\chapter{Optimizations}

\label{sec:optimizations}

\phantomsection

\section{Parallel Input Generation}

\label{sec:parallel\_input\_generation}

\phantomsection

\subsection{Requirements}

\label{subsection:requirements}

\begin{figure}

\centering

\includegraphics[width=0.85\textwidth,keepaspectratio=true,page=1]{tikxinputgeneration.pdf}

\vspace{.2in}

\caption{Flow chart for input generation (continues in \cref{fig:tikxinputgeneration2}).}

\label{fig:tikxinputgeneration}

\end{figure}

\begin{figure}

\centering

\includegraphics[width=0.85\textwidth,keepaspectratio=true,page=2]{tikxinputgeneration.pdf}

\vspace{.2in}

\caption{Flow chart for input generation (continued from \cref{fig:tikxinputgeneration}).}

\label{fig:tikxinputgeneration2}

\end{figure}

We want to calculate the JND of the signal for either a pure tone or a recorded signal, as explained in \cref{subpar:jnd-types}.

We need to measure the JND for multiple $\dlalpha$, so we define a set of powers in dB: $\dalpha[0],\dalpha[1], \dots, \dlalpha$, with

\begin{equation}

\label{eq:delta-alpha-level}

\dlalpha = \dalpha[0] + SL \cdot dBJump

\end{equation}

The series of $\dlalpha$ can be determined at the input level to search for $\JND$ at different resolutions. Testing can be done on the following signals:

\begin{itemize}

\item{Pure tone:} A pure tone is a cosine signal and is defined by its frequency, its amplitude (denoted SL for ``signal level’’), and its length.

\item{Recorded:} A recorded signal can be any signal.

\end{itemize}

Because $\stalpha$ is a noise amplitude, \cref{eq:crlb-approx} requires an AN response to $s(t,\stalpha)$. We thus implement three types of noise:

\begin{itemize}

\item{Quiet:} $n\_{UF}(t) = 0$ for every $0\leq t \leq F\_s T$.

\item{White noise:} $n\_{UF}(t) = \cN(0,1)$ for every $0\leq t \leq F\_s T$ (UF denotes ``unfiltered’’).

\item{Recorded noise:} can be any noise with length $M\leq F\_s T$. If $M < F\_s T$ then $n\_{UF}(t) = 0$ and $M\leq t \leq F\_s T$.

\item{Filtered:} takes white noise or a recorded signal and passes it through a linear filter.

\end{itemize}

If filter noise is not used, then $n(t) = n\_{UF}(t)$

to solve \cref{eq:crlb-approx} for the array of noise levels (NLs) and gains $\stalpha\_{1} \dots \stalpha\_{NL}$.

Therefore, we wish to calculate $\underline{S}$:

\begin{gather} \label{eq:signal-matrix}

\underline{S} =

\begin{Bmatrix}

s(t,\stalpha\_{1}+\dalpha[0]) & \dots & s(t,\stalpha\_{1}+\dalpha[SL]) & s(t,\stalpha\_{1}) \\

\vdots & \ddots & \vdots & \vdots \\

s(t,\stalpha\_{NL}+\dalpha[0]) & \dots & s(t,\stalpha\_{NL}+\dalpha[SL]) & s(t,\stalpha\_{NL})

\end{Bmatrix}.

\end{gather}

\subsection{Normalization}

\label{subsection:normalization}

As shown by \cite{odedst2017}, multiple SPL reference levels need to be tested to numerically approximate the experimental results in \cite{experimentalists}.

If a noise is filtered,

\begin{equation\*} \label{eg:noise-filter}

\begin{aligned}

H\_{Noise~Filter}(z) & = & \frac{\sum\_{j=0}^{hn\_1}b\_j z^{-j}}{\sum\_{j=0}^{hn\_2}a\_k z^{-k}},\\

h\_{Noise~Filter}(t) & = & \mathcal{F}^{-1}\{H\_{Noise~Filter}(z)\},\\

n(t) & = & h\_{Noise~Filter}(t) \conv n\_{UF}(t).

\end{aligned}

\end{equation\*}

Different normalization methods are tested for signal $f$ or noise $n$, which we denote $\signaltonormalize$ with time length $T$. The process is defined as $\signaltonormalize\_{Base~Input}(t) = NormalizeInput\bm(\signaltonormalize(t)\bm)$:

\begin{itemize}

\item Normalizing by power gives $\inv{T}\int\_{0}^{T}\signaltonormalize^2(t)\mathrm{d}t$ with $\signaltonormalize(t)$ being the non-normalized signal pressure over time. The input function is

\begin{equation\*} \label{eg:normalize-power}

\begin{aligned}

\signaltonormalize\_{Base~Input}(t) = \inv{\left(\inv{T}\cdot\int\_{0}^{T}\signaltonormalize^2(t)\mathrm{d}t\right)} \signaltonormalize(t).

\end{aligned}

\end{equation\*}

\item Normalize by energy: to test various signals at the same strength, division is applied to the sum of the signal energy:

\begin{equation\*} \label{eg:normalize-energy}

\begin{aligned}

\signaltonormalize\_{Base~Input}(t) = \inv{\left(\int\_{0}^{T}\signaltonormalize^2(t)\mathrm{d}t\right)} \signaltonormalize(t).

\end{aligned}

\end{equation\*}

\item Skip normalization for experimental values determined outside the program:

\begin{equation\*} \label{eg:un-normalize}

\begin{aligned}

\signaltonormalize\_{Base~Input}(t) = \signaltonormalize(t).

\end{aligned}

\end{equation\*}

\item Normalize for power over different lengths, because the signal period effects the JND. This allows us to test if short signals (period $T\_s$) with the same energy as long signals (period $T\_l$) will have the same JND:

\begin{equation\*} \label{eg:normalize-diffrent-period}

\begin{aligned}

\signaltonormalize\_{Base~Input}(t) = \inv{\left(\inv{T}\cdot\int\_{0}^{T}\signaltonormalize^2(t)\mathrm{d}t\right)} \signaltonormalize(t).

\end{aligned}

\end{equation\*}

\end{itemize}

As shown in \cref{subpar:jnd-types}, we need to calculate the signal for the range of $\dlalpha$, so the equations are

\begin{eqnarray} \label{eg:pre-hearing-aid-signal}

f\_{SL}(t) & = & SPL\_{ref} 10^{\frac{\dlalpha}{20}} f\_{Base~Input}(t), \\

n\_{NL}(t) & = & SPL\_{ref} 10^{\frac{\stalpha\_{NL}}{20}} n\_{Base~Input}(t), \\

s(t,\stalpha\_{NL}) & = & n\_{NL}(t), \\

s(t,\stalpha\_{NL}+\dalpha[SL]) & = & n\_{NL}(t) + f\_{SL}(t).

\end{eqnarray}

$SPL\_{ref}$ (\cref{tab:Lambda-parameters}) is the pressure level at 0 dB SPL.

\begin{figure}

\centering

\includegraphics[width=1.0\textwidth,keepaspectratio=true]{figs/Input\_Signal}

\vspace{.2in}

\caption{Example for implementation of \cref{eq:signal-matrix}. Generated input for $\dalpha[SL]$ at 40 and 50 dB SPL and $\stalpha\_{NL}$ for 0 (noiseless) and pure tones at frequencies of 0.5, 1, and 4 kHz.}

\label{fig:input\_signal}

\end{figure}

\section{Optimizing for Kepler Architecture}

\label{sec:optimizing-for-kepler}

Our project was developed from the work of \citealt{Saboddoron20131215}, which was implemented on a GTX590, CC 2.0. Its architecture includes 32 K registers per SM \cref{sec:gpu\_architecture\_generations} to prevent the tail-end effect described in \ref{itm:achieved-occupancy}. The BMV calculation kernel is limited to two blocks per SM, which gives 512 threads with 63 registers, 31.5 K per SM, full capacity. The improvements factors for each configuration are given in \cref{fig:nvidia-pascal-comparable-run} and the registers needed per thread are given in \cref{fig:nvidia-kepler-register-requirements}.

\begin{enumerate}

\item[M1] Kepler, CC 3.0 has 64 K registers, which is twice the number of registers. This allows 4 blocks to run on each SM without registers spilling, as shown by \cref{sec:occupancy-calculator}. Kepler uses the L1 cache for temporary register spilling \cite[G.4.1]{NVIDIA\_Programming\_guide} and is configurable between cache and SHM. Optimizing the cache requires reducing the SHM use to 16 KB per SM; increasing the occupancy to five blocks allows 3.2 KB. The original program required 11 KB of shared memory, even under 4 blocks.

\item[M2] The cache would have 3 KB per block and would have difficulty containing temporary register spills. Examining \cref{eq:pressure-parallel-estimation}, $\underline{A}$, and $P(X\_n,t)$ and denoting the boundary condition \cref{eq:discrete-pressure-zero}by $P(x\_N,t)=0$, we can change all values of $\underline{A}\_u$ to unity and ignore it completely, thereby saving 1028 B. The quantity $\underline{A}\_l$ can be thus be replaced by the formula $n<N-1$, which removes another 1028 B of SHM. The kernel needs to synchronize the memory fence of three arrays in SHM, so $P$ is used to calculate $P^{(j)}$ from $P^{(j-1)}$, $|f(y(t+\deltat),t+\deltat) - f(y(t),t)|$ from \cref{eq:lifchitz-condition} is used to calculate the maximum value, and the third parameter is available from \citealt{Barzelay2011} (p. 37):

\begin{equation}

\label{eq:error-sum}

e = \sum\_{x\_n}\|\dot{\xi}\_{bm}^{(L)}(x\_n,t\_{n+1})-\dot{\xi}\_{bm}^{(L-1)}(x\_n,t\_{n+1})\|\_2,

\end{equation}

to calculate the sum over the dimension $x$. The remaining SHM representing $\dot{\xi}^{(L)}$, $\xi^{(L)}$, $P\_{TM}$, and $K\_{bm}$ from \cref{tab:Lambda-parameters} are replaced by registers. The last parameter stored, $\dfrac{L \deltat}{2}$, needs to be tested between threads so that $\max\_{x\_N}\dfrac{L \deltat}{2} > 1$. \cref{sec:cochlear\_model\_solution\_initial\_condition} replaces it by a register value and uses $\\_\\_syncthreads\\_or$ to synchronize between the block threads without using SHM. Although this increases the register pressure to 62 per thread, the effect is minimal because of the requirement of copying the SHM to register before using the value. This also reduces the use of the SHM to less than 3.2 KB per block and allows us five blocks.

\item[M3] Many of the operations require arithmetic operations on the same constants. The Kepler architecture allows fast access to constant memory \cref{memorytypes} with 64 KB, which allows us to replace most of those calculations with constants, thus reducing both the unnecessary arithmetic operations and lowering the register pressure to 58. The calculation replaces $\alpha\_s/\alpha\_l$, $\deltax^2$, and $\deltax \sigma\_{ow} \gamma\_{ow}$. We also removed previous tests of linearity that were already set and added unnecessary commands to the code.

\item[M4] Note that the oval window parameter $Y\_0$ used to calculate (\cref{eq:pressure-depends-on-time-isolate}

requires $\xi\_{OW}^{(L)}$, $\dot{\xi}\_{OW}^{(L)}$, $\ddot{\xi}\_{OW}^{(L)}$, $\xi\_{OW}^{(L-1)}$, $\dot{\xi}\_{OW}^{(L-1)}$, and $\ddot{\xi}\_{OW}^{(L-1)}$. This calculation applies only to thread 0, which handles the oval window boundary condition calculation and takes only 24 B. The CUDA compiler optimizes the register allocation to satisfy the executive branch; if needed, other registers will be used for different functions on separate branches. This reduces the registers used per thread to 54.

\item[M5] CUDA fuses the multiply-add command, which allows both operations without passing through the registers. Fusing all possible multiply-add commands reduces the registers used per thread to 51.

\item[M6] By setting the launch-bound maximum number of blocks to five, the compiler attempts to reduce register pressure to allow a maximum occupancy of five blocks, or 48 in this case. The other six registers spill to the L1 cache. The occupancy is $256 \times 5 = 1280$ threads, which takes 7680 B in the L1 cache. By increasing its capacity to 48 KB we avoid round trips to the L2 cache due cache misses. An occupancy of five blocks gives a configuration of 7 frequencies, 10 levels of power ($\Delta \alpha$), 3 noise levels, 40 ms for each input (20 ms for each block interval), plus a reference for each noise of $3\times(10\times7+1)\times2=426$ blocks. The GTX 760 has 9 SMs, so handling immediately improves from 36 to 45. The run time goes from 6.08 to 5 s, which is an improvement of 21.5\%.

\end{enumerate}

\begin{figure}[ht]

\setlength{\subfigcapmargin}{.1in}

\centering

\begin{tikzpicture}

\begin{axis}[

ybar =-2cm,

x axis line style = { opacity = 0 },

xticklabel style=

{align=center,text width = 5mm},

axis y line = none,

tickwidth = 0pt,

enlarge y limits = 0.8,

symbolic x coords = {M1,M2,M3,M4,M5,M6},

nodes near coords,

]

\addplot coordinates { (M1,61) (M2,62) (M3,58) (M4,54) (M5,51) (M6,48) };

\legend{Registers Usage per thread}

\end{axis}

\end{tikzpicture}

\caption{Comparison of Kepler registers per thread requirements for implementations M1--M6; from \cref{sec:optimizing-for-kepler}.}

\label{fig:nvidia-kepler-register-requirements}

\end{figure}

\section{Optimizing for Pascal Architecture}

\label{sec:optimizing-for-pascal}

The main GPU structure changes between Kepler \& the Pascal or Maxwell architecture \cref{fig:nvidia-kepler-maxwell-comparison}. Optimizations for higher occupancy do not result in faster algorithms. Owing to register spills, Pascal devices store in the L2 cache as opposed to the L1 cache in the Kepler architecture. We tested multiple algorithms to compare several convergence methods and division and find that the large L2 cache allows us to compile with launch bounds of six blocks per SM, with the register spilling to cache. We use the GTX 1080 Ti with 28 SM and set up a run of four noise power levels (0, 10, 15, and 20 dB) and 25 signals (0--72 dB by steps of 3) on seven different frequencies with each interval set to 40 ms, which makes 1400 blocks on the entire card. This gives 10 blocks per SM on five blocks. We compare the six methods for speed (normalized) and find that,

\begin{figure}[ht]

\setlength{\subfigcapmargin}{.1in}

\centering

\begin{tikzpicture}

\begin{axis}[

ybar =-2cm,

x axis line style = { opacity = 0 },

xticklabel style=

{align=center,text width = 5mm},

axis y line = none,

tickwidth = 0pt,

enlarge y limits = 0.8,

symbolic x coords = {M1,M2,M3,M4,M5,M6,M7},

nodes near coords,

]

\addplot coordinates { (M1,1) (M2,1.66) (M3,1.66) (M4,1.662) (M5,1.655) (M6,1.65) (M7,1.84) };

\legend{Run Time Normalized Improvement}

\end{axis}

\end{tikzpicture}

\caption{Comparison of Pascal Run times for different configurations, each number indicates $\frac{runtime(M1)}{runtime(Configuration)}$. for example, if M1 to 3.52 seconds M2 will take $\frac{3.52}{1.66}= 2~seconds$}

\label{fig:nvidia-pascal-comparable-run}

\end{figure}

for the first six optimizations, only M2 gives significant acceleration (66\%). The CUDA 8 compiler was changed to standard C++ from C. We combined the template functions with a warp shuffle to reduce synchronizations. As noted at M2, we required three aggregations to test (a maximum and two additions). We used the modified warpReduceSum algorithm with the template to get three accumulator operations on each value and, with a block of 256, we reduced to eight the mid values for each function. We ran single-thread synchronization and took three values for each of the lower threads and ran a second partial sum on the remaining eight threads. This reduced the run time by another 10\%, giving a total run-time improvement of 84\%.

\section{Testing Congestion with Nsight}

\label{sec:nsight-for-congestion}

To test the critical bottlenecks when optimizing in \ref{sec:optimizing-for-pascal}, several criteria from \cref{sec:nvidia-nsight} were tested for M5--M7, including \ref{itm:instructions-statistics}, \ref{itm:issue-efficiency}, and total run time for the kernel as measured by Nsight. Because the architecture counters are limited, the Nsight configuration measured for noiseless input at 16--20 levels of signals (depending on the maximum occupancy of the configuration) and at very short interval of 8~ms. We verified against run time for standard run of four noise levels (0, 33, 43, and 53 dB) for seven frequencies and ten levels of $\delta \alpha$ (40~ms). M8 is M6 with the synchronization method of M7.

\begin{figure}[ht]

\setlength{\subfigcapmargin}{.1in}

\centering

\begin{tikzpicture}

\begin{axis}[

ybar,

major x tick style = transparent,

bar width=10pt,

width = 0.85\*\textwidth,

xtick = data,

x axis line style = { opacity = 0 },

enlarge x limits=0.25,

axis y line = none,

enlarge y limits = 0.8,

symbolic x coords = {M5,M6,M7,M8},

nodes near coords,

legend style={

at={(0.4,0.85)},

anchor=south east,

column sep=1ex

}

]

\addplot[style={bblue,fill=bblue,mark=none}] coordinates {(M5,1.35)(M6,1)(M7,1.5)(M8,1.45)};

\addplot[style={rred,fill=rred,mark=none}] coordinates {(M5,0.7)(M6,1.4)(M7,1)(M8,1)};

\addplot[style={rred,fill=ggreen,mark=none}] coordinates {(M5,1.25)(M6,1)(M7,1)(M8,1.25)};

\addplot[style={ppurple,fill=ppurple,mark=none}] coordinates {(M5,0.83)(M6,1.15)(M7,0.96)(M8,1.13)};

\legend{Run Time,Run Time On Nsight,Intervals tested on Nsight,Instructions Per Cycle}

\end{axis}

\end{tikzpicture}

\caption{Comparison of run time on Nsight as debug mode and on Matlab as run time. The number of intervals tested, M5--M8, are described in \cref{sec:optimizing-for-kepler,sec:optimizing-for-pascal}. A maximum of 8 instructions per cycle are allowed given four warps that can issue two instructions per cycle. This is not the case here, because stall issues block warps at around 75\% of the clock cycles. Also note that the Nsight run time is the negative of the predicted test run time.}

\label{fig:nvidia-pascal-nsight-parameters}

\end{figure}

Although Nsight is not indicative of kernel run time, it can be used to analyze stall issues, e.g., for cases without the triple synchronization mechanism (M5, M6). Synchronization is a stalling issue at a rate of 38\% versus 27\% when the mechanism is added.

\section{Optimizing Convergence Parameters}

\label{sec:optimizing-convergence-parameters}

The program is optimized to generate a large database mapping damage profiles (in OHCs and IHCs) to the JND. The optimal upper bounds to speeds are measured based on the Lipschitz criteria \cref{eq:lifchitz-condition} searched by detect, and the run time for the BMV kernel is measured for 100 damage profiles (Cartesian product of OHCs and IHCs by 10\% jumps). Each profile was tested with 4 level of noise (0, 33, 43, and 53 dB).

\begin{figure}[ht]

\pgfplotstableread[col sep=comma,header=false,columns={[index]0}]{summarize\_pure\_tones.csv}\datarow

\pgfplotstableread[col sep=comma,header=false,columns={[index]1,[index]5}]{summarize\_pure\_tones.csv}\datatable

\pgfplotsset{every x tick label/.append style={font=\small}}

\pgfplotsset{every y tick label/.append style={font=\small}}

\setlength{\subfigcapmargin}{.1in}

\centering

\begin{tikzpicture}

\begin{axis}[

xlabel={Logarithm of convergence speed by cm/second}

,ylabel={Run Time Acceleration Factor,\\ Normalized, Unitless}

,ylabel style={align=center,text width=6cm,color=blue}

,xticklabels from table={\datarow}{[index]0}

,width = 0.85\*\textwidth

,xminorticks=false

,xtick distance=1

,xtick=data

]% table

\pgfplotstablegetcolsof{\datatable};

\pgfmathtruncatemacro\numberofcols{\pgfplotsretval-1}

\pgfplotsforeachungrouped \i in {1}{

\addplot[blue] table[x expr=\coordindex,col sep=comma,header=false,y=\i] {\datatable};

};

\end{axis};

\begin{axis}[

,ylabel style={align=center,text width=6cm,color=red}

,ylabel={Precent of JND differences larger than 2dB}

,hide x axis

,width = 0.85\*\textwidth

,axis y line\*=right

]% table

\pgfplotstablegetcolsof{\datatable};

\pgfmathtruncatemacro\numberofcols{\pgfplotsretval-1}

\pgfplotsforeachungrouped \i in {5}{

\addplot[red,mark=\*] table[col sep=comma,header=false,y=\i] {\datatable};

};

\end{axis};

\end{tikzpicture}

\caption{Testing the JND accuracy and algorithm acceleration as function of minimum $\deltat$ with Lipschitz criteria. We tested two criteria: (i) The run time acceleration factor over the base algorithm run time at a convergence speed of $10^{-15} $~cm. (ii) Percent of JND measurements differs more than 2~dB from the reference algorithm of \citealt{Barzelay2011} to avoid large differences.}

\label{fig:test-convergence}

\end{figure}

The testing shows a nonsignificant increase in error when increasing the minimal $\deltat$ for the algorithm (from ${\sim}10^{-15}$ to ${\sim}10^{-8}$) and a speed increase of nearly 30\%.

\section{Optimize Just Noticeable Difference Calculation for Single Tones}

\label{sec:jnd-single-tone-optimization}

Examination of \cref{eq:ra-crlb} shows us that \ac{crlb} is linear in $T$. If we can prove that, for sufficiently large $T$, $\overline{\lambda}$ is independent of $T$ for pure tone input. We can show by numeric evaluation that, after initial conditions pass, $t> T\_{start}$ for every input of the form of $S\_{input}$ and for a frequency that is a multiple of 250 Hz.

It has been shown \cite{cohen2004int,Barzelay2011} that, for gains greater than 10 to 20 dB, the Wenzel--Kramers--Brillouin linear approximation became unreliable near $\omega\_{CF}$ because of backward propagation. We numerically test to obtain the shortest possible optimal interval but vary minimally from similar intervals. We denote the main $Int$ with length $T$ and composed of concatenated $k\_{JND}$ [\cref{eq:time-shortened}] (we round down for noninteger $k\_{JND}$) intervals each of length [\cref{eq:time-shortened}]

\begin{subequations}

\begin{align}

T\_{shortened}=\frac{T}{k\_{JND}}, \label{eq:time-shortened} \\

Int = [I\_1 ,\dots, I\_{k\_{JND}}].\label{eq:intervals}

\end{align}

\end{subequations}

We then process for each interval a sine function of length $t\_{BOP}+T\_{shortened}$ with the tested part of $T\_{shortened}$ instead of $t\_{BOP}+T$ and replace \cref{eq:ra-crlb} with

\begin{equation} \label{eq:substitute-eq17}

CRLB\_{substituted}(\stalpha) = \left\{ \frac{k T\_{shortened}}{\overline{\lambda}(\stalpha)}\left[\frac{\partial \overline{\lambda}(\alpha)}{\partial \alpha}\bigg\vert\_{\alpha=\stalpha} \right]^2\right\}^{-\inv{2}}.

\end{equation}

By substituting \cref{eq:time-shortened,eq:intervals} into

\begin{equation} \label{eq:lambda-substitute-eq17}

\overline{\lambda}(\alpha) = \inv{T\_{shortened}} \int\limits\_{t\_{BOP}}^{t\_{BOP}+T\_{shortened}} \lambda(t,\alpha)dt,

\end{equation}

\begin{figure}[ht]

\setlength{\subfigcapmargin}{.1in}

\centering

\begin{tikzpicture}

% Basic cardboard

\draw[fill=white]

(-5.2,0) rectangle (6.2,1.2);

\foreach \x in {0,1,2}{

\draw[fill=black]

(\x+0.5,0.6) circle (2pt);

}

\draw[fill=black]

(-5.16,0) rectangle (-2.96,1.2);

\draw[fill=gray!20]

(-2.96,0) rectangle (-2.06,1.2) node[pos=.5] {$I\_1$};

\draw[fill=gray!20]

(5.4,0) rectangle (6.2,1.2) node[pos=.5] {$I\_k$};

\draw [decorate,decoration={brace,amplitude=6.3pt}] (-2.95,1.27) -- node[above=2pt] {$T\_{shortened}$} (-2.06,1.27);

\draw [decorate,decoration={brace,amplitude=6.3pt}] (5.4,1.27) -- node[above=2pt] {$T\_{shortened}$} (6.2,1.27);

\draw [decorate,decoration={brace,amplitude=6.3pt}] (-5.16,1.27) -- node[above=2pt] {$t\_{BOP}$} (-2.96,1.27);

\draw [decorate,decoration={brace,mirror,amplitude=6.3pt}] (-2.96,-0.1) -- node[below=5pt] {$Int,T$} (6.2,-0.1);

\end{tikzpicture}

\caption{Division of interval to measure JND. $T\_{shortened}$ is from \cref{eq:time-shortened}, $Int$ is from \cref{eq:intervals}, and $t\_{BOP}$ is described in \cref{fig:time-divisions}.}

\label{fig:interval-division}

\end{figure}

the covariance of $I\_j$ from $I\_k$ is calculated as

\begin{subequations}

\begin{align}

T\_{offset}(j) & = & Tr+(j-1) T\_{shortened}, \\

I\_j-I\_l & = & \lambda(x,t+T\_{offset}(j),\alpha) - \lambda(x,t+T\_{offset}(l),\alpha), \\

var(I\_j,\alpha) & = & \sqrt{\int\limits\_{0}^{T\_{shortened}}\int\limits\_{0}^{L\_{co}} [\lambda(x,t+T\_{offset}(j),\alpha)]^2 dtdx} \label{eq:var-Ik-Ij} \\

& = & \sqrt{\sum\limits\_{t=0}^{T\_{shortened}/T\_s}\sum\limits\_{x=0}^{Sections-1} [\lambda[x,t+T\_{offset}(j),\alpha]]^2 dtdx}, \notag\\

covar(I\_j,I\_k,\alpha) & = & \frac{var(I\_j-I\_k,\alpha)}{var(I\_k,\alpha)}, \\

covar\_{avg}(I\_j,I\_k,\alpha) & = & \inv{k-1}\sum\limits\_{m=1}^{k-1}covar(I\_m,I\_k,\alpha) \label{eq:covar-Ik-Ij}.

\end{align}

\end{subequations}

We use \cref{eq:substitute-eq17} with \cref{eq:ra-crlb} to find the bound

\begin{equation}

\vert CRLB(\stalpha)-CRLB\_{substituted}(\stalpha)\vert \leq covar\_{avg}(I\_1,I\_k,\stalpha),

\end{equation}

so we now can test multiple $T\_{shortened}$ values to find the optimal value.

\usepgfplotslibrary{colorbrewer}

\pgfplotsset{cycle list/Set1-7}

\begin{figure}[ht]

\pgfplotstableread[col sep=comma,header=false,columns={[index]0}]{frequencies.csv}\datalegend

\pgfplotstableread[col sep=comma,header=false,columns={[index]0}]{input\_sizes.csv}\datarow

\pgfplotstableread[col sep=comma,header=false]{mean\_frequencies.csv}\datatable

\pgfplotsset{every x tick label/.append style={font=\small}}

\pgfplotsset{every y tick label/.append style={font=\small}}

\setlength{\subfigcapmargin}{.1in}

\centering

\captionsetup{width=.8\linewidth}

\begin{tikzpicture}

\begin{axis}[

xlabel={Interval length seconds}

,ylabel={Normalized Covariance}

,xticklabels from table={\datarow}{[index]0}

,xtick=data

,width = 0.95\*\textwidth

,cycle multi list={Set1-7}

,legend style={font=\tiny}

]% table

\pgfplotstablegetcolsof{\datatable};

\pgfmathtruncatemacro\numberofcols{\pgfplotsretval-1}

\pgfmathsetmacro\ydataglobal{0} % y coord 1

\pgfplotsforeachungrouped \i in {0,1,...,\numberofcols}{

\addplot table[x expr=\coordindex,col sep=comma,header=false,y=\i] {\datatable};

\pgfplotstablegetelem{\i}{[index]0}\of\datalegend;

\pgfmathtruncatemacro\ydata{\pgfplotsretval} % y coord 1

\global\let\ydataglobal=\ydata;

\ifnumcomp{\ydataglobal}{>}{999}{\global\let\prefixhertz=K}{\global\let\prefixhertz=\empty}

\ifnumcomp{\ydataglobal}{>}{999}{\DIVIDE{\ydataglobal}{1000}{\ydataglobal}}{\empty}

\edef\temp{

\noexpand\addlegendentry{$\ydataglobal \prefixhertz Hz$};

}

\temp

};

\end{axis};

\end{tikzpicture}

\caption{Testing JND accuracy as function of interval length. The $X$ axis is the tested interval length in seconds, the $Y$ axis is the normalized covariance from \cref{eq:covar-Ik-Ij}. Each plot measures for different frequency tone, as described in the legend.\label{fig:test-intervals}

\end{figure}

From \cref{fig:test-intervals}, the optimal $T\_{shortened}$ is 20~ms instead of 0.2~s, and the acceleration is by a factor of $\frac{200ms+T\_r}{20ms+T\_r}=\frac{212\text{ ms}}{32\text{ ms}}=6.625$.

\section{Optimizing Execution Flow}

\label{sec:execution-flow-optimization}

One of our main goals is to improve the run time for calculating BMV, ANR, and JND, after which we also need to present the results. We do this graphically by using Matlab. \citealt{Saboddd2013682} is used to launch the $dos$ \cite{matlab-dos} command that runs the program from the operating system. This method has several drawbacks that slow the program. We upgrade the flow for Matlab to load the $DLL$ from a $.mex$ file. The process is described in \cref{fig:flow-execution1,fig:flow-execution2}. It is a high-level description of the flow; if a box contains a solidus (/), the text before it describes the Sabo et al. version and the text after the solidus describes our version. The stages in light orange are identical for both versions, whereas the older program did not contain ANR or JND calculations. At this level of abstraction they are identical to BMV flow. Two types of identical stages are possible: the first uploads data to the GPU Read Access Memory (RAM) and downloads it to the CPU RAM. The second type is the calculation themselves, which is done on the GPU.

\begin{enumerate}

\item The first change in the flow is described in the purple boxes. Executing the $dos$ command causes the operating system to open a new process for each run, which involves the scheduler \cite{os-scheduling} and creates a stack and heap for the program, which are released at the end of the execution process. Our implementation shares memory space with Matlab, which means that we do not need to allocate and release those resources at each run: Matlab preserves the loaded function in memory after the first run, which saves 0.7 s per run regardless of the size of the input or output.

\item The second change involves the input-output: the older program and Matlab did not share memory space; their only option to communicate was through a hard-disk read-write, which is slow. The differences are noted in the green boxes. The older method required reading data from multiple files, albeit small ones, and writing data, including large files. ANR and BMV output take together 80 MB per second of input at a 20 kHz sampling rate. A typical execution can process up to 20 s of sound at one time. If we want to see the results, our program needs write 1 GB to the disk and Matlab has to read the same amount of data, which takes 2~s (the rest of the program take only 4~s). Reading and writing to the hard disk should be avoided. Our program shares memory space with Matlab and can copy results through C Matrix Api \cite{matlab-c-matrix-api}. A memory copy of 1 GB to a Matlab array takes less than 50~ms, which is 40 times faster.

\item The third change is the result of memory allocation and deallocation. When the program executed from the operating System, it had to acquire and release its own memory or it would become inaccessible. Arrays are allocated both on the GPU for computation and on the CPU for output, which takes approximately 20~s for 1.25 GB. Our program allocates memory only when current buffers are not sufficiently large or when the program runs for the first time, which usually happens relatively few times in normal operation mode (creating a large database involves running thousands of times on the same size input).

\end{enumerate}

%Destroying MexResources...

%clearing 24 params

%Pre delete creator

%AudioLab Init Lambda Calculations values time: 0 (msec)

%AudioLab Init Lambda Calculations memory time: 12521 (msec)

%AudioLab Load physical parameters time: 0 (msec)

%AudioLab copy IHC/OHC time: 0 (msec)

%AudioLab Init IHC Kernel time: 81 (msec)

%AudioLab Run and copy IHC Kernel time: 10562 (msec)

%AudioLab Post Processing Lambda time: 10380 (msec)

%AudioLab Store Lambda array time: 0 (msec)

%AudioLab Free Lambda calculator time: 49 (msec)

\begin{figure}

\centering

\includegraphics[width=0.82\textwidth,page=1]{tikflowexecute.pdf}

\vspace{.2in}

\caption{Flow chart of program.Input and Cochlear solver}

\label{fig:flow-execution1}

\end{figure}

\begin{figure}

\centering

\includegraphics[width=0.84\textwidth,page=2]{tikflowexecute.pdf}

\vspace{.2in}

\caption{Flow chart of program. Output and release of memory.}

\label{fig:flow-execution2}

\end{figure}