ABSTRACT
3D transpose is an important operation in many large scale scientific applications such as seismic and medical imaging. This paper proposes a novel algorithm for fast in-place 3D transpose operation. The algorithm exploits SIMD multicore architecture with software managed memory hierarchy. Such architectural features are present in the next generation processors, such as the Cell BE processor. The algorithm performs transposition at two levels of granularity: at coarse level, where logical transposition is done by merely transposing the address map at each access; and at a fine grain level, where transposition is done at the physical level. Such mix combines the benefits of allowing for fast on-chip bandwidth by providing for large transfer sizes, and at the same time allows for fine-grain SIMD operations. The transfer rate is further enhanced by allowing for batch transposing spatially joined data along a major axis. Results on the Cell BE processing show substantial utilisation of on-chip communication bandwidth, and negligible processing time.

1. INTRODUCTION
Transposing 3D data volumes is a fundamental primitive operation used in many multi-dimensional data processing applications. Examples include seismic, medical imaging, media industry for 3D TV, biomedical, and 3D FFT applications. 3D FFT is in turn used in solving many mathematical problems including Poisson’s equation in cylindrical coordinates [2], partial differential equations and X-ray diffraction data processing.

The transpose operation could conceptually be implemented by merely reordering access indices. However, that does not achieve high performance on memory systems. Memory systems are hierarchically organised with data managed in blocks. A block is a contiguous chunk of data of sizes big enough to hide access overheads and achieve high bandwidth of the underlying communication interconnect; the block is also small enough to allow for higher usage of the block’s data. Therefore the reordering of access indices, for implementing the transpose operation, effectively requires a block size of one. That obviously increases overhead per access and does not achieve high bandwidth. However, it has the advantage that no physical data reorganisation is required.

Transpose operation is typically implemented as physical data reorganisation. However, such operation would require accessing the memory in non-contiguous manner, not achieving high memory bandwidth during the transpose. Moreover, an out-of-place implementation is typical, requiring allocating two 3D volumes of data in memory, increasing the memory requirements in the underlying computer system. Such approach is beneficial if such transposition overhead is amortised by repeatedly accessing the transposed data afterwards.

This paper proposes a novel approach [3] that allows for sparse, infrequent access to the transposed data. Moreover, it achieves comparable speeds for repetitive access to that of the physical transpose approach. The novel approach performs transpositions at two levels of granularity, using both physical and logical mappings. Physical mapping happens at a relatively fine granularity allowing for high memory bandwidth speeds (and high computation bandwidth for SIMD processors). While logical mapping happens at a coarser granular achieving low overhead mapping. Thus our approach essentially combines the benefits of both approaches discussed above.

Our approach targets multicore processors with software managed memory hierarchy such as the Cell BE processor [5]. Such processors are currently widely used in industries such as media, medical imaging and seismic, where 3D transpose operation is a necessity.

This paper is organised as follows: Section 2 provides a survey of related research to our work. Section 3 describes our novel transpose algorithm. Section 4 discusses experimental results of our new algorithm. And finally Section 5 provides conclusions and future work of this work.
for multi-dimensional transpose, where the transposition is
Wapperoma et al. [6] propose a parallelisation algorithm for
physical transpositions. Moreover, it performs
transposes 'in-place' using hybrid/integrated logical
and communication among processors. That contrasts with our
algorithm where the 3D volume is decomposed into smaller
sub-volumes, each processed by a processor. Moreover, no
logical transposition is performed.

Wapperoma et al. [6] propose a parallelisation algorithm for
3D FFT. That involves a transpose operation. The proposed algorithm divides the 3D data into planes, splitting each plane
among multiple processors. That contrasts with our algorithm where the 3D volume is decomposed into smaller
sub-volumes, each processed by a processor. Moreover, no
logical transposition is performed.

It is also worth noting that there are other approaches for
distributed memory architecture such as Choi et al. [1] and
Cohl et al. [2], but they are not relevant to the target do-
main of shared-memory architecture typical in the multicore
paradigm.

3. TRANSPOSE ALGORITHM

The transpose algorithm is concerned about transposing a 3D volume of data. We refer to that volume as 'cuboid' as the dimensions need not be equal. For the purpose of this description, we define the cuboid to have dimensions $L \times M \times N$ along the cuboid's orthogonal axes $X$, $Y$, and $Z$.

The transpose operation changes axes order of the cuboid. As an example, if the $X$ and $Y$ axes are swapped, an element $(x, y, z)$ would map into $(y, x, z)$ in the transposed volume. We therefore use the axes order to describe the transpose operation. We use a three-letter string formed by concatenating axes names to describe the transpose. For example, the transpose `YZX' will exchange axes $Y$ and $Z$ in the original `ZYX' order. For the sake of simplicity we refer to the original axes order as 'baseline' axes order, and refer to each axis as 'major', 'middle', and 'minor', corresponding to $Z$, $Y$, and $X$ axes respectively. Figure 1 shows the use of such naming while performing the transpose `YZX'.

For the underlying computer system, we assume a multicore
processor with two levels of memory hierarchy: the first is
a private local storage, and the second is a global shared
memory. The memory system is software managed, requiring
explicit local to global memory transfer operations. A
typical example of such system is the Cell BE processor [5].

The cuboid is stored in memory in typical major-middle-
minor axes order. For the purpose of logical transposition
(described below), we define a 'cube' to be the primitive unit of logical access. The cube is of dimensions $p \times p \times p$.
The cube side length $p$ should divide each of the cuboid
dimensions $l$, $m$, and $n$. Thus we may refer to the cuboid as a
3D volume of cubes, with dimensions $\frac{l}{p} \times \frac{m}{p} \times \frac{n}{p}$.

We refer to a row (in the minor axis direction) of cubes as
'bar'. We also qualify middle-minor planes as 'faces' (for
cuboid, bars, and cubes).

To access an element $(x, y, z)$ we would require:

1. Determining the cube index $(i, j, k) = ([x/p], [y/p], [z/p])$.
2. Accessing the data element $(i \mod p, j \mod p, k \mod p)$
   inside the cube.

The first step determines the cube indices, and the second
step determines intra cube offsets. Such access mechanism
allows for simple logical transpose at the cubes level, and
physical transpose at the cube elements level.

The following algorithm transposes the whole cuboid:

**Algorithm 1. Transpose cuboid at direction 'axes order'**

1. **Chop the cuboid into bars of dimensions $p \times p \times L$.**
2. **Distribute the bars evenly among processors.**
3. **Each processor loads bars into its local memory.**
4. **Each processor chops back the bars into cubes into its local memory.**
5. **Each processor transposes each cube into its local memory in the direction of 'axes order'.**
6. **Each processor combines back the bar from local memory into the global shared memory, in its original place.**

The algorithm starts chopping the cuboid into bars of equal
dimensions, $p \times p \times N$, (step 1). The reason for that is
each bar’s minor-middle planes are stored contiguously inside
global memory (stride-1 order). Such order is much more
efficient in data transfer from global memory to local
memories; in current multicores, the larger the stride-1 ac-
cess pattern is, the higher the achieved memory bandwidth.
The chopping operation is trivial, and thus do not increase
execution time.

After obtaining bars, the algorithm uses the parallel proces-
sors to speedup the transpose operation. That is achieved
by distributing bars to processors (step 2). This step is also
trivial.
Each bar is read into local memory

Each cube is transposed

Each bar is chopped into cubes

The cubes are combined to form bars

The cubes are stored back into memory forming transposed cuboid

Figure 2: The first five steps of Algorithm 1

The core work starts at step 3, where each processor transposes its share of bars. Each bar is loaded from global memory and stored into processor’s local memory. The load operation is fast due to bar organisation as mentioned above.

Each processor then chops the bars into cubes (step 4). That operation happens in local memory and in parallel. Moreover, memory transfer and processing happens in parallel, thus chopping and loading of bars could be overlapped. Therefore the chopping operation does not introduce overheads.

Each processor will now have a current bar chopped into cubes. It will then transpose each cube, in local memory (step 5). Steps 1 through 5 are illustrated in Figure 2.

Finally, each processor combines transposed cubes into bars (step 6), then stores back the bar into its original position. Those operations are the opposite to the bar reading and chopping operations (steps 1 to 5). Figure 3 illustrates the write operation.

Upon finishing execution, data inside each cube will be transposed, but not cube locations. Post-transpose programs that access the cuboid need to use a logical mapping to access a particular cube. Such mapping is trivial (merely requires re-ordering cube coordinates as per transpose string) and could also be provided by a simple cube mapping function, or performed directly by the programmer. Moreover, the programmer can use the same bar access step, defined above, for accessing many cubes at once, saving memory transfer time (if access pattern permits).

Therefore, as an application programming interface (API), the transpose algorithm provides the following functions:

1. GetCube(i, j, k, g_handle, l_handle) This function loads the cube with coordinates (i, j, k) into local storage. The ‘g_handle’ is a global handle in the shared memory, it stores information about the cuboid such as its dimensions. The ‘l_handle’ is the local cube handle, it specifies where the cube is stored, and its dimensions.

2. PutCube(i, j, k, g_handle, l_handle) This function stores a cube defined by ‘l_handle’ into the global cuboid defined by ‘g_handle’ at cube index (i, j, k).

3. GetBar(j, k, g_handle, l_handle) This function loads bar with coordinates (*, j, k) from the global cuboid defined by ‘g_handle’ into local storage bar defined by ‘l_handle’

4. PutBar(j, k, g_handle, l_handle) This function stores a bar defined by ‘l_handle’ into the global cuboid defined by ‘g_handle’ at bar index (*, j, k).

5. TransposeCube(l_handle, axes order) This function transposes the cube specified by ‘l_handle’ according to ‘axes order’ parameter.

Transposing a Single Cube

This subsection provides details about cube transpose operation (step 5) of Algorithm 1. This subsection is a generic description for all possible transposes that could be performed on a p x p x p cube. Vectorisation is used to achieve high performance of the element displacement.

Possible transposes for a cuboid are (note that the trivial no transpose case is omitted):

1. cw = clockwise rotation (Z Y X → X Z Y).

2. ccw = counter clockwise rotation (Z Y X → Y X Z).

3. xy = XY swapping (Z Y X → Z Y X).

4. xz = XZ swapping (Z Y X → X Z Y).

5. yz = Y Z swapping (Z Y X → Y Z X).

For the p x p x p, Z Y X cube, the cube elements for all x values for a given z and y are defined as yz cube row. We always swap only two adjacent axes at each single step. So the swap is either between major and middle axes, or middle and minor axes. Thus a maximum of three swap operations are required for any transpose.

Major-middle swapping is performed through reordering of the cuboid rows. So each yz row will be swapped with yx row. This swapping is performed through simple out of place memory copying from the source to a temporary destination (for the Cell BE, this memory copy and swap process is performed using SPU C intrinsics (load/store) to achieve high performance, as a row fits in a single 128 bit register). Middle-minor swapping is basically a 2D transpose
In order to achieve any one of the above mentioned 3D transposes, the transpose process involves three steps all utilising local memory:

1. Load a cube into a temporary space from a bar.
2. Perform face transpose on each of the p faces of the $p \times p \times p$ cube in the temporary space.
3. Store cube from temporary space into the bar.

Steps 1 and 3 may or may not involve major-middle axis swapping depending on the required transpose (which is one out of five possibilities as mentioned above). Also step 2 may or may not be performed based on the requested transpose as well.

The algorithm below handle the first case (clockwise rotation), the remaining 4 cases are presented in Appendix A.

Algorithm 2. Clockwise rotation – $cw(ZYX \rightarrow XZY)$

1. Load a cube into temporary space with no major-middle swapping ($ZYX \rightarrow ZYX$).
2. Perform face transpose on each of the $p$ faces of the cube in the temporary space. This involves middle-minor swapping ($ZXY \rightarrow ZXY$).
3. Store the cube from temporary space into destination bar, at local memory, with major-middle swapping ($ZXY \rightarrow XZY$).

4. EXPERIMENTAL RESULTS

To assess the performance of our transpose algorithm, we consider the test case of transposing a large cuboid. Such case would provide an application independent measure of our algorithm. And since transpose operation is generally memory bound, we use the metric ‘achieved memory bandwidth’ (AMB) to measure how well the algorithm performs.

The achieved memory bandwidth, ‘AMB’, is calculated by:

$$AMB = \frac{2 \times \text{Cuboid Size}}{\text{Total Execution Time}}$$

We multiply the ‘Cuboid Size’ by 2 as the transpose test case requires reading the whole cuboid and writing the new transposed one.

The algorithm is implemented on a Cell Blade server. The server has two Cell processors each capable of a theoretical maximum memory bandwidth of 25 GB/sec. The server has five 512 Mbyte memories each is associated with a Cell processor.

We have fixed the cube size to $4 \times 4 \times 4$, to fit with the Cell BE’s 128 bit registers and SIMD operations. It is worth noting that increasing the cube side length cubically increases volume and hence local memory allocated space; whereas decreasing the cube side length decreases read/write speeds and ultimately the degree of SIMD processing possible.

Our rationale in choosing cube dimension is to choose a suitable size with respect to SIMDisation degree of the underlycing architecture. The sizes should allow for efficient vectorisation of the transpose operation and possibly the post-transpose operations. We then rely on bars to achieve high communication speeds.

We also fix the number of SPUs to 8 processors, thus measuring the performance of a single Cell processor.

Table 1 lists the results we achieved using our transpose algorithm. The first column (Cuboid dimension) gives the dimensions for the currently transposed cuboid. The cuboid is a 3D volume of 4 byte elements. The second column (Config) states whether or not the two Cell processors and memories are used. For the two Cell processors, the SPUs are distributed among the two Cells as well as cuboid data (among memories). The third column (Execution Time) gives the total execution time for the whole transpose operation over the 8 SPUs. The reported numbers are the average of 1000 runs (to minimise experimentation errors). The fourth columns (AMB) reports our achieved memory bandwidth metric.

For the first cuboid ($256 \times 512 \times 512$) we achieved memory bandwidth of 19.07 GByte/sec, which is 76% of the maximum possible memory bandwidth. We also achieve similar results using two Cells. However, for larger cuboids we achieve memory bandwidth higher than 25 GBytes/sec. The reason for which is that the cuboid size is allocated at the two memories, and the Cell processor exploits the double bandwidth available.

From the results it is clear that our algorithm utilises most of the memory bandwidth, that indicates that processing time is almost overlapped with memory communication time. However to more accurately assess processing time, we repeated the experiment, turning off any processing routines and allowing only global-local memory transfers. Table 2 reports the percentage of decrease in execution time when processing is turned off.

It is interesting to show that execution time increases in the first test case (with restricting one Cell processor and memory); processing spaces memory transfers to achieve higher memory bandwidth. Whereas for the all other cases (two Cell processors and memories) execution time decreases with the increase of the cuboid size. That indicates that memory transfer scales better with increasing cuboid size and hence transfer block size. However, in all cases execution times are
Table 2: Percentage of decrease in execution time when processing is turned off

<table>
<thead>
<tr>
<th>Cuboid dimensions</th>
<th>Config</th>
<th>Percentage of decrease</th>
</tr>
</thead>
<tbody>
<tr>
<td>256 × 512 × 512</td>
<td>one cell</td>
<td>-23.4%</td>
</tr>
<tr>
<td>256 × 512 × 512</td>
<td>two cells</td>
<td>0.2%</td>
</tr>
<tr>
<td>348 × 512 × 512</td>
<td>two cells</td>
<td>11.6%</td>
</tr>
<tr>
<td>512 × 512 × 512</td>
<td>two cells</td>
<td>22.3%</td>
</tr>
</tbody>
</table>

increased by a maximum of 22.3%, showing that memory is the dominant factor.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a novel 3D transpose algorithm. The algorithm performs transpose at a logical and a physical level. Such approach allows for exploiting available on-chip memory bandwidth in multicore processors, with software managed memory. Besides, it allows for utilising SIMD instruction on such processors. The algorithm also allows for lazy transpose operation, making it applicable to sparse volumes of data.

Experimental results verify the utility of our algorithm on the Cell BE processor. Results shows that more than 70% of maximum bandwidth is achieved.

Future work will consider integrating our algorithm with real applications. And also for providing for implementations of higher dimension transpose operations.

6. REFERENCES


APPENDIX

A. CUBE TRANSPPOSITION DETAILS

In this appendix we provide the details for the remaining 4 transpose directions.

Algorithm 3. Counter clockwise rotation – cw (ZYX → YZX)

1. Load a cube into temporary space with major-middle swapping (ZYX → YZX).

2. Perform face transpose on each of the p faces of the cube in temporary space. This involves middle-minor swapping (YZX → YXZ).

3. Store the cube from temporary space into destination bar (at local memory) with no major-middle swapping (YXZ → YZX).

Algorithm 4. XY swapping – xy (ZYX → ZXY)

1. Load a cube into temporary space with no major-middle swapping (ZYX → ZYX).

2. Perform face transpose on each of the p faces of the cube in temporary space. This involves middle-minor swapping (ZYX → ZXY).

3. Store the cube from temporary space into destination bar (at local memory) with no major-middle swapping (ZXY → ZXY).

Algorithm 5. XZ swapping – xz (ZYX → XYZ)

1. Load a cube into temporary space with major-middle swapping (ZYX → YZX).

2. Perform face transpose on each of the p faces of the cube in temporary space. This involves middle-minor swapping (YZX → YXZ).

3. Store the cube from temporary space into destination bar (at local memory) with major-middle swapping (YXZ → YZX).

Algorithm 6. YZ swapping – yz(ZYX→ YZX)

1. Load a cube into temporary space with major-middle swapping (ZYX → YZX).

2. Do nothing. No middle-minor swapping (YZX → YZX).

3. Store the cube from temporary space into destination bar (at local memory) with no major-middle swapping (YXZ → YZX).