multiplication

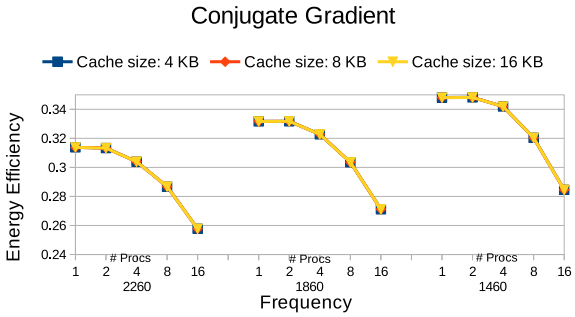

Fig. 14: Energy Efficiency: Conjugate Gradient

2, 4, 8, 16). Different curves correspond to different capacities of the last level cache. The overall trends are as follows:

1) For all three benchmarks, the groups of clustered lines move upwards as we go from left to right on the charts, representing a decrease in the voltage/frequency. For a fixed number of cores and LLC capacity, lower frequencies lead to higher bounds on attainable energy efficiency. This is because there is a nonlinear decrease in the core's energy per operation (energy is proportional to $V^2 f$, and voltage and frequency are linearly related, so that energy is proportional to f^3) as voltage/frequency are decreased. There is also an increase in the total static leakage energy since the computation will take longer to complete, but there is an overall reduction in the lower bounds for energy, or an increase in the upper bounds on energy efficiency.
2) Increasing the number of cores (with fixed frequency and LLC) is detrimental to energy efficiency, especially for bandwidth-bound computations. This is seen clearly for FFT and CG, with each of the

lines curving downwards as the number of processors is increased. This effect is mainly due to the increased static energy for the active cores. While additional cores can divide the parallel work among themselves, the computation rate is limited by the rate at which data is delivered from memory to cache, so that there is no reduction in the lower bound for execution time. For matrix multiplication, the curves are relatively flat since the computation is compute-bound. Using more cores enables the work to be done faster, and there is no increase in the total static leakage energy aggregated over all cores: using twice as many cores halves the lower bound on execution time and doubles the static leakage power of the cores.

3) Increasing LLC capacity has two complementary effects: i) potential for improving energy efficiency by increasing operational intensity due to the larger cache, but 2) decreased energy efficiency due to higher static leakage energy from the larger cache.
There is no useful benefit for CG since its OI is independent of cache capacity. At larger cache sizes, there is a detrimental effect (not seen in the charts since the cache sizes used were quite small). For FFT, the benefits from improved OI clearly outweigh the increased static leakage energy. For matrix multiplication, although OI increases with cache size, it is already so high at the smallest cache size that the incremental benefits in reducing data transfer energy from main memory are very small. Hence the curves representing different cache sizes for fixed frequency do not have much separation.

We next characterize the maximal possible energy efficiency for the three benchmarks, if have the flexibility to choose the number of cores to be turned on, the amount of cache area to be used, and voltage/frequency at which the cores are to be run. From Equations (1) and (2), we have the energy efficiency,

$$E_{eff} = \frac{W}{E} = \frac{1}{\varepsilon_{flop}(f)} + \frac{OI(S)}{\varepsilon_{mem}} + \frac{\min\left(P.F_{flops}(f), B_{mem}.OI(S)\right)}{P.\pi_{cpu} + \pi_{cache}(S)} \tag{3}$$

Depending on the upper bound on the operational intensity of the algorithms, we analyze different cases below and determine what is the best configuration for the architecture to obtain maximum energy efficiency. Finally, we show the

Case I: *The algorithm is completely bandwidth bound.*
This corresponds to the case where the maximal $OI(S)$ of the algorithm for a given cache size S, is too low that it is bandwidth bound even on a single core at its lowest frequency. We can see from the performance roofline viewpoint that this leads to the condition $B_{mem}.OI(S) < F_{flops}(f)$. With increasing frequency, $\varepsilon_{flop}(f)$ increases and thus the energy efficiency deteriorates. Hence, highest energy efficiency is achieved when $P = 1$ and f is set at its minimum, and, Equation (3) reduces to $E_{eff} = \frac{1}{\varepsilon_{flop}(f_{min})} + \frac{OI(S)}{\varepsilon_{mem}} + \frac{B_{mem}.OI(S)}{\pi_{cpu} + \pi_{cache}(S)}$, where, f_{min} is the minumum allowable frequency for the architecture.

Case II: *The algorithm is compute bound with* p *or less cores and all frequencies.*
From Equation (3), we can see that at any given frequency f, increasing P improves the E_{eff}. Hence, $P = \mathsf{p}$ is the best choice irrespective of the frequency. Also, with increasing f, $F_{flops}(f)$ increases linearly, while $\varepsilon_{flop}(f)$ increases super-linearly. Thus, the optimal energy efficiency depends on the machine parameters, and the energy efficiency is calculated as For a fixed value of p, the optimal energy efficiency is dependent on the machine parameters. like $\pi_{cache}(S)$ and $\varepsilon_{flop}(f)$. The maximal energy efficiency obtained,
$E_{eff} = \max_{f \in [f_{min}, f_{max}]} \left(\frac{1}{\varepsilon_{flop}(f)} + \frac{OI(S)}{\varepsilon_{mem}} + \frac{\mathsf{p}.F_{flops}(f)}{\mathsf{p}.\pi_{cpu} + \pi_{cache}(S)} \right)$, where, f_{min} and f_{max} are the minumum and maximum allowable frequencies for the architecture, respectively.

Case III: *The algorithm is compute bound with* p *cores at lower frequencies and becomes bandwidth bound with* p *cores at higher frequencies.*
Let f_{cutoff} be the frequency where the algorithm transitions from compute bound to memory bound region. For the region where $f \geq f_{cutoff}$, from case I, we know that once the algorithm becomes bandwidth bound, increasing the frequency further has detrimental effect on the energy efficiency. Hence, the best energy

efficiency is achieved when $f = f_{cutoff}$, where, $\mathsf{p}.F_{flops}(f_{cutoff}) = B_{mem}.OI(S)$. When $f < f_{cutoff}$, analysis in case II showed that in the compute bound region when the number of cores, p, is held constant, optimal frequency depends on the machine parameters. Also, we have, $\mathsf{p}.F_{flops}(f) < B_{mem}.OI(S)$. Hence, $E_{eff} = \max_{f \in [f_{min}, f_{cutoff}]} \left(\frac{1}{\varepsilon_{flop}(f)} + \frac{OI(S)}{\varepsilon_{mem}} + \frac{\mathsf{p}.F_{flops}(f)}{\mathsf{p}.\pi_{cpu} + \pi_{cache}(S)} \right)$.

Case IV: *The algorithm is compute bound at all frequencies with* p *cores, and becomes bandwidth bound with* $\mathsf{q} = \mathsf{p} + 1$ *or higher number of cores.*

This case gives rise to two scenarios: (1) Performance at higher frequencies ($f_p \in [f_1, f_{max}]$) with p cores overlaps with the performance at lower frequencies ($f_q \in [f_{min}, f_2]$) with q cores, and hence, the algorithm becomes bandwidth bound at frequencies $f > f_2$ and q cores. (2) There is a gap between maximum performance achievable with p cores and minumum performance achieved with q cores, i.e., $\left(\mathsf{q}.F_{flops}(f_{min}) - \mathsf{p}.F_{flops}(f_{max})\right) > 0$. In both these scenarios, the maximum achievable energy efficiency depends on the machine parameters.

Similar to the performance roofline in Fig. 11, energy roofline has been plotted for the three benchmarks in Fig. 15 with a range of operational intensities corresponding to the range of LLC sizes for our testbed architecture.

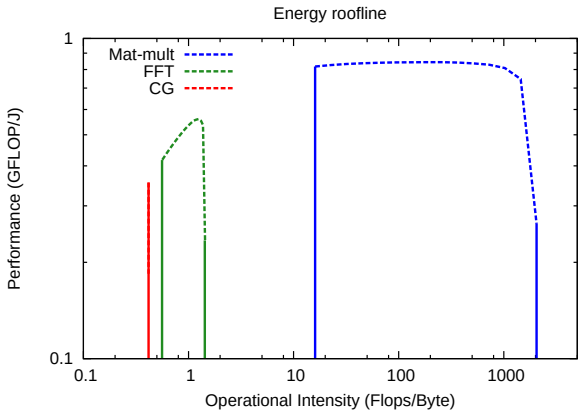

Fig. 15: Upper bounds on energy efficiency

Fig. 15 shows a similar trend for the energy efficiency as that of the performance in Sec. VI-C. Mat-mult is always compute bound and hence falls under case II for all values of S. Based on the machine parameters for the testbed, the optimal energy efficiency is achieved for the lowest frequency value, in general. But, as the number of active cores approaches one (as the size of LLC increases), the higher frequency value provides maximal energy efficiency. On the other hand, CG becomes bandwidth bound as the number of active cores for the given LLC size exceeds 3. Hence, CG starts at case IV with $P = 3$, and as the LLC size increases (and the number of allowable cores goes below 3), it enters into case II. CG has the best upper bound on energy efficiency of 0.35 GFLOP/J for the cache size of 4 KB with 3 cores @ 1.26 GHz. The upper bound on energy efficiency for FFT initially increases with the increasing cache size and then starts decreasing for the similar reasons as that of the performance upper bounds. FFT achieves its maximal energy efficiency for the cache size of 8 MB at $P = 10$ and $f = 1.26$ GHz.

VII. Related Work

Williams et al. [1] developed the roofline model that attempts to analyze bottlenecks in performance of an architecture due to memory bandwidth limits. Choi et al. [2], [3] developed the energy version of the roofline model. Czechowski et al. [25] have developed balance principles for algorithm-architecture co-design. These models characterize algorithms using operational intensity, which however is not a fixed quantity for an algorithm-architecture combination. Our work complements the above efforts - we are not

aware of any other efforts that have developed methods for characterizing upper bounds on operational intensities of algorithms.

Several efforts have focused on developing I/O lower bounds, which is equivalent to the problem of finding upper bounds on operational intensity. Hong & Kung provided the first characterization of the I/O complexity problem using the red/blue pebble game and the equivalence to 2S-partitioning of CDAGs [4]. Several works followed Hong & Kung's work on I/O complexity in deriving lower bounds on data accesses [5]–[20]. Aggarwal et al. provided several lower bounds for sorting algorithms [5]. Demmel et al. have developed lower bounds as well as optimal algorithms for several linear algebra computations including QR and LU decomposition and all-pairs shortest paths problem [15]–[17], [19].

Extending the scope of the Hong & Kung model to more complex memory hierarchies has also been the subject of research. Savage provided an extension together with results for some classes of computations that were considered by Hong & Kung, providing optimal lower bounds for I/O with memory hierarchies [10]. Valiant proposed a hierarchical computational model [14] that offers the possibility to reason in an arbitrarily complex parameterized memory hierarchy model. While we use a single-level memory model in this paper, the work can be extended in a straight forward manner to model multi-level memory hierarchies.

The use of Hong & Kung's model has required manual algorithm-specific reasoning to find *S*-partitions, even for regular graphs. Savage's [10] *S*-span model also requires problem-specific insights. In addition to these works, other approaches [5], [9], [14], [15], [17] also require problem-specific insights to develop bounds. In contrast, we have developed an approach that can be used to develop I/O lower bounds for an arbitrary CDAG without any problem-specific reasoning. We note here that very recent work from U.C. Berkeley [18] has developed a very novel approach to developing parametric I/O lower bounds that does not require problem-specific reasoning. The approach is applicable/effective for a class of nested loop computations but is either inapplicable or produces weak lower bounds for other computations (e.g., stencil computations, FFT, etc.).

VIII. Conclusion

The roofline model is very useful in depicting bounds on time efficiency (operations per second) and energy efficiency (operations per Joule) as a function of operation intensity, the ratio of arithmetic operations per byte of data moved from/to memory. While operational intensity can be measured for a given implementation of an algorithm, it is not a fixed quantity, but depends on cache capacity, and will also be affected by dependence-preserving loop transformations.

In this paper, we address the problem of developing upper bounds on the operational intensity of computations as a function of cache capacity. We develop techniques to bound the maximum possible operational intensity of a computation as a function of cache size. This bound applies to all possible valid schedules for the operations constituting the computation, i.e., the bound is a fundamental upper limit for the operational intensity, irrespective of any optimization/transformation like loop tiling, fusion etc. In addition, we demonstrate the use of bounds on operational intensity on algorithm-architecture co-design, by modeling (i) the maximal achievable performance for different algorithms (operations per second) for a given VLSI technology, considering different fractional chip area being allocated to cache versus cores; and (ii) the maximal achievable energy efficiency (operations per Joule) for different algorithms on a given VLSI technology, considering variation in number of cores, cache capacity, and voltage/frequency scaling.

Appendix

Operation Intensities of various algorithms

A. Matrix-Matrix multiplication

The standard matrix-matrix multiplication of size $n \times n$ has an operation count, $W = 2n^3$. Irony et al. [7] showed that the I/O lower bound Q for matmult satisfies, $Q \geq \frac{n^3}{2\sqrt{2S}}$, where S is size of LLC in words. Hence, the operational intensity OI satisfies, $OI \leq \frac{2n^3}{n^3/(2\sqrt{2S})} = 4\sqrt{2S}$.

B. FFT

The n-point FFT (of height $\log(2n)$) has an operation count of $n\log(2n)$. The following theorem derives the I/O lower bound for FFT using the mincut approach discussed in Sec. III.

Theorem 4. *For the n-point FFT graph, the minimum I/O cost, Q, asymptotically satisfies*

$$Q \geq \frac{2n\log(n)}{\log(S)}$$

where S is the number of red pebbles

Proof: Consider a DAG for an FFT of size m. Consider a pebble game instance $\mathcal{P}$ with minimum I/O and the time-stamp at which the first output vertex (i.e. vertex with no successors) o is fired by this schedule. Let $\mathcal{S}$ be the vertices already fired strictly before o, and $\mathcal{T}$ the others. As o is an output vertex, $\mathcal{S}$ contains all the m input vertices. By construction, $\mathcal{T}$ contains all the m output vertices. Hence, the corresponding wavefront, $|W_{\mathcal{P}}(o)| \geq m$.

Now, a DAG for a n-point FFT (of height $\log(2n)$) can be decomposed into disjoint sub-DAGs corresponding to m-points FFTs (and of height $\log(2m)$). This gives us $\lfloor n/m \rfloor \times \lfloor log(2n)/log(2m) \rfloor$ full sub-DAGs. From Lemma 2, the I/O cost of each sub-FFT (by counting a full spill for each input node) is at least $2\times(m-S)$. If we consider $m = S\log(S)$,

$$\begin{aligned} Q &\geq \left\lfloor \frac{n}{S\log(S)} \right\rfloor \times \left\lfloor \frac{\log(2n)}{\log(2S\log(S))} \right\rfloor \times 2\,(S\log(S) - S) \\ &\approx \frac{n\log(n)}{S\log^2(S)} \times 2(S\log(S)) \\ &\approx \frac{2n\log(n)}{\log(S)} \end{aligned}$$

□

Finally, from the operation count and the I/O complexity for FFT, we obtain the upper bound on the operational intensity, $OI \leq \frac{n\log(2n)}{(2n\log(n))/(\log(S))} \approx \frac{\log(S)}{2}$.

C. Conjugate Gradient

Conjugate Gradient (CG) is an iterative method for solving sparse system of linear equations that generally arise from discretization of partial differential equations (PDEs). We consider the use of CG in solving a sparse linear system arising from the discretization of the PDE for the heat equation, described below.

Consider the heat flow on a long thin bar of unit length, of uniform material and insulated, so that the heat can enter and exit only at the boundaries (refer Fig. 16(a)). Let $u(x,t)$ represent the temperature at position $0 \leq x \leq 1$, and time $t \geq 0$. The objective is to determine the change in temperature over time ($u(x,t)$). The governing *heat equation* that describes this distribution of heat is given by the PDE:

$$\frac{du(x,t)}{dt} = \alpha \times \frac{d^2u(x,t)}{dx^2}$$

where, α is the thermal diffusivity of the bar. (For mathematical treatment, it is sufficient to consider $\alpha = 1$).

Since the problem is continuous, to numerically solve the heat equation, it needs to be *discretized* (through *finite difference* approximation) to reduce it to a finite problem. In the discretized problem, the values of $u(x,t)$ are only computed at discrete points at regular intervals of the bar, called the *computational grid* or *mesh*. The state variables at these grid points are given by $u(x(i),t(m))$, where $x(i) = i\times h,\ 0 \leq i \leq n+1 = 1/h$ and $t(m) = m \times k$; h and k are the *grid spacing* and *timestep*, respectively. Fig. 16(b) shows an example grid obtained by discretizing the one-dimensional bar.

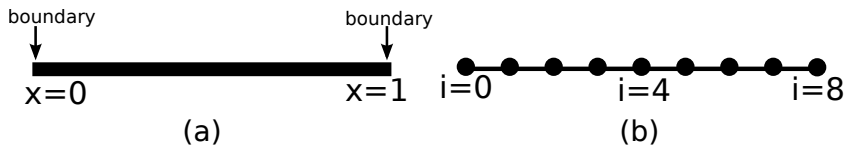

Fig. 16: One-dimensional heat flow problem

The governing equation, after discretization, yields the following equation at grid point i and timestamp $m+1$.

$$\frac{-a}{2} \times U(i-1,m+1) + (1+a) \times U(i,m+1) - \frac{a}{2} \times U(i+1,m+1) = \frac{a}{2} \times U(i-1,m) + (1-a) \times U(i,m) + \frac{a}{2} \times U(i+1,m)$$

where, $U(p,q) = u(x(p),t(q))$ and $a = k/h^2$. Hence, the solution to the problem involves solving a linear system of $n-1$ equations at each timestamp till convergence. Each timestamp $m+1$ is dependant on values of the previous timestamp m.

This linear system can be represented in tridiagonal matrix form as follows:

$$\begin{pmatrix} 1+a & -\frac{a}{2} & & & & \\ -\frac{a}{2} & 1+a & -\frac{a}{2} & & & \\ & -\frac{a}{2} & 1+a & -\frac{a}{2} & & \\ & & \ddots & \ddots & \ddots & \\ & & & -\frac{a}{2} & 1+a & -\frac{a}{2} \\ & & & & -\frac{a}{2} & 1+a \end{pmatrix} \times \begin{pmatrix} U(1,m+1) \\ U(2,m+1) \\ U(3,m+1) \\ \vdots \\ U(n-1,m+1) \\ U(n,m+1) \end{pmatrix} = \begin{pmatrix} b(1,m) \\ b(2,m) \\ b(3,m) \\ \vdots \\ b(n-1,m) \\ b(n,m) \end{pmatrix}$$

where, $b(i,m)$ represents the right-hand side of the i-th equation at timestamp $m+1$. In general, for a d-dimensional problem, the coefficient matrix is of size n^d-by-n^d, while the vectors are of size n^d. In practice, the elements of the matrix are not explicitly stored. Instead, their values are directly embedded in the program as constants thus eliminating the space requirement and the associated I/O cost for the matrix.

Each iteration of CG involves one sparse matrix-vector product, three vector updates, and three vector dot-products. The complete pseudocode is shown in Fig. 17.

```
1  x is the initial guess
2  p ← r ← b − Ax
3  do
4     v ← Ap                                   //SpMV
5     b ← (r.r)                                //Dot−prod
6     a ← b/(p.v)                              //Dot−prod
7     x ← x + ap                               //AXPY
8     r ← r − av                               //AXPY
9     g ← (r.r)/b                              //Dot−prod
10    p ← r + gp                               //AXPY
11 until ((r.r) is small)
```

Fig. 17: Classical Conjugate Gradient method

Operation count: The vector dot-product at line 6 requires $2n^d$ operations. The computation of $(\mathbf{r}.\mathbf{r})$, at lines 9, 11 and 5, requires a single vector dot-product of operation count $2n^d$. The vector update operations at lines 7, 8 and 10 have operation count of $2n^d$ each. The SpMV operation at line 4 is a stencil computation that requires $2.(2d+1).n^d$ operations. This provides a total operation count of $20n^2T$ for a two-dimensional problem.

Before we derive the I/O lower bound for CG, we state the following theorem that is useful to obtain tighter bound for CG.

Theorem 5 (Vertex Splitting)**.** *Let $C = (I, V \cup dV, E, O)$ be a CDAG. Let $C' = (I, V \cup dV' \cup dV'', E', O)$ be the CDAG induced by splitting the vertices in dV of C. Then, $IO(C)$ can be related to $IO(C')$ as follows: $IO(C') = IO(C)$*

Proof: Consider a complete calculation $\mathcal{P}$ for C, of cost $IO(C)$. We will build a complete calculation $\mathcal{P}'$ for C', of cost $IO(C)$. This will prove that $IO(C') \leq IO(C)$. We build $\mathcal{P}'$ from $\mathcal{P}$ as follows: (1) for any vertex $v \in dV$, the transition $R3$ involving v in $\mathcal{P}$ is suffixed by a transition $R3$; (2) any other transition in $\mathcal{P}$ is reported as is in $\mathcal{P}'$.

Now, consider the complete calculation $\mathcal{P}'$ for C'. We will build a complete calculation $\mathcal{P}$ for C of cost $IO(C')$. This will prove that $IO(C) \leq IO(C')$. We build $\mathcal{P}$ from $\mathcal{P}'$ as follows: (1) any transition of any vertex in V in $\mathcal{P}'$ is reported as is in $\mathcal{P}$; (2) any transition of a vertex in dV' in $\mathcal{P}'$ is reported as is in $\mathcal{P}$; (3) any transition $R3$ in $\mathcal{P}'$ on a vertex $v'' \in dV''$ is ignored; (4) any transition other than $R3$ on a vertex $v'' \in dV''$ in $\mathcal{P}'$ is reported as is in $\mathcal{P}$. □

We now provide the I/O lower bound for CG using the min-cut approach.

Theorem 6 (I/O Lower Bound for CG)**.** *For a d-dimensional grid of size n^d, the minimum I/O cost to solve the corresponding linear system using CG, Q, satisfies $Q \geq 6n^d T$, when $n^d \gg S$; here, T represents the number of outer loop iterations.*

We provide a proof sketch here. Formal proof is provided below. Let C be the CDAG for CG. C is decomposed into T sub-CDAGs, each corresponding to a single iteration of the loop. Each of these sub-CDAGs are subdivided into two sub-CDAGs by splitting the vertices corresponding to computations of elements of **r** in line 8 and removing the split edges (Theorem 5).

Consider the dot-product computation at line 6. Let x be the vertex corresponding to the computation of a. Vectors **p** and **v**, that are used as operands for computing a are needed later, at lines 7 and 8, along with scalar a. Hence, any graph min-cut wavefront induced by x should be of size at least $2n^d$.

Similarly, consider line 9. Let y be the vertex corresponding to computation of g. Vector **r**, which is an operand for computing g, is used later in line 10, along with g. Thus, any graph min-cut wavefront induced by y should be of size at least n^d.

By summing up the individual lower bounds of all the sub-CDAGs and applying theorem 3, we have, $Q \geq T \times 2(3n^d - 2S) \approx 6n^d T$, when $n^d \gg S$. □

Proof: Let C be the CDAG for CG. Consider the vertices V_p corresponding to computation of elements of **p** (line 10) at all the iterations. By applying Theorem 5 split $v \in V_p$ into v' and v'', and remove the edge between v' and v''. This provides us T sub-CDAGs, C^1, $C^2, \cdots$, C^T, each corresponding to a single iteration of the loop.

Subdivide $C^i, 1 \leq i \leq T$, into two sub-CDAGs, $C^{i,1}$ and $C^{i,2}$, by splitting $v \in V_r^i$ – vertices corresponding to computation of elements of **r** (line 8) in iteration i – into v' and v'', and removing the edges between v' and v''.

Let $\mathfrak{v}_a^{i,1}$ be the vertex in $C^{i,1}$ corresponding to computation of scalar a at line 6. The $2n^d$ vertices in $\mathsf{Anc}(\mathfrak{v}_a^{i,1})$, corresponding to elements of **p** and **v** computed in iterations $i-1$ and i, respectively, have edges to the vertices in $\mathsf{Desc}(\mathfrak{v}_a^{i,1})$ (lines 7 and 8), i.e., $|W_{C^{i,1}}^{\min}(\mathfrak{v}_a^{i,1})| = 2n^d$. Hence, from lemma 2, we have that $Q_{V^{i,1}}$, the minimum I/O restricted to $C^{i,1}$, is bounded by $2\left(2n^d - S\right)$.

Let $\mathfrak{v}_g^{i,2}$ be the vertex in $C^{i,2}$ corresponding to scalar g at line 9. The n^d vertices in $\mathsf{Anc}(\mathfrak{v}_g^{i,2})$, corresponding to elements of **r** computed in iteration i, have edges to the vertices in $\mathsf{Desc}(\mathfrak{v}_g^{i,2})$ (line 10), i.e., $|W_{C^{i,2}}^{\min}(\mathfrak{v}_g^{i,2})| = n^d$. Hence, $Q_{V^{i,2}}$ is bounded by $2\left(n^d - S\right)$.

From theorem 2, by summing up the IO cost of sub-CDAGs, we get, minimum I/O cost $Q \geq T \times 2(3n^d - 2S)$. When $n^d \gg S$, we have, $Q \geq 6n^d T$. □

Hence, from Theorem 6 and the operation count for CG, we obtain the upper bound on the operational intensity, $OI \leq 20/6 = 10/3$.

REFERENCES

[1] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009.

[2] J. W. Choi, D. Bedard, R. Fowler, and R. Vuduc, "A roofline model of energy," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '13, 2013, pp. 661–672.

[3] J. W. Choi, M. Dukhan, X. Liu, and R. Vuduc, "Algorithmic time, energy, and power on candidate hpc compute building blocks," in *Proceedings of the 2014 IEEE 28th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '14, 2014, pp. 1–11.

[4] J.-W. Hong and H. T. Kung, "I/O complexity: The red-blue pebble game," in *Proc. of the 13th annual ACM sympo. on Theory of computing (STOC'81)*. ACM, 1981, pp. 326–333.

[5] A. Aggarwal and J. S. Vitter, "The input/output complexity of sorting and related problems," *Commun. ACM*, vol. 31, pp. 1116–1127, 1988.

[6] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir, "A model for hierarchical memory," in *19th STOC*, 1987, pp. 305–314.

[7] D. Irony, S. Toledo, and A. Tiskin, "Communication lower bounds for distributed-memory matrix multiplication," *J. Parallel Distrib. Comput.*, vol. 64, no. 9, pp. 1017–1026, 2004.

[8] G. Bilardi, A. Pietracaprina, and P. D'Alberto, "On the space and access complexity of computation dags," in *Graph-Theoretic Concepts in Computer Science*, ser. LNCS, 2000, pp. 81–92.

[9] G. Bilardi and E. Peserico, "A characterization of temporal locality and its portability across memory hierarchies," *Automata, Languages and Programming*, pp. 128–139, 2001.

[10] J. Savage, "Extending the Hong-Kung model to memory hierarchies," in *Computing and Combinatorics*, ser. LNCS, 1995, vol. 959, pp. 270–281.

[11] J. E. Savage, *Models of computation*. Addison-Wesley, 1998.

[12] D. Ranjan, J. Savage, and M. Zubair, "Strong I/O lower bounds for binomial and FFT computation graphs," in *Computing and Combinatorics*, ser. LNCS. Springer, 2011, vol. 6842, pp. 134–145.

[13] D. Ranjan, J. E. Savage, and M. Zubair, "Upper and lower I/O bounds for pebbling r-pyramids," *J. Discrete Algorithms*, vol. 14, pp. 2–12, 2012.

[14] L. G. Valiant, "A bridging model for multi-core computing," *J. Comput. Syst. Sci.*, vol. 77, pp. 154–166, Jan. 2011.

[15] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-optimal parallel and sequential QR and LU factorizations," *SIAM J. Scientific Computing*, vol. 34, no. 1, 2012.

[16] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, "Minimizing communication in numerical linear algebra," *SIAM J. Matrix Analysis Applications*, vol. 32, no. 3, pp. 866–901, 2011.

[17] ——, "Graph expansion and communication costs of fast matrix multiplication," *J. ACM*, vol. 59, no. 6, p. 32, 2012.

[18] M. Christ, J. Demmel, N. Knight, T. Scanlon, and K. Yelick, "Communication Lower Bounds and Optimal Algorithms for Programs That Reference Arrays Part 1," UC Berkeley, EECS Technical Report EECS–2013-61, May 2013.

[19] E. Solomonik, A. Buluc, and J. Demmel, "Minimizing communication in all-pairs shortest paths," in *IPDPS*, 2013.

[20] J. E. Savage and M. Zubair, "Cache-optimal algorithms for option pricing," *ACM Trans. Math. Softw.*, vol. 37, no. 1, 2010.

[21] V. Elango, F. Rastello, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, "Data access complexity: The red/blue pebble game revisited," OSU/INRIA/LSU/UCLA, Tech. Rep., Sep. 2013, oSU-CISRC-7/13-TR16.

[22] S. Rusu, S. Tam, H. Muljono, D. Ayers, J. Chang, R. Varada, M. Ratta, and S. Vora, "A 45nm 8-core enterprise xeon processor," in *Solid-State Circuits Conference, 2009. A-SSCC 2009. IEEE Asian*, Nov 2009, pp. 9–12.

[23] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches," HP Labs, Tech. Rep. HPL-2009-85, 2009.

[24] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, December 2009, pp. 469–480.

[25] K. Czechowski, C. Battaglino, C. McClanahan, A. Chandramowlishwaran, and R. Vuduc, "Balance principles for algorithm-architecture co-design," in *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, ser. HotPar'11, 2011, pp. 9–9.