Final Report

2D AUTOCORRELATORS AND LTSA

ENCE 463: EMBEDDED SOFTWARE ENGINEERING

14/10/2013

Ian Glass (igl14)

Wayne Laker (wal33)

1. INTRODUCTION

The purpose of this document is to provide an extensive overview of the design and performance of a bank of continuous 2D spatial autocorrelators, implemented on NVidia GPUs. The bank will be installed in a time machine, based on a prototype designed by Abe and Aaron and on Kurt Gödel's time loop solutions of general relativity. Complementary to the auto-correlator bank is a Labelled Transition System Analyser (LTSA), to model the Finite State Processes (FSP). A University of Canterbury Dynamic Distributed Data Processing (UC D3P) model is modelled and assessed using LTSA to ensure that no deadlocks or race conditions can occur.

2. CUDA

2.1. GPU AND CPU INTERACTION

Efficient implementation of computer hardware requires extensive use of both the Central Processing Unit (CPU) and Graphics Processing Unit (GPU). Such cross-coupled functionality produces complications including synchronization, coupling and cohesion, which in time critical systems becomes complex at both the software and hardware level. All real computer processes requires some form of master-slave collaboration and as such considerations must be made preceding software implementation. For the designed autocorrelator bank it was deemed more appropriate to allocate the CPU as the master *device* and drive the GPU as a slave, which are quite appropriately termed the *host* and *device* respectively. The GPU

cannot access system RAM directly and data must be copied back and forth between the *host* and *device*. This means the CPU possesses direct access to system RAM, which was the driving factor behind master selection.

2.1.1. Graphics Processing Unit

GPUs are specifically designed to efficiently perform the kinds of calculation required for 3D graphics. Often the calculations are simple and numerous; however they must be performed in a short time span, stimulating the need for concurrency. To achieve this, GPUs are composed of a large collection of modestly powerful cores, capable of concurrent operations with core-core interaction through internally shared memory.

2.1.1.1. Threads, Thread Blocks, the Grid and the Kernel

The *kernel*, the highest level software implementation to the GPU, is a CPU executable 'function', which provides a method of CPU-GPU synchronization. Generally speaking, the *kernel* is instantiated with a generic code to be executed in parallel multiple times. Succeeding CPU prompted execution of the *kernel*, the *grid* is initialised with pre-determined constituent thread *blocks*, as shown in Fig. 1. A *block* is an array of threads executed in parallel, capable of inter thread communication through a shared memory structure, where a thread is an instance of a single execution of the *kernel* code. All GPUs are restricted by a maximum *block* size, where modern thread blocks are limited to either 1024 or 512 threads per *block*. Successful implementation of a *kernel* is achieved by utilizing the current thread and *block* ID, using the `threadIDx` and `blockIDx` indexing commands. Not illustrated in Fig. 1, is the concept of a *warp*, which is defined as a group of threads executing identical instructions on multiple processors in one clock cycle.

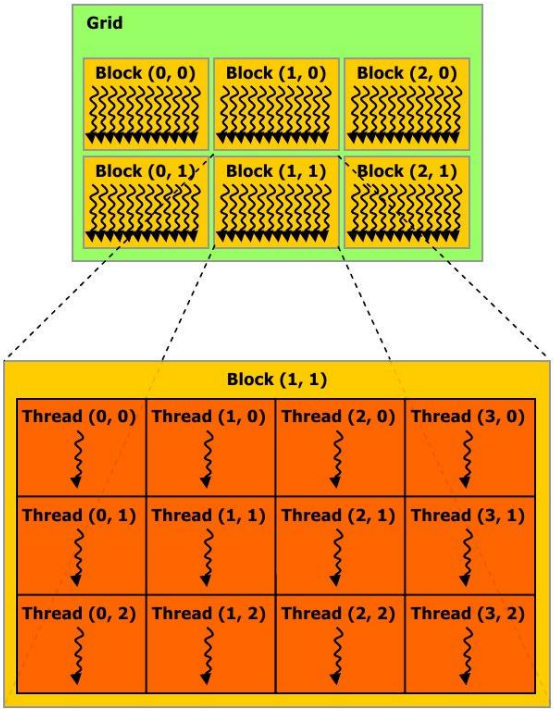

Fig. 1 - Visual depiction of the kernel, grid, blocks and threads.

2.1.1.2. Memory Structure

The internal GPU memory architecture is in many ways, synonymous to the memory structure of a CPU. It contains various registers, local memory, cached memory and global memory, accessible to all peripherals. Fig. 2 shows the internal memory architecture of a GPU including GPU-CPU interaction, where both the CPU and GPU are coupled through the global memory block. Global memory is large in size; however it is much slower to access than other memories, such as the shared memory and internal registers. The shared memory is global to threads within a block, providing enormous speed advantages over coupling threads through the global memory. Thread registers, are the fastest (red) memories available within a GPU, however they are only accessible from the respective thread. Finally the local (or private) memory is a provision to mitigate the detrimental effects of register spilling, however like global memory it is slow (yellow) compared to registers.

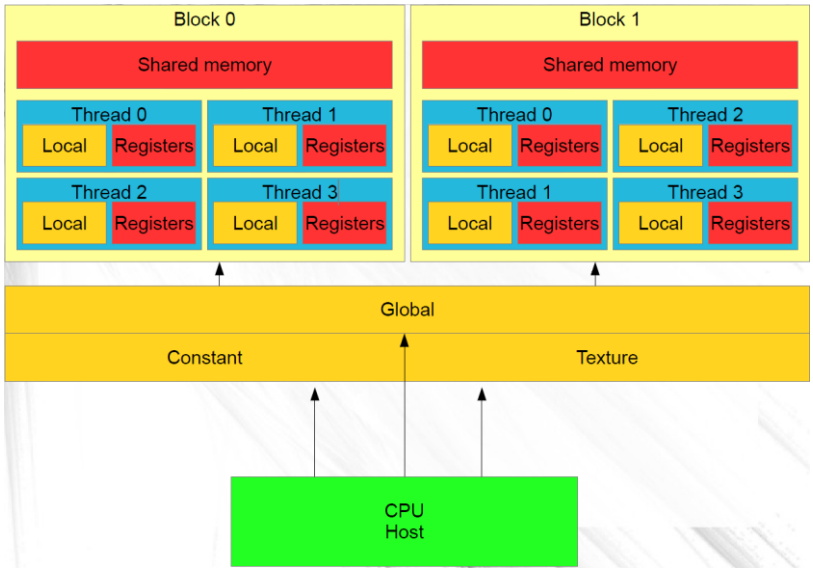

Fig. 2 - Internal GPU memory structure with host interaction.

2.1.2. Central Processing Unit

A CPU is designed for fast, sequential operations, which in contrast to the GPU, utilizes a small collection of very powerful cores. Specifically, a CPU is optimized for low-latency access to cached data sets. Intricate interaction between the CPU and GPU requires some form of synchronisation and memory sharing. Chronologically, the CPU creates a pointer to an allocated section of *host* memory. The section is then filled with relevant data and copied into the *device* global memory. Following this, the *kernel* is initiated by the CPU, providing a pointer to the section of *device* memory which was copied to, allowing the GPU to index and utilize it. Respective data is then moved to thread registers, where the *warp* is executed to perform intended computation. Once this is complete (with appropriate use of thread synchronization), memory is relocated back to global memory, where it can be copied back to the *host* using and identical process to the above memory copying process.

2.2. MATHEMATICS AND ALGORITHMS

2.2.1. Main Module

The main module provides the user with a facile function called 'run_programs', which takes an input matrix, dimensions and three pointers to store the results of an FFT on the *device* and convolution on the *device* and *host*. Complementary to the output results, the processing time for each method is displayed on the terminal as a unit of milliseconds. Users are not required to deal with more intricate details, such as *block* and *grid* size, as these are autonomously calculated. Autocorrelation can be defined as a cross-correlation of a signal with itself, using 'self-convolution' as shown in Fig. 3. Truncating to one dimension, respective elements in a strip are multiplied and the result is accumulated to produce one element in the auto-correlated matrix.

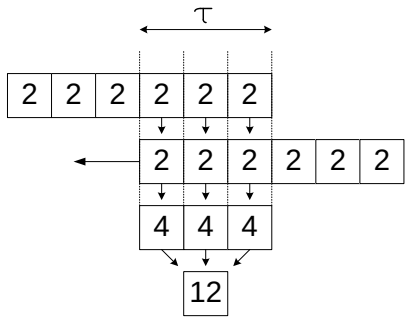

Fig. 3 - Discrete convolution using the sliding strip method.

2.2.2. Fast Fourier Transform on Device

The FFT is a popular spectral estimation technique, employed for its simplicity and computational efficiency. The technique does possess inherent issues, as it is unable to deal well with noisy data, accurately approximate spectra for short time signals, neglects inter-harmonics due to periodicity and provides only frequency and phase information, in contrast to more elaborate techniques such as Prony's method and Pisarenko's harmonic decomposition. The Wiener-Khinchin method is implemented in this design. The method capitalizes on the property that multiplication in the frequency domain results in convolution in the time domain. More specifically, by performing a FFT on the input $X(t)$ and multiplying by its conjugate (equivalent to squaring by the absolute value), the Power Spectral Density (PSD) $S_X(j\omega)$ can be obtained, as shown in Equation 1

$$S_X(j\omega) = (F\{X(t)\})^2 = \int_{-\infty}^{\infty} R_X(\tau) e^{-j\omega\tau} d\tau\,, \qquad (1)$$

where τ is the overlap (shown in Fig. 3.), F is the FFT operator and $R_X(\omega)$ is the autocorrelated output. Following this, the autocorrelation can be determined using Equation 2. By taking the inverse FFT of $S_X(j\omega)$, $R_X(\omega)$ can be found.

$$R_X(\omega) = \frac{1}{2\pi}\int_{-\infty}^{\infty} S_X(j\omega) e^{j\omega\tau} d\omega = F^{-1}\{S_X(j\omega)\}, \qquad (2)$$

Data abstraction is achieved using a CUDA built in data type, called `cuDoubleComplex`. The data type is effectively a *struct* with two *double* type variables x and y for real and complex values respectively. To avoid segmentation faults and memory corruption, *kernel* dimensions must first be calculated, based on the input matrix dimensions. The zero padded matrix size is compared against the maximum thread *block* size and adjusted accordingly should it be exceeded, by `if (size_padded < thread_limit) { num_threads = size_padded`. Next, the *grid* dimensions are calculated by `int grid_size = size_padded/thread_limit+1`. This works by truncating the integer result using a flooring function and adding one to the result to accommodate for the 'remainder' threads. Although this implementation is successful, some computational redundancy will occur in situations for size_padded $\neq n \times$ thread_limit.

There are two main reasons for padding. Consider the sliding strip method depicted in Fig. 3. The final non-zero output occurs when the two opposite ends overlap, corresponding to a length of 2 × width - 1. From the non-padded input signal shown in Fig. 3, there is a clear dimensional mismatch. Consider now performing an FFT of a constant valued matrix, producing only a DC component. By taking the inverse FFT, the output matrix will be representative of a stationary signal. Consider now the transition of the padded matrix, inducing the Gibb's phenomenon. By introducing the transition, non-DC frequencies are induced and a non-stationary output matrix corresponding to the autocorrelated matrix is produced.

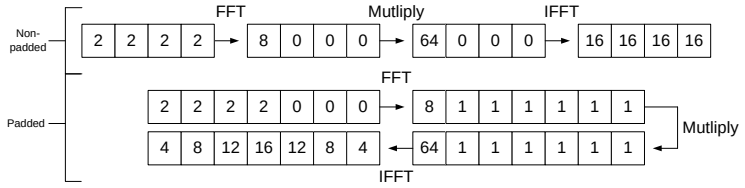

Fig. 4 - Padding versus non-padding preceding and FFT operation.

Zero-padding is performed on the *device*, using the `Pad_FFT` function. The current 1D index is calculated using `int ID = blockIdx.x*blockDim.x+threadIdx.x`, where the current row and column position is calculated using `int j = ID%width_out` and `int i = ID/width_out` respectively. Using this information, the output matrix is zero-padded. Using the `cufftPlan1d` CUDA function, a plan is created for the FFT and inverse FFT. This is succeeded by a call to `cufftExecZ2Z`, which performs a complex to complex FFT of the input data. Matrix multiplication is also performed on the GPU using the `Matrix_Multiply` *kernel*, which takes an input and output matrix as arguments. This function effectively produces the PSD of the input by multiplying its complex conjugate pair. As this operation performs an absolute operation (as shown in Equation 3), the complex component in the *kernel* is automatically set to zero.

$$(a + jb)(a - jb) = a^2 + jab - jab + b^2 = a^2 + b^2, \qquad (3)$$

The inverse FFT is computed using the `cufftExecZ2Z` function; however `CUFFT_INVERSE` is passed in as a parameter. Prior to returning the result to the user, the matrix must be shifted to the correct position. Conceptually, shifting is performed by moving the blocks, illustrated in Fig. 5, diagonally. The current row and column position is calculated synonymous to padding, however and initial check is performed to prevent segmentation faults caused by 'remainder' threads indexing outside of the matrix. Each thread moves the element representative of its calculated ID to the correct position in the output. Finally, the result is copied back to the *device* and returned to the main module.

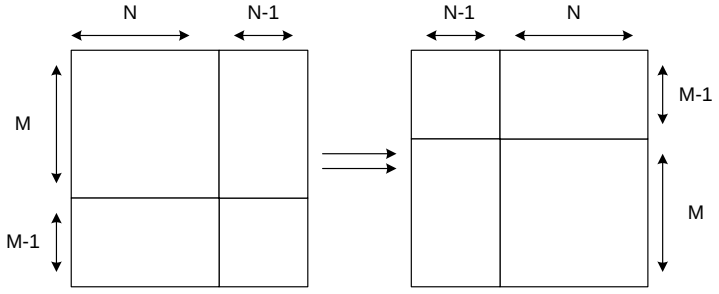

Fig. 5 - Visual conception of shifting.

2.2.3. Convolution on Host and Device

Convolution on both the *host* and *device* is performed using the sliding strip method, outlined in section 2.2.1. Padding is performed using a simple for loop for the *host* and using the `Pad` *kernel* on the *device*. The only difference in padding function between convolution and FFT is the shift required in convolution, where the non-zero data is centred in 2D by applying an offset to the index. Further implementation of convolution was achieved by recycling code obtained from the 'simpleCUFFT' module, provided by the CUDA sample code.

2.3. PERFORMANCE MEASURE

Performance measures were obtained for all three implementations for up to 140 by 140 output matrices and 2500 by 2500 for the FFT implementation, as shown in Fig. 6. An NVidia GTX 560 was used for performance testing and as such it is important to note that results will vary for different GPU architectures. As expected, parallel computing on the *host* incurs serious speed penalties, with an exponential rise in computation speed for small matrices, due to limited core numbers. Interestingly, the *host* is more efficient for very small matrices (10 by 10, not shown), which is attributed to the lack of computationally expensive memory copying operations. Convolution on the *device* bares resemblance to the trend observed in the *host*. This is simply due to the nature of convolution, where each output matric element is an accumulation of the overlap, increasing as a parabolic function of width and height and rapidly increasing the required *grid* size.

The aberrant results of the FFT implementation for small matrices (top left) is a good demonstration of the seemingly stochastic variability of GPU performance. Averaging the plot, reveals a near constant computation time for up to 140 by 140 matrices. Although difficult to interpret, Fig. 6 top right shows the performance gain observed in matrix dimension of prime numbers or powers of two, as well as the decrease in performance as the *grid* size increases.

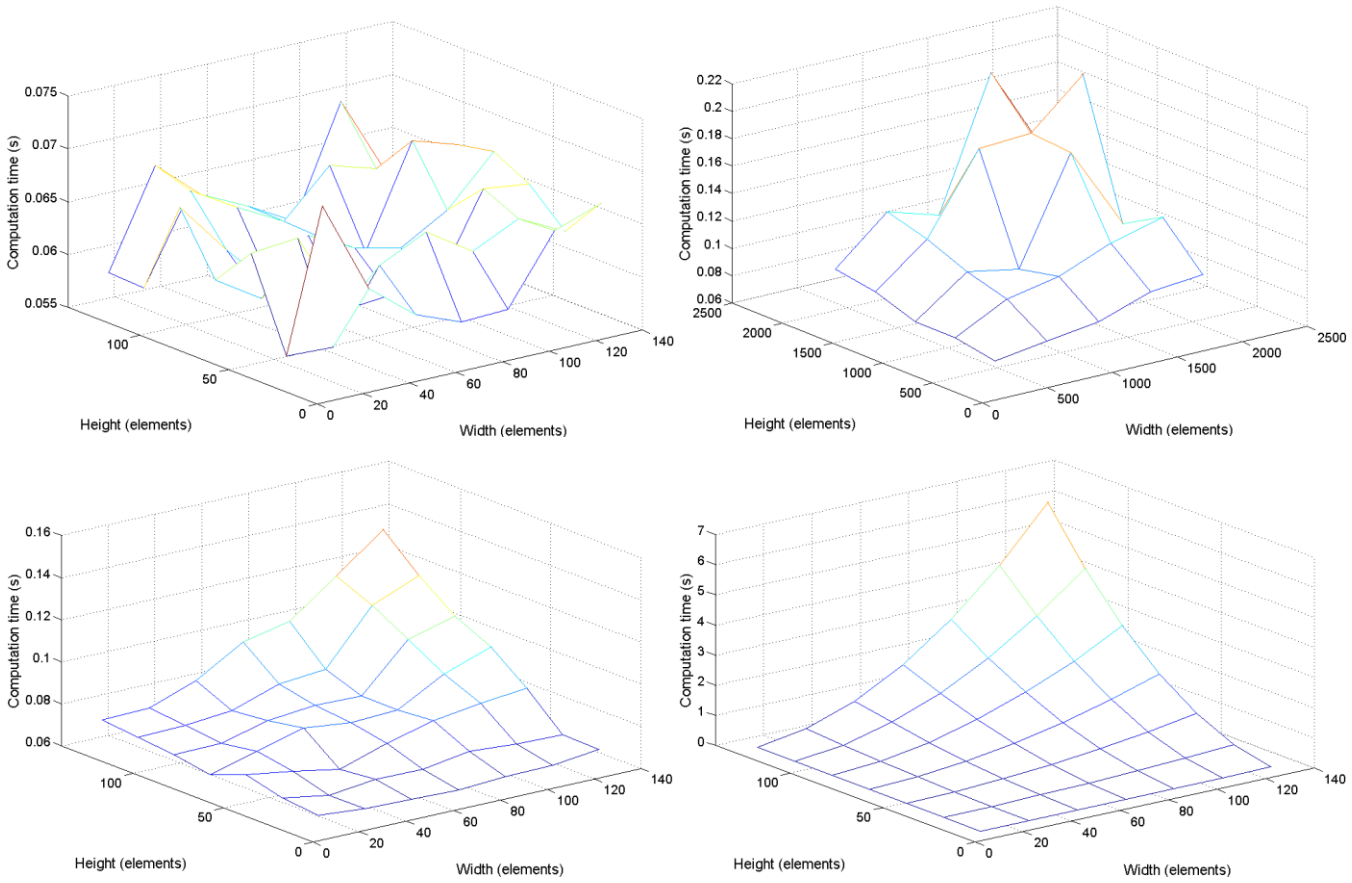

Fig. 6 - 3D graphed performance measure for varying output widths and heights up to 140x140 for correlation on the GPU (bottom left), correlation on the CPU (bottom right), FFT on the GPU (top left) and FFT on the GPU for up to 2500x2500 (top right).

2.3.1. Scalability

Software design should always factor potential for implementation in future products, and as such provisions should be made to accommodate accordingly. To provide scalability in the autocorrelator bank, the software could retrieve the GPU global memory size and warn users of potential memory overflow; however determining the register size to convert the bit value returned by `CudaMemGetInfo` into byes proves difficult. To increase portability to GPUs with varying maximum *block* sizes, the maximum *block* size must be retrieved and replaced with `thread_limit`. For this implementation it was deemed sufficient to `#define` the maximum *block* size to 512, assuming all modern GPUs are capable of at least 512 threads per *block* in the x and y direction. However doing so comes at the cost of decreased performance, as twice the number of *blocks* are required for any one matrix size.

2.3.2. Limits of Operation

Under the previous assumption, the maximum 3D *grid* size is 512 by 512 by 64. To assess the internal efficiency of the FFT implementation, individual *kernel* executions were timed individually for an output matrix size of 2047 by 2047, as shown in Table 1. As conjectured in 2.3, the CPU performance for small matrices is due to the lack of memory copying operation, which is proven to increase computation time significantly. The FFT and inverse FFT *kernels* are also deemed critical, as they all impose similar speed reductions. Interestingly, copying memory from the *host* to the *device* is much slower than copying from the *device* to the *host*, by a factor of three.

Table 1 - Function partitioned computation time for FFT in matrix size of 2047 by 2047.

	Computation time (ms)
Initialization	0.001
Mem copy to device	69.32
Zero-pad	0.319
FFT	67.177
Multiply	0.173
IFFT	67.818
Shift	0.097
Mem copy to host	22.604
Total	227.529

3. FINITE STATE PROCESS MODEL

3.1. OVERVIEW

This section describes use of the Labelled Transition System Analyser (LTSA) to create and check a model for the University of Canterbury's fictional Dynamic Distributed Data Processing (UC D3P) product. LTSA is a verification tool that uses Finite State Processes (FSP) to model concurrent systems. It uses a process algebra notation to describe the states, actions and transitions of a system.

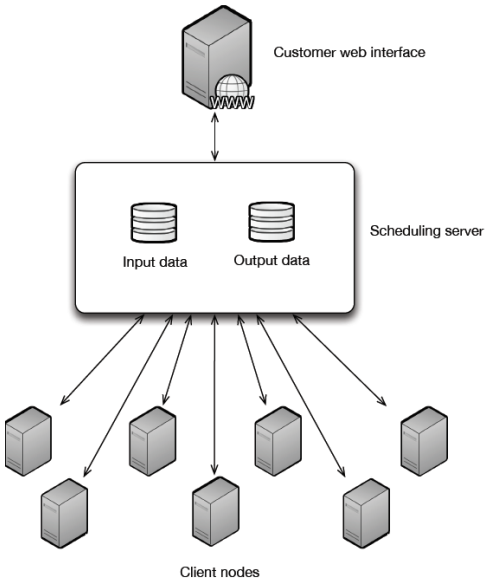

Fig. 7 - UC D3P System Overview.

The task for this part of the project was to design the internal thread model for the scheduling server. The server depicted in Fig. 7 assigns work to an unknown number of client nodes (PCs) and handles concurrent access to its databases.

The design process that was followed is documented. The design started with a simple model and was later expanded to include features such as disconnecting clients, deadlock prevention, race condition tests and multiple jobs. FSP source code that was used is shown in a different font for readability.

3.2. LTSA

LTSA uses process and actions to describe a concurrent program. To model a process than has two states and two actions, the FSP is:

```
PROCESS = (action1 -> action2 -> PROCESS).
```

Fig. 8 shows the corresponding LTSA representation of the process. If the process is in state 0 and action1 is executed, then the process will now be in state 1. The execution of action2 will place the process back in state 0 again.

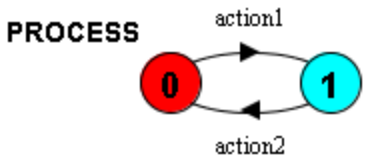

Fig. 8 – Example of an LTSA process with 2 actions.

This series of actions and processes can be extended to model complex concurrent systems.

3.3. DESIGN

3.3.1. Model overview

The scheduling server has the following major components:

- Node interface
- Customer web interface
- Input database
- Output database

The model has following features:

- Critical sections
- Mutexes
- Deadlock avoidance
- Rendezvous
- Non-determinism
- Malicious test process

3.3.2. Node Interface

The node interface is used to interact with the client nodes. A thread is created for each client node that is registered with the scheduling server. A client node must first register with the scheduling server, get assigned a work unit, read input data from an input file, process the data, write output data to an output file and finally be ready for another work unit. A node interface is modelled as a process named *Node* which contains a series of actions that must occur sequentially. Initially *Node* was defined in FSP as:

```
Node = (register -> assign_work -> read -> processing -> write -> Node).
```

This model was sufficient for a single node thread, but multiple node threads running concurrently required a more sophisticated model. The model was expanded to include subprocesses to add more functionality such as disconnecting nodes, critical sections and monitoring job progress.

The final *Node* FSP is:

```
Node = ( register -> READY ),
READY = ( assign_work -> READ_FILE),
READ_FILE = ( read.lock -> rd.start -> rd.end -> read.unlock -> PROCESSING
                | read.lock -> disconnected -> read.unlock -> Node),
PROCESSING = ( processing -> WRITE_FILE ),
WRITE_FILE = ( write.lock -> wr.start -> wr.end -> write.unlock -> INCREMENT
                | write.lock -> disconnected -> write.unlock -> Node),
INCREMENT = (value.lock->value.read[x:T] -> value.write[x+1] -> value.unlock -> READY)
    +VarAlpha.
```

Using the new *Node* model a node registers with the server as before, but then enters the READY state. Once READY, the node gets assigned a work unit. Next the node acquires a read lock before reading the input database file and releasing the lock. After processing the node acquires a write lock, writes the result to the output database file and increments a progress counter before returning to the READY state. The node can optionally get disconnected after receiving a file lock.

3.3.3. Mutex

A mutual exclusion (mutex) model was used to control access to shared resources. A mutex was modelled as a process with lock and unlock actions. A basic mutex is shown below:

```
Mutex = (lock -> unlock -> Mutex).
```

3.3.4. Database Files

Database files can be read and written by multiple client nodes simultaneously so access to them is protected. A critical section was added to the read and write operations to ensure only one process has access at any one time. The critical section for the read operation shown below includes a *rd_count* action which is used by the *NoRace* property later. *rd_count* is incremented when a process enters the critical section, and decremented when it leaves.

```
RD(N=0) = RD[N],
RD[v:Int] = ( rd_count[v] -> (rd.start -> RD[v+1] | rd.end -> RD[v-1]) ).
```

Each *Node* process uses read and write mutexes to enter the critical sections so that only the owner process can unlock the mutex. The read mutex FSP is:

```
ReadMutex = (p[node:Nodes].read.lock -> p[node].read.unlock -> ReadMutex).
```

The *READ_FILE* subprocess in a *Node* includes the read mutex and read critical section actions, as shown below:

```
READ_FILE = ( read.lock -> rd.start -> rd.end -> read.unlock -> PROCESSING
```

The database file write operation is the same as the read process, with *read* replaced with *write*.

3.3.5. Unreliable Client Nodes

The client nodes are only available for work when they are idle. At any time a node can disconnect when a person starts using the PC again. Any locks that have been placed on files must be cleared if a node has disconnected.

An unreliable node that can disconnect was modelled using non-determinism. When a node is ready to read the input file the *Node* process acquires a read lock and performs a file read and then releases the read lock. Alternatively, after a read lock is acquired, the node can be disconnected and then the read lock is removed before program flow returns to the initial *Node* state. In reality the client node can disconnect at any time but the model only handles the case where it disconnects immediately after the lock is acquired to demonstrate the process. The FSP used to model a disconnected client after a read lock is:

```
READ_FILE = ( read.lock -> rd.start -> rd.end -> read.unlock -> PROCESSING                    |
read.lock -> disconnected -> read.unlock -> Node),
```

Fig. 9 shows that from state 2 the *p[1].read.lock* action has two different paths. The *p[1].rd.start* path is the normal program flow, and *p[1].disconnected* path represents the case where the node is no longer available.

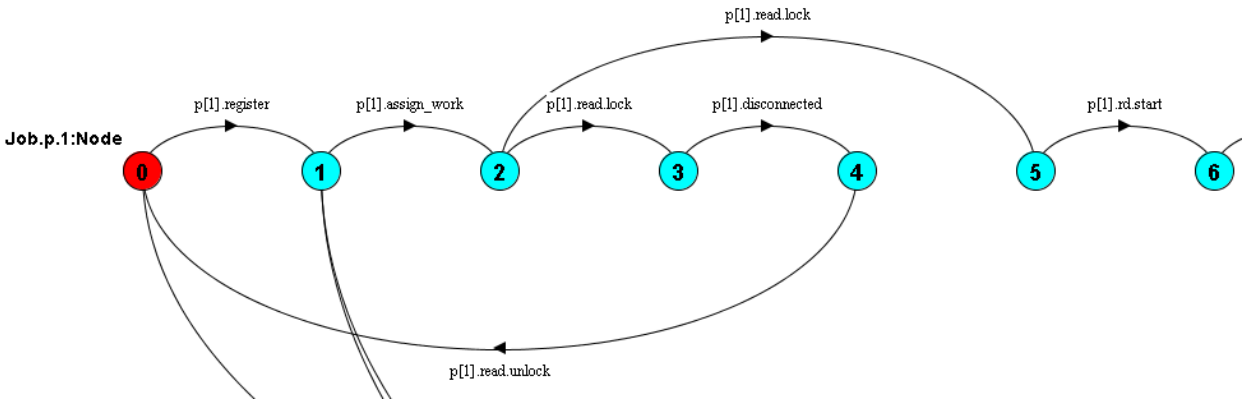

Fig. 9 - Disconnected Client Handling.

3.3.6. Customer Web Interface

The UC D3P system includes a customer web interface which queries the scheduling server for job progress updates. The web interface was modelled as a process that has read only access to a count variable which is updated by a node thread upon completion of a unit of work. The count variable is protected with a mutex to prevent corruption. The web interface FSP is shown below:

```
WEB =(value.read[T]->WEB)+VarAlpha.
```

The nodes update the count variable in the *INCREMENT* subprocess:

```
INCREMENT = (value.lock->value.read[x:T] -> value.write[x+1] -> value.unlock -> READY)
    +VarAlpha.
```

Note that the *WEB* and *Node* alphabets are extended with *VarAlpha* (defined below) which contains actions associated with the count variable.

```
set VarAlpha = {value.{read[T], write[T], lock, unlock}}
```

3.4. MODEL TESTING

The model was tested to ensure that no errors, deadlocks or race conditions occur. It was tested using a variety of parameters including multiple nodes, multiple work units per node, multiple jobs and malicious processes.

3.4.1. Model Safety Check

The model safety check was run using the *Check Safety* command within LTSA. The model was configured with 3 client nodes (including a malicious node p[3]) and 2 work units per node. The program output after compilation was:

```
Composition:
CheckJob = Job.p.1:Node || Job.p.2:Node || Job.RD || Job.ReadMutex || Job.WR || Job.WriteMutex ||
Job.p.3:BadNode || Job.web:WEB || Job.{p.1,p.2,p.3,web}::value:VarMutex.Mutex ||
Job.{p.1,p.2,p.3,web}::value:VarMutex.VAR || NoRace
State Space:
 31 * 31 * 8 * 4 * 8 * 4 * 1 * 1 * 2 * 7 * 1 = 2 ** 24
Composing...
Depth 11372 -- States: 10000 Transitions: 120737 Memory used: 81340K
```

```
Depth 18791 -- States: 20000 Transitions: 248431 Memory used: 117949K
Depth 20940 -- States: 30000 Transitions: 378465 Memory used: 155399K
Depth 13589 -- States: 40000 Transitions: 510135 Memory used: 101434K
Depth 1312 -- States: 50000 Transitions: 635695 Memory used: 138589K
-- States: 51016 Transitions: 648200 Memory used: 177781K
Composed in 408ms
```

The program output after running the safety check was:

```
No deadlocks/errors
```

This result indicates that there are no conditions under which deadlock can occur.

3.4.2. Concurrent Access

A malicious process was added that attempts to break the write file critical section. *BadNode* is a loop which continuously tries to unlock the write file mutex. *BadNode* was unsuccessful in unlocking the write mutex because there is no path that allows it. The *BadNode* FSP is:

```
BadNode = (write.unlock -> BadNode) +{write.lock, wr.start, wr.end, read.lock, rd.start, rd.end,
rd.unlock}.
```

The malicious process was then modified to attempt to write to the file without using the mutex at all:

```
BadNode = (wr.start -> BadNode)
              + {write.unlock, write.lock, wr.end, read.lock, rd.start, rd.end, rd.unlock}.
```

Compilation now reveals a "property NoRace violation", with the following safety check output:

```
Trace to property violation in NoRace:
       wr_count.0
       p.3.wr.start
       wr_count.1
       p.3.wr.start
       wr_count.2
```

As expected, if the mutex is not used then there is a possibility for a race condition to occur because there is more than one process in the critical section at the same time.

3.4.3. Model Alphabet

The final model alphabet (as tested in 3.4.1) is:

```
Alphabet:
{
p[1].assign_work, p[1].disconnected, p[1].processing,
p[1].rd.{end, start}, p[1].read.{lock, unlock}, p[1].register, p[1].value.{lock, read[0..6], unlock,
write[0..7]}, p[1].wr.{end, start}, p[1].write.{lock, unlock},

p[2].assign_work, p[2].disconnected, p[2].processing,
p[2].rd.{end, start}, p[2].read.{lock, unlock}, p[2].register, p[2].value.{lock, read[0..6], unlock,
write[0..7]}, p[2].wr.{end, start}, p[2].write.{lock, unlock},

p[3].rd.{end, start, unlock}, p[3].read.{lock, unlock},
p[3].value.{lock, read[0..6], unlock, write[0..6]}, p[3].wr.{end, start}, p[3].write.{lock, unlock},

rd_count[0], rd_count[1], rd_count[2], rd_count[3], web.value.lock, web.value.read[0..6],
web.value.unlock, web.value.write[0..6],
wr_count[0], wr_count[1], wr_count[2], wr_count[3]
}
```

4. CONCLUSIONS

A bank of continuous 2D spatial autocorrelators was implemented using two different algorithms on an Nvidia GTX 560 GPU and an Intel® Core™ i7 CPU. The first algorithm used convolution directly and the second used FFT, multiplication and inverse FFT. The FFT method was more efficient than the convolution method, especially for larger matrix sizes. The CPU was more efficient at processing small matrices (10x10) due to the overhead involved with copying data to and from the GPU. For larger matrices the GPU was significantly faster because it was able to take advantage of parallel processing. The FFT method using the GPU allowed a 2D correlation of a grid of 2500x2500 elements to be performed in less than 0.2s.

A FSP model was created for the scheduling server in the UCD3P system. The model was designed and tested using LTSA. It was designed to simulate node interface threads and send a webserver progress updates. The design contained various features including critical sections, mutexes, a disconnecting client and a malicious client. The model was tested using multiple jobs and nodes, was found to be robust and did not contain any deadlocks.